UNIVERSITÄT KOBLENZ · LANDAU

Semantics4J – Programming Language Integrated Semantic Queries and Types

Masterarbeit

zur Erlangung des Grades eines Master of Science
im Studiengang MSc Informatik

vorgelegt von

Carsten Hartenfels

Erstgutachter:	Prof. Dr. Ralf Lämmel Institut für Informatik
Zweitgutachter:	MSc. Martin Leinberger Institut für Informatik

Koblenz, im August 2017

Erklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbständig verfasst wurde und ich keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe und die Arbeitvon mir vorher nicht in einem anderen Prüfungsverfahren eingereicht wurde. Die eingereichte schriftliche Fassung entspricht der auf dem elektronischen Speichermedium (CD-ROM).

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Carsten Hartenfels)

Zusammenfassung

Semantische Daten zusammen mit General-Purpose-Programmiersprachen zu verwenden stellt nicht die einheitlichen Eigenschaften bereit, die man für eine solche Verwendung haben möchte. Die statische Fehlererkennung ist mangelhaft, insbesondere der statischen Typisierung anbetreffend. Basierend auf vorangegangener Arbeit an λ_{DL}, welches semantische Queries und Konzepte als Datentypen in ein typisiertes λ-Kalkül integriert, bringt dieses Werk dessen Ideen einen Schritt weiter, um es in eine Echtwelt-Programmiersprache zu integrieren. Diese Arbeit untersucht, wie λ_{DL}s Features erweitert und mit einer existierende Sprache vereinigt werden können, erforscht einen passenden Erweiterungsmechanismus und produziert *Semantics4J*, eine JastAdd-basierte Java-Sprachintegration für semantische Daten für typsichere OWL-Programmierung, zusammen mit Beispielen für ihre Verwendung.

Abstract

Using semantic data from general-purpose programming languages does not provide the unified experience one would want for such an application. Static error checking is lacking, especially with regards to static typing of the data. Based on the previous work of λ_{DL}, which integrates semantic queries and concepts as types into a typed λ-calculus, this work takes its ideas a step further to meld them into a real-world programming language. This thesis explores how λ_{DL}'s features can be extended and integrated into an existing language, researches an appropriate extension mechanism and produces *Semantics4J*, a JastAdd-based Java language semantic data extension for type-safe OWL programming, together with examples of its usage.

Acknowledgements

I would like to thank my supervisors Prof. Dr. Ralf Lämmel and Martin Leinberger for introducing me at length to the topic at hand, pointing me in the right direction, providing examples and feedback, as well as leaving me free reign with everything else. Thanks also to Prof. Dr. Steffen Staab for additional feedback from perspective of a semantic webber and suggesting the tremendously useful signature checking and partial typing features.

Further thanks go out to all the people behind ExtendJ, for providing an amazingly useful and documented basis for my implementation, and the denizens of `#perl6` for being an exceedingly friendly and helpful bunch.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Semantic data is a way to turn knowledge into an operable format. Ontologies in formats such as OWL [MVH$^+$04] or RDF [BG00] are used to describe the conceptualizations of the data and the known facts about their relationships. From this, new knowledge can be inferred via reasoning. There are many varied use cases for semantic data, such as the collaborative knowledge base of Wikidata [VK14], the linking of knowledge as done by the BBC [KSR$^+$09] or the enhanced internet search and personal assistance services of Google and Microsoft.

However, when it comes to integrating this multitude semantic data into a typeful programming language, much of the distinction of the semantic data is lost. When using some interface such as SPARQL [PS08] or OWL API [HB11], individuals are represented as some generic `Value` or `NamedIndividual` instance, which does not represent the actual semantic concept that the individual belongs to. And while approaches exist to map a semantic data model into a class model [KPBP04], the large number of concepts and possible interactions between them makes this infeasible when applied to real-world ontologies. The resulting type error messages may be difficult to understand, as they refer to the generated names for the mapping, rather than the actual names in the ontology.

To address this issue, λ_{DL} has been developed: a typed λ-calculus that integrates semantic concepts as types of the language itself [LLS16]. It allows definition of semantic concept type constraints using a description logic syntax and verifies these types at compile-time by forwarding the checks directly to a knowledge system. Additionally, it uses the same description logic to execute queries and projections, as well as allowing for type distinctions based on specific individuals at run-time.

However, while it proves the concept, λ_{DL} is not useful for practical application. Being just a λ-calculus at its core, it does not provide any interface to the outside, such as user input, output and libraries that real-world languages make available.

That is the problem this thesis intends to address: taking the core ideas from λ_{DL}– semantic concepts as types built from description logic, using a knowledge base for type-checking, integrated queries and projections – and building an extension for a real-world, general-purpose language with them. This tight integration of semantic data into the type system and syntax of the language would allow for more comfortable development of programs dealing with semantic data, leveraging the possibilities of type checking and other static analysis to prevent common errors.

Language extension like that is of course not a new problem. There are various different language workbenches [SSV$^+$], extensible compilers [EH07a, NCM03, Zen04] or type systems [MME$^+$10, SBT$^+$13] and many other technologies that make customizing programming languages possible using well-defined interfaces, rather than making ad-hoc compiler modifications or building custom preprocessors.

There are several questions raised by this endeavor. How can structural semantic concepts as types fit into a hierarchical type system? How can the description logic syntax be integrated into the existing language's syntax without changing its semantics? What is an appropriate language for extension and what technology should be used for doing so? And what advantages over conventional methods of programming with semantic data does this extension bring? Over the course of this thesis, these issues are be addressed, discussed and attempted to be answered.

The work is structured as follows. Chapter 2 lays out the syntax and semantics of λ_{DL}, upon which the rest of the thesis is built. Chapter 3 points out related work with regards to existing practices of programming with semantic data. Chapter 4 examines the design requirements of a semantic data language extension, chapter 5 seeks out an appropriate platform to build the extension on and chapter 6 shows the workings of the eventual implementation. Chapter 7 shows the extension's features in action and chapter 8 concludes with a list of limitations and possibilities for future work.

Chapter 2

Background

In the following sections, the basics of the λ_{DL} language are laid out, with a focus on its semantic data features. The information about the language is taken from the preliminary report on λ_{DL} [LLS16] and the prototypical implementation by Martin Leinberger[1].

All semantic data examples in this thesis, including source code listings, refer to the Wine Ontology[2], published in the W3C OWL Guide [WMS04].

2.1 Description Logic

Description logic (DL) is responsible for the latter part of λ_{DL}'s name. The language uses a kind of DL for specifying its semantic concept types and for describing queries and projections over semantic data. This section describes the theory of the flavor of description logic that λ_{DL} uses, while the next section deals with the practical aspects of using it for programming with OWL data.

A knowledge base's signature is made up of a triple $(\mathcal{A}, \mathcal{Q}, \mathcal{O})$, where $\mathcal{A}$ is a set of concept names, $\mathcal{Q}$ is a set of role names and $\mathcal{O}$ a is set of object names. DL uses interpretation-based semantics. Interpretations $\mathcal{I}$ are pairs consisting of a non-empty set $\Delta^{\mathcal{I}}$ (the domain or universe) and an interpretation function. The function maps each object from $\mathcal{O}$ to an element in $\Delta^{\mathcal{I}}$, and assigns each concept name from $\mathcal{A}$ to a set $A^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and each role name from $\mathcal{Q}$ to a relation $Q^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ [Baa03, LLS16].

The particular dialect of DL that λ_{DL} uses is an *attributive language with complements, nominal concepts and inverse role expressions* ($\mathcal{ALCOI}$). Table 2.1 describes the syntax and semantics of this particular dialect of DL [Baa03, LLS16].

[1] http://west.uni-koblenz.de/sites/default/files/research/software/lambdadl/precompiled.zip, accessed 2017-08-18

[2] https://www.w3.org/TR/owl-guide/wine.rdf, accessed 2017-08-18

Concepts are built from atomic concepts, nominal concepts, $\top$ (everything or universal concept) and $\bot$ (nothing or bottom concept). They can be negated or composed by intersection, union and existential or universal quantification [LLS16].

Role expressions are atomic roles, $\triangledown$ (top role, relating all possible pairs of individuals) and $\vartriangle$ (bottom role, relating no pair of individuals). Role expressions can be inverted. [LLS16, PC12, MPSP$^+$09].

A knowledge base is compromised of a set of terminological axioms (the TBox), which defines the conceptualizations of the data, and a set of assertional axioms (the ABox), which represents actual data as assertions over named individuals [Baa03, LLS16].

The knowledge base can check if concepts are equivalent and if a concept subsumes another. It can also check if an object is a member of a concept and if two objects are equivalent, as well as relate an object to another via a role [LLS16]. From these primitives, operations such as queries and projections can be built.

This thesis elides further discussion on how to construct a knowledge base like that, as building ontologies is outside of its scope.

Expression	**Syntax**	**Semantics**
Everything	$\top$	$\Delta^\mathcal{I}$
Nothing	$\bot$	$\emptyset$
Top Role	$\triangledown$	$\Delta^\mathcal{I} \times \Delta^\mathcal{I}$
Bottom Role	$\vartriangle$	$\emptyset \times \emptyset$
Atomic Concept	A	$A^\mathcal{I} \subseteq \Delta^\mathcal{I}$
Atomic Role	Q	$Q^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$
Nominal Concept	$\{a\}$	$\{a^\mathcal{I}\}$
Negation	$\neg C$	$\Delta^\mathcal{I} \setminus C$
Inversion	R^-	$\{(b,a) \in \Delta^\mathcal{I} \times \Delta^\mathcal{I} \mid (a,b) \in R^\mathcal{I}\}$
Intersection	$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
Union	$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
Existential Quantification	$\exists R.C$	$\{a^\mathcal{I} \in \Delta^\mathcal{I} \mid \exists b^\mathcal{I} : (a^\mathcal{I}, b^\mathcal{I}) \in R^\mathcal{I} \land b^\mathcal{I} \in C^\mathcal{I}\}$
Universal Quantification	$\forall R.C$	$\{a^\mathcal{I} \in \Delta^\mathcal{I} \mid \forall b^\mathcal{I} : (a^\mathcal{I}, b^\mathcal{I}) \in R^\mathcal{I} \land b^\mathcal{I} \in C^\mathcal{I}\}$

Table 2.1 Syntax and semantics of DL expressions.
With definitions $A \in \mathcal{A}$, $Q \in \mathcal{Q}$ and $a, b \in \mathcal{O}$. C and D stand in for concept expressions. R stands for a role expression. [Baa03, LLS16, PC12]

2.2 OWL API for Knowledge Base Access

In practice, a knowledge base is represented as an ontology, for example in the OWL format. As with the theoretical foundation described in the previous chapter, these ontologies contain the conceptualizations and assertions over the semantic data. To interact with this data, a semantic reasoner is used.

In the prototype implementation for λ_{DL}, the OWL API interface [HB11] has been used to provide knowledge base functionality, in conjunction with the HermiT reasoner [SMH08]. It operates on ontologies in various formats, such as OWL and RDF.

While its terminology relating to DL is slightly different due to using OWL's names for them (concepts are referred to as classes and roles are referred to as object properties), it supports all of description logic and reasoning capabilities required for λ_{DL}. See table 2.2 for the equivalences between OWL API and λ_{DL}'s description logic.

Expression	λ_{DL}	**OWL API**
Everything	$\top$	`Thing()`
Nothing	$\bot$	`Nothing()`
Top Role	∇[3]	`TopObjectProperty()`[4]
Bottom Role	$\triangle$[3]	`BottomObjectProperty()`[4]
Atomic Concept	C	`Class(C)`
Atomic Role	R	`ObjectProperty(R)`[4]
Nominal Concept	$\{a\}$	`ObjectOneOf(a)`
Negation	$\neg C$	`ObjectComplementOf(C)`
Inversion	R^-	`ObjectInverseOf(R)`
Intersection	$C \sqcap D$	`ObjectUnionOf(C, D)`
Union	$C \sqcup D$	`ObjectIntersectionOf(C, D)`
Existential Quantification	$\exists R.C$	`ObjectSomeValuesFrom(R, C)`
Universal Quantification	$\forall R.C$	`ObjectAllValuesFrom(R, C)`

Table 2.2 Description logic syntax and OWL API equivalencies.

[3] λ_{DL} itself does not make use of top and bottom roles, so they are only listed here for completeness. Section 4.4.4 explains what these roles are to be used for.

[4] OWL API also has `DataTypeProperty`, but the knowledge base implementation only uses `ObjectProperty`.

2.3 λ_{DL} Language Features

The λ_{DL} language is at its core a simply-typed λ-calculus [Loa98]. It includes features such as let-bindings, function definition and application, if-else expressions and operations `cons`, `head`, `tail` and `null` to operate on lists. For recursive functions, a fixed point operator is included.

The semantic data features are based on the DL described in section 2.1. Concept and role literals are represented by a kind of abbreviated Internationalized Resource Identifiers (IRIs). Parentheses may be used for grouping DL expressions.

```
λ(wine:Wine ⊓ ∃hasMaker.Winery) . doSomethingWith wine
```

Listing 2.1 Concept type definition example in λ_{DL}.

Semantic concepts can be used as types in λ_{DL}, and are specified with the DL syntax described previously. For instance, the concept atom `Wine` describes a type, as does the structural concept expression `Wine` ⊓ ∃ `hasMaker.Winery`. Subtyping is resolved by the knowledge base: a concept is a subclass of another if that fact is known to be true. For example, `RedWine` is a subtype of `Wine`, as is defined by the ontology. Importantly, the type hierarchy is *not* calculated ahead of time, subtype relationships are checked by the knowledge base at the point that compilation requires it. Listing 2.1 shows an example of a type specification on a function.

```
query RedWine ⊓ (DryWine ⊔ OffDryWine)
```

Listing 2.2 Querying example in λ_{DL}.

Another application of the DL is to retrieve individuals from the knowledge base using the `query` operator. Given a concept expression, it queries the knowledge base and returns an appropriately-typed list of all known individuals that are members of the concept. If a query is deemed unsatisfiable by the knowledge base, it is rejected at compile-time. Listing 2.2 shows an example query that searches for all red wines that are also dry or off-dry wines. The query is satisfiable and therefore accepted at compile time. It returns a list with individuals of the same type as the DL expression it was given, i.e. `RedWine` ⊓ (`DryWine` ⊔ `OffDryWine`).

```
let getWineColor = λ(wine:Wine ⊓ ∃hasColor.Color) . head wine.hasColor
```

Listing 2.3 Projection example in λ_{DL}.

To retrieve properties of an individual, the projection operator `.` followed by a DL expression describing a role is applied to the individual. It queries the knowledge base for the given projection and returns a list of matching individuals. Listing 2.3 shows an example of retrieving the color of a wine by projecting the `hasColor` role onto it. As a wine is expected to only have one color, the `head` operator is applied to the result to retrieve it.

```
case head query Wine of
  type ∃hasBody.{Full}   as full   -> "full, strong body"
  type ∃hasBody.{Medium} as medium -> "medium, balanced body"
  type ∃hasBody.{Light}  as light  -> "light, soft body"
  default "you found an incorporeal wine"
```

Listing 2.4 Type casing example in λ_{DL}.

In cases where the ontology cannot prove a type relationship, a run-time type disambiguation may be necessary that checks the type membership of a specific individual. For this, λ_{DL} provides a type casing statement, of which an example can be seen in listing 2.4. The individual given in the `case` clause is checked against each of the `type` cases in order. As soon as one of the cases matches, the value is bound to the name after the `as` keyword and the associated expression is executed. If none of the cases match, the `default` expression is executed instead.

2.4 Sample Program

```
let getWines = λ(producer:Winery) . producer.hasMaker⁻ in
  let producedWines = (getWines head query {ChateauChevalBlanc}) in
    if null producedWines
      then
        "no wine is known for this winery"
      else
        case head producedWines of
          type RedWine   as red   -> "red wine for meat"
          type WhiteWine as white -> "white wine for fish"
          type RoseWine  as rose  -> "rose wine for rice"
          default "don't drink colorless wine at all!"
```

Listing 2.5 λ_{DL} example program.

This section walks through the program shown in listing 2.5. The program's intent is to recommend a pairing of food for a wine from a specific winery.

The program begins by defining a function `getWines` that retrieves the wines of a `Winery` by performing a projection of `hasMaker`$^{-}$ onto it. This will return a list of what the given winery produces according to the Wine Ontology.

It then queries for the nominal `ChateauChevalBlanc` and retrieves the `head` of the result, which is expected to be the only result of the query. It then calls the previously defined `getWines` function with this individual as a parameter. This passes the type checking phase as the knowledge base can determine that the nominal `ChateauChevalBlanc` is a `Winery`. The retrieved list is bound to `producedWines`.

It is then checked if the list is empty by use of the `null` operator. If this is the case, the program returns a string describing the issue: the given winery does not produce any wines. While this would be a strange occurrence in the real world, an open-world ontology can reasonably have such deficiencies.

If the list is not empty, a type case is performed on the first individual of the list. If this individual is a red, white or rosé wine, an appropriate food recommendation is given for it. If the wine is of an unknown color, which is once again probably just a shortcoming of the data, an appropriate warning is given to the user.

While this program is sound, the limitations of λ_{DL} become apparent when one wishes to extend it. In particular, to receive a recommendation for a wine from a different winery than Chateau Cheval Blanc, one would have to modify the program's source code and replace the winery's IRI with something else. Given a real-world application where such recommendations need to be given according to user input, λ_{DL} quickly becomes infeasible.

This is the point where the work in this thesis takes off from. To make the ideas brought forth by λ_{DL} viable in practical applications, it attempts to integrate them into an existing general-purpose language. This will combine the features of λ_{DL}– type-safe, integrated handling of semantic data – with the vast array of libraries and functionality of a real-world programming language.

Chapter 3

Related Work

This work is associated with programming with semantic data and integrating ontologies as foreign data models into programming languages as types. This chapter shows work that relates to these topics in four different ways: the classical way of programming with semantic data using *generic representations* for virtually typeless access, *mapping* ontology data models to the programming language's types, integrating *query* facilities into the language and, finally, extending the language's *type system*.

As this thesis builds upon research and development done for λ_{DL}, further related work can be found in the preliminary language report [LLS16]. While the implementation in this work deals with language extension, extensible compilers and language workbenches, it is merely a user of these tools and therefore does not include them as related work. The relevant discussion about these platforms can be found in chapter 5.

3.1 Generic Representation

The classical way to program with semantic data does not integrate the ontology model into the type system at all. Instead, generic types like `Individual` are used. This is analogous to the generic `Node` types of XML's DOM [WLHA+98]. As described in chapter 1, this almost typeless approach does not fit well into statically typed languages. The advantages of compile-time type-checking are lost when dealing with the semantic data.

SPARQL[1] [PS08] is a common way to access semantic data this way. Libraries such as RDF4J[2] (formerly Sesame), ARQ for Apache Jena[3], RDF::Query[4] and many others provide a programmatic interface to querying a triple store using

[1]Standing for the recursive acronym "SPARQL Protocol and RDF Query Language"
[2]`http://rdf4j.org/`, accessed 2017-08-18.
[3]`https://jena.apache.org/documentation/query/index.html`, accessed 2017-08-18.
[4]`https://metacpan.org/pod/RDF::Query`, accessed 2017-08-18.

SPARQL queries. The queries are generally represented as strings in the program, which brings with it the same injection issues that similar SQL interfaces suffer from [OAA$^+$10]. Individuals retrieved from query result sets use some generic `Individual` type or similar.

A method of dealing with semantic data without using a separate query language can be found in the OWL API [HB11], which the λ_{DL} protoype uses for its knowledge base implementation. The OWL API library provides access to ontology data in various formats, such as OWL and RDF, and enables querying that data via pluggable reasoners such as HermiT [SMH08]. Queries are built by instantiating a description-logic-like object model, which is then handed to the reasoner to perform the actual operation. While this method precludes injection issues, compared to a domain-specific language intended for semantic queries, manually building the description logic object model is comparatively clunky. The generic type for individuals is `OWLIndividual`.

3.2 Mapping Approaches

A common approach to integrating foreign data with the type system of statically typed languages is by mapping the hierarchy of the data into regular types of the language, such as classes. However, this approach has several downsides. For one, it requires generating all type information ahead of time, which in the case of large ontologies can be a prohibitively large amount.

This is especially the case with λ_{DL}'s types, as it allows constructing new concepts via description logic expressions on a whim, leading to a nigh-infinite amount of possible types. It can also lead to confusing type errors, as the mapping of a generated type to the original data is not necessarily obvious from the type name, and may require mapping the name back to the original description. The information in the original data model is duplicated by the mapping, rather than using it directly.

Jastor[5] [KPBP04], agogo [PSW$^+$09] and RDFReactor [VS05] generate a class model of the ontology data ahead of time via code generation of a class model for the ontology data. LITEQ [LSL$^+$14] uses a similar approach, but instead of directly generating code, it uses type providers of the F# language [SBT$^+$13] to accomplish creation of the types instead. ActiveRDF [ODG$^+$07] generates the necessary classes dynamically at run-time, but at the expense of deferring type checking to run-time as well.

[5]`http://jastor.sourceforge.net/`, accessed 2017-07-18

3.3 Query Integration

There have been efforts to integrate semantic queries into the C# language via LINQ [MBB06]. While this language integration provides some assistance in regards to well-formedness of queries and can aid in proper parametrization to prevent possible injection, the results received from these queries once again fall back to using generic representations. However, while they lack result types, using SPARQL gives them a more fully-featured query language than λ_{DL}'s description-logic-based queries and projections provide.

LINQ to RDF[6] is a prototypical example for this technique, giving an overview of integrating semantic queries via LINQ. LINQtoSPARQL [7] is a library implementation for performing SPARQL queries through LINQ. There also exists a JSON-view-based approach [KAK15], which provides an abstraction over constructing raw SPARQL queries.

3.4 Type System Extension

Semantic concepts as types are one of the most significant aspects of λ_{DL}. The closest analog to this is Zhi# [PV11], which provides integration of OWL and XSD data models as types into the C# programming language. It is implemented via a custom compiler framework [Paa09]. Like λ_{DL}, Zhi# does not perform any mapping of the new types to a class hierarchy or similar, but instead performs type-checking directly by using a semantic reasoner. However, it does not allow for structural types like λ_{DL} does, so a type like $\exists \texttt{hasMaker}^{-}.\top$ is not directly constructible using Zhi#'s types. It also lacks integrated query facilities, instead deferring this to existing APIs to be used in conjunction with the language.

While not directly related to semantic data, the XJ Java language extension provides an integration of XML Schema types and XPath queries [HRS+05]. Rather than attempting to map schemas to a Java class hierarchy, it uses an XPath engine to type-check the XML Schema types, similar to how λ_{DL} uses a knowledge base to do its type checking. XJ is implemented using the Polyglot extensible Java compiler [NCM03].

[6]https://virtuoso.openlinksw.com/whitepapers/rdf%20linked%20data%20dotNET%20LINQ.html, accessed 2017-08-18

[7]https://github.com/Efimster/LINQtoSPARQL, accessed 2017-08-18

Chapter 4

Language Extension Design and Requirements

The following chapter details the specifications of the λ_{DL} language extension. It explains the goals, shows some exploratory implementation work and then dissects the requirements for the features of the language extension, the language it is to be based on and the extension mechanism to be used. Finally, the findings are summarized in a requirement catalog.

4.1 Goal

The intent of this thesis is to design and implement an *extension* to an existing general-purpose programming language, which aims to *integrate* the central concepts of λ_{DL} into that language. The goal is *not* to integrate λ_{DL} itself as a domain-specific language into the existing language.

Concretely, the key features and principles of λ_{DL} [LLS16] should be integrated into the base language, and should act like built-in features of that language:

- Using semantic concepts as native types, specified by a description logic syntax that allows for structural type specifications.
- Inferring the subtype relationship between these types when the type check occurs during compile-time, rather than computing them ahead of time.
- Allowing the user to distinguish semantic concept types at run-time as well, in cases where the ontology provides insufficient information and a decision must be based on specific individuals.
- Querying the knowledge base and performing projections on individuals retrieved via these queries, also using the description logic syntax used for semantic concept types.

4.2 Prototyping

```
use Semantics "wine.rdf";

sub getWines($producer where (* ⊑ <:Winery>)) {
  $producer → <:hasMaker>⁻
}

my @producedWines = getWines((query <:ChateauChevalBlanc>).first);

say do if !@producedWines {
  "no wine is known for this winery"
}
else {
  given @producedWines.first {
    when * ⊑ <:RedWine>   { "red wine for meat"   }
    when * ⊑ <:WhiteWine> { "white wine for fish" }
    when * ⊑ <:RoseWine>  { "rose wine for rice"  }
    default { "don't drink colorless wine at all!"}
  }
}
```

Listing 4.1 Equivalent Perl 6 example for listing 2.5.

For exploring how a language extension would look like and which kinds of applications there would be for it, a prototype implementation was created, using the *Perl 6* programming language[1]. As it supports gradual typing, custom operators and run-time type constraints, building a compiler extension was not necessary to create the prototype.

While it has an imperfect syntax and defers most type-checking to run-time, the prototype is still very useful to create sample applications and explore functionality beyond what λ_{DL} itself is capable of. For instance, λ_{DL} has no facilities for external input, which is a necessity to create any kind of interactive application.

Differences in the description logic syntax compared to λ_{DL} are as follows:

- Semantic concept type constraints are checked via a custom ⊑ subtype operator, as description logic expressions on their own do not trigger a type check.

[1]This implementation is available at `https://github.com/hartenfels/Semantics`, accessed 2017-08-18.

- Concept atoms, role atoms and nominal concepts are strings referencing a proper, optionally abbreviated IRI. To make them visually distinct from regular strings, the code examples delimit them with angle brackets (`<>`)[2].
- The dot operator is already used for method calls in Perl 6 and cannot easily be overloaded. Therefore, instead of a dot, quantifiers use the pair constructor operator `=>` and projections use a new `→` projection operator.

The semantics however, remain identical to the way these operations work in λ_{DL}. Listing 4.1 shows the sample program from section 2.4 translated to the Perl 6 prototype implementation. It tries to stick very close to the λ_{DL} example to make the equivalences clear, and as such is not an example of idiomatic Perl 6 code.

4.3 Knowledge Base Service

```
Request:  ["wine.rdf","query",["C",":Wine"]]
Response: [
  "http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#BancroftChardonnay",
  "http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#RoseDAnjou",
  "http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#TaylorPort",
  // more results follow…
]
```

Listing 4.2 Request and abbreviated response for a `query <:Wine>` operation.

To perform type checking, queries, projections et cetera, a semantic reasoner is required. However, reasoner implementations are based on various different languages [MK11], which are not necessarily directly accessible from whatever language the extension is eventually created in.

To mitigate this, a knowledge base *server* application has been built[3], which provides the necessary reasoner functionality via a TCP connection over network sockets, much like a relational database management system would. As sockets are standard I/O functionality available by default in virtually any programming language, this should provide an interface accessible by any kind of platform. As the server is able to run persistently, it can also provide in-memory and on-disk caching for repeated queries to improve performance.

[2]Angle brackets are actually the quote-words operator, rather than regular string delimiters. However, as none of the IRIs used contain whitespace, the operator effectively produces a single string.

[3]The implementation of the knowledge base server is available at `https://github.com/hartenfels/Semserv`, accessed 2017-08-18.

The implementation is roughly based on the prototypical implementation of λ_{DL}[4] It is written in the Scala programming language [OAC+04a] and uses the HermiT reasoner [SMH08] via the OWL API interface [HB11].

The server uses a simple line-delimited JSON [Bra14] protocol: the client sends a line of JSON requesting a certain action to be performed, such as a query or a subtype check. Each request also contains the data source the operation is to be performed on (see section 4.4.1). The server then executes the requested operation and returns the results encoded in a line of JSON. See listing 4.2 for an example.

A JSON schema document [GZ+13] specifies the structure of the request operations. The structure of the responses is either a simple boolean value in the case of subtype or membership checks or an array of IRIs (represented as strings) in case of queries or projections. In case of error, an object is returned containing the exception details.

4.4 Features

The following sections discuss the individual features that the language extension will have to implement: specification of an ontology as data source, semantic concepts as types, semantic concept type casing, queries and projections. Where applicable, it shows an example of the feature using the Perl 6 prototype from section 4.2.

4.4.1 Data Sources

```
use Semantics "wine.rdf";
```

Listing 4.3 Prototype data source specification.

Operations on semantic data require an ontology to operate on to make any sense. A query such as `<:RedWine> ⊓ <:WhiteWhine>` does not have any meaning on its own, it is also necessary to specify that this query is supposed to be performed on the semantic data of the ontology described in `"wine.rdf"`. Similarly, concept types also require a knowledge base to reason about them when performing type-checking.

Listing 4.3 shows how the Perl 6 prototype accomplishes this: the semantic data module receives a data source in the form of a path to an ontology file as an argument upon importing it. All operators and subroutines imported into the current scope will close over this argument and pass it to the knowledge base as

[4]Available under “Prototypical implementation” at https://west.uni-koblenz.de/lambda-dl, accessed 2017-08-18.

the context for all operations. The scope of the specification is lexical, inner scopes can override it by importing the module again with a different argument.

Ideally, a full implementation would handle this similarly by attaching a data source specification to a lexical scope, such as file scope or class scope. This way, both a compiler and a human reading the code would be able to discern the ontology context simply by the enclosing scope.

Alternatively, a data source could be attached to each semantic concept type declaration, query statement and projection operator. However, specifying it separately on every single operation would be too fine-grained and add much unnecessary clutter.

Another alternative would be to specify the data source as an option to the compiler. The disadvantage of this approach is that the contextual information would be outside of the code that uses it, hampering readability and adding complexity to the compilation invocation. It may also impede projects that want to use multiple different ontologies, as restricting a compiler option to a subset of source files could be a tricky task for the user, depending on the build system used.

4.4.2 Semantic Concepts as Types

Semantic concept types are the defining feature of λ_{DL}. They are constructed from description logic expressions describing conceptualizations of semantic data. These types are to be integrated into the base language's type system so that they can be used, for instance, in function parameter declarations. Where appropriate, they should also integrate in whatever other aspects of the language that types are involved in, such as pattern matching, type casts, generic type arguments or run-time type assertions.

A particular challenge for the integration of these types into a language is that subtype inference and type identity must be performed by a knowledge base. Attempting to map these types to a type hierarchy or a set of implicit type conversions is infeasible due to the exorbitantly high amount of possible concepts that can be constructed using a description logic expression [LLS16]. This means a language extension needs to go deeper than simply extending the surface syntax of the language, and instead must also augment the compiler's type checking rules.

A possible alternative to extending the typing rules themselves would be to analyze all conceptual types that are actually *used* within a program and only generate a type mapping for those types. While such a mapping is possible [KPBP04], it has the drawback of requiring *whole-program analysis*: all semantic concept types that are used within a program need to be known before compilation to generate the necessary type mapping. This leads to limited modularity, as two different program modules will generate a different set of types, which will be incompatible with each other even if they refer to the same ontology. The resulting type errors would potentially cause usability issues, since error messages would refer to the generated types, rather than the actual description logic expression the programmer used. And finally, this approach goes against the philosophy of λ_{DL}, which explicitly rejects mapping in favor of using the knowledge base directly [LLS16]. As such, the method is heavily disfavored.

Another unusual property of these types is that they do not possess a source definition: they are simply declared as a description logic expression and spring into existence as needed. This is opposed to the usual case, where types are defined as structures, classes et cetera, which are then later referred to. This may require further adjustment to the type system that expects definitions to exist before the types they define are accessed.

To catch typos and otherwise incorrect types, all concept atoms, role atoms and nominal concepts in types should be validated against the ontology's signature [BCM99]. Any concepts, roles or nominals that are not part of the ontology are likely to be invalid, and therefore should be diagnosed at compile-time. As there may be valid cases of using elements that do not exist in the ontology, such as the ontology being a work in progress, the diagnosis should either be a warning or a suppressible error.

4.4.3 Type Casing

```
multi sub getProduction($producer where (* ⊑ ∃<:hasMaker>⁻ => ⊤)) {
  return join ", ", $producer → <:hasMaker>⁻;
}

multi sub getProduction($winery where (* ⊑ <:Winery>)) {
  return "{$winery.name} is a winery without wines!";
}

multi sub getProduction(Any $) {
  return "There's only sour grapes here.";
}
```

Listing 4.4 Multiple dispatch via run-time pattern matching on concept types.

Type casing is λ_{DL}'s facility for run-time type dispatch. Its semantics are simple and very similar to a `switch` construct or a chain of `if`/`else` statements. In fact, the prototype does not provide any special construct for it at all. Instead, Perl 6's switch-like `given`/`when` statement (see listing 4.1) or multiple dispatch functions based on run-time pattern matching (see listing 4.4) can be used.

However, even in a language that provides built-in features that could replace type casing, having a dedicated construct for semantic concept types would add more chances for static checks that may not be possible with generic type dispatch constructs.

The cases can be checked for proper ordering at compile-time. If earlier types subsume later types, the later cases will never be executed and should therefore be reported as invalid.

Cases with useless type constraints can be detected as well. For example, a case with type ⊤ would match every time, and an unsatisfiable type like ⊥ would never match any individual. Those cases can then trigger appropriate errors.

Finally, the existence of a `default` case can be checked, ensuring that all possible types are handled and control cannot accidentally pass through the type casing block.

This feature should not be a challenge for any language extension framework, as it is simply adding to the language, not integrating into an existing facility. It requires the addition of some new syntax, static checks and code generation of a regular `if-else` chain or similar for execution.

4.4.4 Queries

```
sub searchWines(Str %input) {
  my $dl = <:Wine>;
  # possibly amend expression with strings from user input
  $dl ⊓= {%input<body> } if defined %input<body>;
  $dl ⊓= {%input<color>} if defined %input<color>;
  return query $dl;
}
```

Listing 4.5 Dynamic queries in the prototype implementation.

To actually retrieve any individuals to interact with, a means to query the knowledge base must be provided. The prototype handles this like λ_{DL} does: a `query` operator is provided that takes a concept expression as its argument and returns a list of matching individuals.

There is a major difference to λ_{DL} however: query expressions can be dynamic, rather than being fixed at compile-time. This is necessary to allow for queries based on user input, which is a much more useful and interesting feature in real applications than static queries. However, this poses several problems not addressed by λ_{DL} itself.

There needs to be some way to construct description logic expressions at runtime. A simple example would be a search function that looks up a concept and returns the matching individuals. The prototype implementation therefore allows using strings to construct concept atoms, role atoms and nominal concept expressions. These strings can come from literals, variables or anywhere else. It also allows using description logic expressions as values, which lets one construct a dynamic query expression incrementally by, for instance, storing it into variables. See listing 4.5 for an example of this.

These kinds of dynamic description logic expressions should of course not be available for semantic concept types, as those *must* be known at compile-time to check them in the first place. While the prototype implementation does not pay attention to this as it performs all type-checking at run-time, the final implementation needs to enforce that its types are constant. This may either be accomplished by ensuring that the strings used in types are literals, or by using a different form of literal altogether.

Expression	**Inferred Type**
$?$	$\top$ or $\triangledown$
$\neg C_?$	$\top$
$R_?^-$	$\triangledown$
$\exists\, ?.\, C$	$\exists\, \triangledown . C$
$\forall\, ?.\, C$	$\forall\, \vartriangle . C$
$\exists\, R\, .\, ?$	$\exists\, R\, .\top$
$\forall\, R\, .\, ?$	$\forall\, R\, .\top$
$C \sqcup ?$	$C \sqcup \top$
$C \sqcap ?$	$C \sqcap \top$

C and R stand for a concept and a role expression respectively. $C_?$ and $R_?$ stand for a concept or role expression that contain any unknown elements anywhere in them. ? stands for an unknown element not known at compile-time, such as a reference to a variable.

Table 4.1 Query type inference.

As dynamic query arguments are not known at compile-time, the type of values that the query will return cannot be completely inferred. However, depending on which parts of the expression are unknown, an upper bound can be calculated regardless, see table 4.1 for an overview of the rules for it. If no better type can be inferred, the upper bound of a query is $\top$.

To improve typing of dynamic queries where quantifiers with unknown roles are involved, the top role $\triangledown$ and bottom role $\vartriangle$ defined in section 2.1 are used to provide a more precise upper bound. Without them, the entire expression would be bounded only by $\top$.

For example, assuming that `x` is some variable whose value is not known at compile-time, a query for `<:Wine> ⊓ {x}` would have an upper bound of `<:Wine> ⊓ T`. A query for `¬x` would have an upper bound of $\top$, as the negation of a concept with unknown constituents yields an unknown value.

Queries should return lists or sets of the appropriate inferred type. This type should also be used to check if the query is unsatisfiable at compile-time, yielding an error if that is the case, as happens in λ_{DL}. This check should not happen with dynamic queries at run-time, as they may be unsatisfiable for valid reasons.

For example, a search query initiated through user input may be unsatisfiable simply because the user searched for something that does not exist, which is not an exceptional or erroneous circumstance.

As with description logic expressions for types, concept atoms, role atoms and nominal concepts that are not part of the ontology's signature [BCM99] are likely to be errors when queried for. However, as queries may be dynamic, these potential errors should be diagnosed at compile-time where possible and at run-time when the query is actually executed. The diagnosis must not be a fatal error, as dynamic query arguments can come from anywhere, and should not require additional validation to prevent the program from crashing.

4.4.5 Projection

Projection is a similar case to querying from the previous section: a data source must be in scope to type the projection properly and there must be some sort of projection operator that receives a role expression as its argument.

The type of a projection of the form $i.R$, where i is some individual of concept type C and R is some role type, is a collection of $\exists R^{-}.C$ as its type. As with queries, if the projection's role argument contains elements not known at compile-time, they are resolved to an upper bound as described in table 4.1. In particular, a role that is entirely dynamic resolves to a collection of $\exists \triangledown.C$, again making use of the top role defined in section 2.1.

In λ_{DL}, satisfiability of projections is not checked like it is for queries [LLS16]. According to the authors, this is an unintentional shortcoming. Therefore, it would make sense to at least apply the same satisfiability requirements as for queries, which means that the resulting type of $\exists R^{-}.C$ must not be unsatisfiable.

However, this is a weak condition, which is unlikely to be possible to discern as unsatisfiable unless the ontology defines this relatively specific case. A more useful condition would be to ensure that C is a subtype of $\exists R.\top$. This condition is of course also more restrictive and may exclude valid projections, so there would need to be a way to get around this somehow and execute a projection regardless. For example, by using a type case to check if the individual has the correct type or replacing the projection with an equivalent query using the individual as a nominal concept.

Projections should also return lists or sets as collections for their resulting individuals. Signature validation should behave as it does for queries as well: a non-fatal diagnosis at compile-time where possible using the inferred upper-bounded type, as well as at run-time when the projection is executed with its eventual values.

4.5 Extension Considerations

This section deals with the requirements of the language extension internals: the language it is based on and the mechanism by which that language is extended. Special care is taken to discuss backward compatibility issues, to ensure that the language extension does not break the semantics of existing and working code.

4.5.1 Base Language

There are some considerations to make in regards to the language to extend. The concepts and ideas from λ_{DL} should mesh properly with the base language, as well as with the general ecosystem of semantic data programming, to eventually provide a usable and cohesive unit.

As integrating semantic concepts into the type system is the primary thesis of λ_{DL} (see section 4.4.2), the base language should have an appropriate type system to integrate into. A strong static type system, which is what λ_{DL} itself uses, would be preferred for this, since the language would already have the necessary type checking in place that could be amended by the new semantic concept types.

Other methods of checking types at compile-time may be an option as well, such as typed extensions of dynamically typed languages. Examples include Typed Clojure [BSDTH16], Typed Lua [MMI14] or TypeScript [BAT14]. However, as these type systems are still built upon dynamically typed languages, the type integration would not be as complete as with a static type system.

As this type-checking should occur at compile-time, a language that has a separate compilation phase is required. An interpreted language that provides run-time type checks only is insufficient.

Much of the ecosystem of semantic data software is centered on the Java language and it's Java Virtual Machine (JVM) platform [SET09]. To allow for the best interoperability with these existing programs and libraries, using Java or some other JVM-based language would be the most fitting approach. Languages that can interface with Java in some other manner are to be considered as well.

Since λ_{DL} is based on the functional λ calculus [LLS16], the base language for the extension should ideally also provide a degree of functional programming support. This way, already existing example programs for λ_{DL} could be more easily and directly ported to the new environment.

4.5.2 Extension Platform

To allow for a clean implementation with possible future developments, the extension should be built upon a platform that is intended for such a task. It should allow for a structured and documented approach to adding the necessary features to the base language.

Ad-hoc solutions, such as modifying compiler internals or constructing an impromptu source-to-source compiler, are disfavored, and only considered as a last resort if no appropriate platform can be found.

Due to the many different aspects of the language that are to be affected by the extension, a sufficiently powerful mechanism for creating this extension is necessary. Based on the discussion in section 4.4 above, the following aspects of the language will need to be augmented, and the extension platform therefore needs to be able to support these augmentations:

- Attaching contextual information to a scope at compile-time, to allow for specification of a data source (section 4.4.1).
- Description logic expression syntax, in particular support for adding prefix, infix, postfix and circumfix operators, and semantics. See sections 4.4.4 and 4.4.5.
- Description logic type syntax. This is a separate concern to the above point, as a language may allow definition of custom operators, but may not allow these operators to be used to construct structural types at compile-time. See section 4.4.2.
- New syntactical constructs with custom static checking, to implement type casing (section 4.4.3) and querying (section 4.4.4).
- Integrating a new set of types into the existing type system and specifying a custom way of specifying their subtype relationship without mapping them to some explicit hierarchy, to implement semantic concepts as types checked by the knowledge base (section 4.4.2).

4.5.3 Backward Compatibility

As the language being extended will have an ecosystem of existing code, some thought will need to be given in regards to compatibility with that code. For instance, introducing a new keywords to a language like Java would break any existing code that uses this word as an identifier. Especially common words such as `query` or `type` that are present in λ_{DL} would cause conflicts, to the point of requiring a re-write of the affected code – a major usability issue.

To avoid such issues, any code that is valid in the base language should retain its semantics even when the extension is in use. Code that is syntactically or otherwise statically invalid in the base language may be given different semantics in the extension. This approach should not cause any breakage, as existing code is unlikely to use constructs deemed invalid at compile-time, lest it would not compile and function in the first place.

A possible strategy for this is to simply use symbols that are unused in the base language. For example, Java does not use the symbol `⊔` in its syntax, therefore it could freely be added as a new infix operator without causing any conflicts.

Similarly, existing keywords of the language can be re-used in contexts where they are not valid in the base language. For example, the `case` keyword in Java is only valid inside of a `switch` statement block. To implement syntax for a type casing feature (see section 4.4.3), the `case` keyword could be used at the beginning of a statement to introduce the type case. There are no conflicts with existing code, as the keyword is never valid at the beginning of a statement in the base language.

A different strategy would be to use compiler pragmas that are restricted to some scope. The compiler would only enable the extension functionality where the pragma is in scope, and otherwise behave normally. Since only new code intended for use with the extension would enabling the behavior, existing code would continue to function as before. However, this method makes more sense if pragmas are already part of the language, as otherwise there also needs to be additional syntax using the above method to allow for specifying pragmas in the first place.

4.6 Requirements Catalog

The following catalog summarizes the results from the discussion in this chapter into a set of requirements. They lay out the prerequisites for the *language being extended*, into which the features are integrated, the *extension mechanism*, which is used to implement these integrations, and the *extension* itself, which concerns the functionality of the final product.

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this section are to be interpreted as described in RFC 2119 [Bra97]. Relevant sections that discuss the reason behind a requirement are given in parentheses.

- The language being extended...
  - ...MUST have a separate compile-time phase, during which static checking can be performed. (Section 4.5.1.)
  - ...MUST have the ability to check types at compile-time, and ideally SHOULD have a static type system for this. (Section 4.5.1.)
  - ...SHOULD provide support for functional programming. (Section 4.5.1.)
  - ...SHOULD be compatible with the Java Virtual Machine (JVM). (Section 4.5.1.)
- The extension mechanism MUST provide facilities to implement syntax for...

  - ...description logic expressions. Their syntax SHOULD be analogous to the one used in λ_{DL}. (Section 4.4.4.)
  - ...semantic concept type declarations. Their syntax SHOULD be analogous to the one used in λ_{DL}. (Section 4.4.2.)
  - ...a query operator. This MAY be the keyword `query`, as in λ_{DL}. (Section 4.4.4.)
  - ...type casing with semantic concept types. This MAY use the keywords `case`, `type` and `default`, as in λ_{DL}. (Section 4.4.3.)
  - ...projection with a semantic role. This MAY use a dot operator followed by a description logc expression, as in λ_{DL}. (Section 4.4.5.)

- The extension mechanism...
  - ...SHOULD be a well-defined interface intended for extending the language. (Section 4.5.2.)
  - ...MUST enable semantic concept type inference of queries and checking of their satisfiability at compile-time. (Section 4.4.4.)
  - ...MUST enable compile-time checking of static projections. (Section 4.4.5.)
  - ...MUST enable compile-time checking of type casing, to ensure that the cases are satisfiable and ordered properly. (Section 4.4.5.)
  - ...MUST enable code generation for queries, projections and type casing. This SHOULD NOT require source-to-source translation, and instead SHOULD be embedded in the compiler's regular code generation pass. (Sections 4.4.4, 4.4.5, 4.4.3 and 4.5.2.)

- Either the language being extended or the extension mechanism SHOULD provide facilities for attaching a data source specification to a lexical scope. This scope MAY be one of file scope, class scope, module scope or package scope. (Section 4.4.1.)

- The extension mechanism MUST allow for compile-time checking of semantic data types. This checking SHALL NOT be in the form of a mapping approach, but instead SHALL use the knowledge base to perform these checks. (Section 4.4.2, see also [LLS16].)

- The extension...
  - ...MUST provide run-time communication with the knowledge base for execution of queries, projections and type casing. (Sections 4.3, 4.4.4, 4.4.5 and 4.4.3.)

- ...SHOULD diagnose concept and role atoms used by description logic describing type, queries and projections that are not part of the ontology's signature. If the concept and role atoms are known at compile-time, the diagnosis SHOULD occur at compile-time. Otherwise, the diagnosis SHALL occur at run-time. This SHALL NOT be a fatal error, but instead SHALL be a warning or a suppressible error. (Section 4.4.2, 4.4.4 and 4.4.5.)
- ...SHOULD retain compatibility with the language being extended. Specifically, the semantics of *valid* code in that language SHOULD NOT change between the original and the extended language. Code that is *invalid* at compile-time in the base language MAY be repurposed freely. (Section 4.5.3.)
- ...SHOULD allow semantic concept types to be used in the same way that other types are used in the language. (Section 4.4.2.)
- ...SHOULD represent results from queries and projections as a form of list or set. (Sections 4.4.4 and 4.4.5.)
- ...MUST trigger a compile-time error when an unsatisfiable query or projection is detected. They SHALL NOT be validated at run-time. (Sections 4.4.4 and 4.4.5.)

Chapter 5

Platform Research

In this chapter, possible languages and extension mechanisms to build the language extension on are presented. An overview of each is given and the most suitable technologies are discussed in detail and compared against each other. From these options, the most appropriate platform is chosen to base the implementation on.

5.1 Language and Technologies Overview

	Type System	**Primary Platforms**	**JVM Interop**	**Extension Mechanisms**
Perl 6	gradual	MoarVM, JVM	JNI, natively	slangs, custom operators, custom type checking
Scala	static	JVM	natively	compiler plugins, custom operators
Java	static	JVM	natively	Polyglot, ExtendJ, JaCo, SugarJ, JavaCOP
Clojure	dynamic	JVM, JavaScript	natively	Lisp macros
F#	static	.NET	JNI	type providers
Haskell	static	Native	JNI	Template Haskell

Table 5.1 Languages considered as an extension base.

Table 5.1 shows the languages considered as a base for being extended. As described in the previous chapter, the focus lies on JVM-based languages with support for functional programming, but other platforms were also considered. For languages from different platforms, JVM compatibility still exists via the Java

Native Interface (JNI), which would at least make it possible to communicate with existing software on that platform.

The table also lists the typing discipline of these languages. From these, only Clojure is excluded from being an appropriate extension base, as it heavily relies on dynamic typing and prefers specifying the structure of data to types[1], which does not fit in with λ_{DL}'s types. Perl 6 supports gradual types, a discussion of which follows in section 5.2. Static type systems should meld well with concepts as types, if they can be extended as such.

	Kind	**Syntax Extension**	**Type System Extension**
Perl 6	extensible language	yes	partial
Scala	extensible language	partial	partial
SugarJ (Java)	language extension framework	yes	no
JavaCOP	pluggable type system	no	yes
Polyglot (Java)	extensible compiler	yes	yes
ExtendJ (Java)	extensible compiler	yes	yes
JaCo (Java)	extensible compiler	yes	yes
F#	type providers	no	only mapping
Template Haskell	compile-time meta programming	yes	no

Table 5.2 Extension mechanisms considered.

The actual mechanisms for implementing the extension are summarized in table 5.2. The primary features such a mechanism is required to support (see section 4.6) are adding new *syntax*, for describing concept types, type cases, queries and projections, and extending the *type system* to implement type checking using a knowledge base.

As described in section 4.5.2, only platforms *intended* for language extension were considered. Performing ad-hoc preprocessing, modifying compiler internals or

[1] `https://clojure.org/about/spec`, accessed 2017-08-18.

creating a new, custom compiler – while virtually always possible – were therefore excluded.

In the case of Perl 6 and Scala, the languages already provide some level of extensibility without any additional technologies. For instance, both languages allow defining custom operators and extending the compiler via a plugin mechanism. They are discussed in detail in sections 5.2 and 5.3 respectively.

SugarJ for Java enables the necessary syntax extensions, but does not allow extending the type system enough. This may be mitigated by using it in conjunction with JavaCOP, which in turn only deals with type system extension. The discussion for this set of technologies follows in section 5.4.

There also exist several compilers for Java that are intended to provide a framework for extending the language cleanly: Polyglot (section 5.6), ExtendJ (section 5.5) and JaCo (section 5.7). As they all provide direct access to the compiler, syntax and type system can be extended directly.

The F# language's type providers [SBT+13] are intended for integrating external data as types into the program. However, in the end they result in types that are native to the language, and are therefore closer to a mapping approach than extending the type system itself. On top of that, there is no clean way to extend the F# language with description logic syntax to spell out these types, making it an unsuitable approach.

Template Haskell [SJ02] is intended for compile-time meta programming in the Haskell language. While it allows for extension of syntax and creating domain-specific languages, it does not provide any well-defined mechanism to extend the type checking phase. Because of this, the approach is not pursued further.

5.2 Perl 6

Given that it was already used to prototype the language extension, it is worth considering to use the existing Perl 6 implementation as a basis for a full Perl-6-based language extension.

The Rakudo Perl 6 compiler provides a JVM backend[2], which allows compiling Perl 6 code to JVM class files. Static type checking is supported as well, as long as the compiler can evaluate types at compile-time. Additional type specialization would need to be added to the compiler to support this for semantic concepts as types, as their evaluation is non-trivial.

The language also provides comprehensive functional programming facilities, amongst many other paradigms. It also supports additional features useful in conjunction with semantic concepts as types, such as multiple dispatch via pattern matching at run-time (see listing 4.4 in section 4.4.3).

Perl 6's syntax is malleable enough to allow for the creation of a description logic syntax without requiring any special extensions, it is simply a matter of

[2] https://perl6.org/compilers/, accessed 2017-08-18.

defining new operators. If any syntax modification would turn out to be necessary, so-called *slangs* are available that allow extending the parser's grammar itself[3].

However, Perl 6 is still a very new language with a limited user base. The JVM backend in particular is not production-ready yet and does not perform very well[4]. While static type checking is possible to a degree, the intent is not to provide comprehensive type-checking at compile time, leaving many places where types are only checked at run-time. In particular, all method calls are resolved at run-time by design, which makes static type-checking virtually impossible when objects are involved. This philosophy does not mesh well with the intentions of λ_{DL}.

5.3 Scala

On the surface, Scala already fulfills several of the requirements with built-in features: it runs on the JVM and interoperates well with Java, it has good support for functional programming and it supports the definition of custom operators [OAC+04a, OAC+04b]. The remaining functionality of static checking and code generation could be added in the form of plugins for the Scala compiler[5].

However, compiler plugins do not have a stable and documented interface – there hardly exists any documentation at all. There is also no public interface to extend a compiler phase, such as extending the type checking phase to support semantic concepts as types. Therefore, it would either require implementing the semantic concept type check as a separate phase, which would cause issues with features such as generic types that use semantic data types as one of their compounds, or to extend the type checking phase in the compiler itself, modifying the internals directly.

Structural description logic types pose a significant obstacle in this regard. The Scala language does not allow an expression where a type is expected, which rules out using custom operators for them, which only function in expressions. Instead, the parser would need to be extended to recognize description logic types as a new syntactic construct.

This in turn is not supported. It would either require modifying the internals of the existing parser phase, or performing a pre-processing step that transforms description logic types in the input into something that the regular Scala parser can understand.

In the end, this would effectively result in ad-hoc modifications to compiler internals, which are not intended to be augmented in that way and do not provide an appropriate interface. This is something the implementation should avoid (see section 4.6), making Scala an unfavorable choice.

[3]https://mouq.github.io/slangs.html, accessed 2017-08-18.

[4]http://rakudo.org/2017/07/24/announce-rakudo-star-release-2017-07/, accessed 2017-08-18.

[5]http://www.scala-lang.org/old/node/140, accessed 2017-08-18.

5.4 SugarJ and JavaCOP

SugarJ is a library-based language extension framework [ERKO11]. Aside from Java, Haskell is also supported in the form of SugarHaskell [ERRO12].

As the intention of the framework is to embed domain-specific extensions into the host language, syntax modifications are supported well. Implementing features such as operators for description logic expressions or a type case statement would be easily possible in SugarJ. As sugar libraries are only enabled where they are imported, the issue of backward compatibility would already be solved as well.

However, semantic concepts as types once again prove to be a major hurdle. While the syntactical aspects are achievable, performing proper type-checking is exceedingly difficult. As the name implies, SugarJ functions by *desugaring* the extended input into something the original language's compiler can understand, which is not possible with semantic concept types, as they cannot be directly represented in a hierarchical type system.

Instead, the type checking of semantic concept types would need to occur before desugaring, which would require re-implementing all type checking of the base language to make it function properly with, for instance, generic types in Java. This would of course be an excess amount of work, and not what SugarJ is intended for.

A possible mitigation strategy for this would be to integrate JavaCOP, a framework for pluggable types in Java [MME$^+$10], into the build pipeline. The SugarJ layer would desugar the syntax of the semantic concept type declarations into regular Java annotations, and not perform any semantic type checking on its own. The JavaCOP layer would then parse these annotations and perform the necessary type checks using the knowledge base.

While this is a possible solution, it is clunky at best. It would require separating the extension into two parts and constructing a metadata-based protocol to communicate information forward. As no prior work exists that combines SugarJ and JavaCOP, their interoperability is questionable as well. Therefore, to avoid the split in two and the additional effort of communicating between them, a more integrated solution would be preferred over this one, where the parsed representation can be operated on directly.

5.5 JastAdd/ExtendJ

JastAdd is a meta-compilation system [HM03], and built upon it there is an extensible Java compiler called ExtendJ (formerly JastAddJ) [EH07a]. It is based on reference attribute grammars [Hed00] and is extensible via aspect-oriented programming [KLM$^+$97]. Comprehensive documentation exists on JastAdd and ExtendJ, including tutorials and source code examples.

ExtendJ is a full Java compiler, with its own backend to produce JVM bytecode. It supports Java 8 [Hog14], which adds sufficient support for functional programming to the Java language in the form of lambda functions, method references and functional interfaces [GJS+15]. It is under active development as of the time of writing[6].

The compiler allows augmentation of syntax and semantics, including extensions to the type system. This interface has been used to add non-null types to Java [EH07b] and implement multitudes of objects [SÖH14], showcasing the inclusion of new categories of types, analyzing data flow at compile-time and adding new operators. These extensions are similar in nature to features such as semantic concept types and description logic expressions. The extensions' source code is publicly available[7], thereby providing an ideal sample for the areas that will need to be touched by the semantic data extension.

As it fulfills all necessary requirements, and is widely used and supported, the ExtendJ compiler appears to be an ideal platform to build upon.

5.6 Polyglot

Polyglot is an extensible Java compiler framework [NCM03]. It has similar capabilities to ExtendJ [EH07a], but uses traditional compiler phases and tree rewriting instead of a reference attribute grammar. It supports extensions to its parser and type system, and a similar non-null type extension has been developed for it [MJ]. The XJ language extension mentioned in section 3.4, which implements similar type system and querying extensions for XML processing, is implemented in Polyglot as well.

However, there is no extension to Polyglot to support Java 8. Therefore, an extension built on it would not support many of the functional programming features from that version of Java.

Additionally, Polyglot is a source-to-source compiler. While it performs all necessary static checks, including type checks, before translation, it still relies on an existing Java compiler for code generation. Custom bytecode generation is therefore not possible.

ExtendJ supports Java 8 [Hog14], as well as providing its own backend for code generation [EH07a]. As it has no obvious drawbacks otherwise, it is simply preferrable over Polyglot.

[6]As evidenced by regular commits in the ExtendJ repository under `https://bitbucket.org/extendj/extendj`, accessed 2017-08-18.

[7]Under `https://bitbucket.org/jastadd/jastaddj-nonnullinference` and `https://bitbucket.org/joqvist/multiplicities`, accessed 2017-08-18.

5.7 JaCo

JaCo is a Java compiler based on extensible algebraic data types [Zen04]. Like ExtendJ, it is a full compiler, including a backend for bytecode generation. It is intended for experiments with extensions to the Java language.

However, opposed to ExtendJ and Polyglot, there is relatively little information on JaCo. Existing extension implementations are primarily exploratory examples [ZO01]. Their source code is difficult to come by.

The compiler supports Java 1.4, which means generics and annotations from Java 5, and the functional programming features of Java 8 are unavailable. Version 2 of the compiler is written in Keris, a custom experimental programming language, which also only has few resources available for it[8]. The most recent release of JaCo and Keris has been in 2004.

Due to this lack of documentation and support, JaCo is not preferable over more prevalent options, such as ExtendJ (section 5.5) and Polyglot (section 5.6).

5.8 Results

Given the requirements gathered in section 4.6, the ExtendJ compiler (section 5.5) appears to be the most appropriate platform to build upon. It is powerful enough to support the necessary extensions to syntax, semantics and code generation, provides a well-documented interface and sports several existing extensions that can be used as a guideline for the implementation. The host language Java 8 supports functional programming to a sufficient degree, has an appropriate, static type system and is native to the JVM.

Therefore, the implementation of the language will be based on JastAdd, and constructed as an extension to the Java language via the ExtendJ compiler.

[8]`http://lampwww.epfl.ch/~zenger/keris/index.html`, accessed 2017-08-18.

Chapter 6

Java Language Extension Implementation

In this chapter, the various aspects of *Semantics4J*, the JastAdd-based Java semantic data language extension, are presented and discussed[1].

Each of the individual features is given an overview of their syntax and semantics from a user perspective. This is the only part that is necessary to understand how to use these features.

Implementation details follow under subsections where appropriate. The *Syntax* sections describe how the scanner and parser were extended to support the feature, and how possible incompatibilities with existing Java syntax were dealt with. The *Frontend* sections explain how the extension is integrated into the ExtendJ compiler's referenced abstract syntax tree, and how static checks are implemented. The *Backend* sections deal with the task of bytecode generation for the executable class files.

6.1 Class Library

To provide the necessary functionality at run-time, such as communication with the knowledge base or query construction, a library of classes is provided that must be available to programs using the extension at compile- and run-time. They are plain Java classes used to implement some of the extension's features. A class diagram summarizing the most relevant aspects can be seen in figure 6.1.

The `semantics.KnowBase` class provides an interface to the knowledge base server (see section 4.3). Each instance of this class carries with it its data source as its `path` member (see section 6.2) and abstracts away the low-level details dealing with socket connections. This class is used to, for example, perform subtype

[1]This implementation is available at https://github.com/hartenfels/Semantics4J, accessed 2018-08-18.

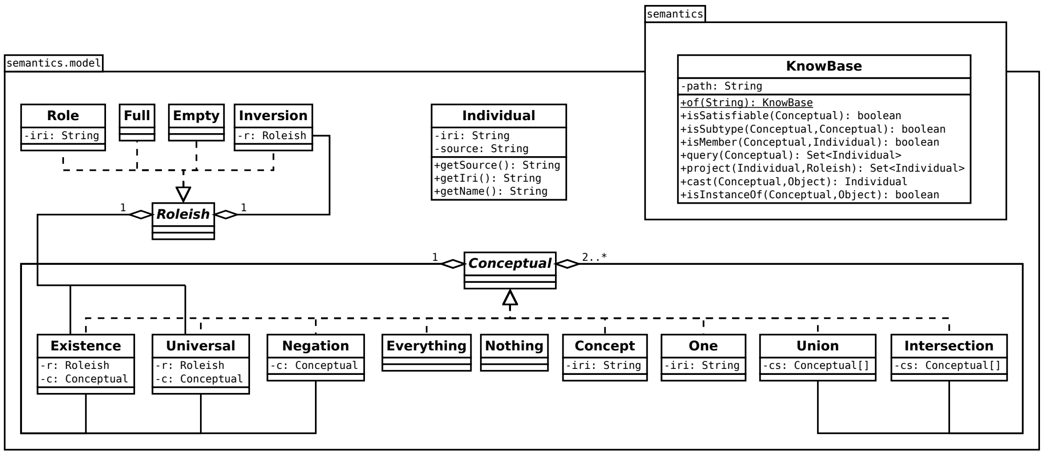


Figure 6.1 Simplified class diagram of the implementation model.

checking at compile-time (see section 6.3) and querying at run-time (see section 6.5).

To represent individuals returned from queries or projections against the knowledge base, the class `semantics.model.Individual` is used. Each instance is simply a wrapper around its IRI and a reference to the data source it originated from. Individuals implement the `Serializable` and `Comparable` interfaces. They can be hashed and compared for equality as well.

To represent description logic expressions at run-time, the interfaces `semantics.model.Conceptual` and `semantics.model.Roleish` are used. They form the supertype for all concept and role expressions respectively. When communicating with the knowledge base, the instances of the concrete children of these classes serialize themselves into the appropriate JSON representation.

The class `semantics.model.Concept` represents a concept atom, while `semantics.model.Role` represents a role atom. The exact mapping of description logic expressions to these classes can be found in table 6.2 in section 6.5.

6.2 Data Source Specification

The data source is specified on a per-class level. Within its scope, the knowledge base will use the ontology from the data source as context for all semantic concept types (section 6.3), type casing (section 6.4), queries (section 6.5) and projections (section 6.6).

Syntactically, the keyword `from` is used, followed by a string that specifies the path to the data source for the knowledge base (see listing 6.1). The keyword must be placed after any possible `extends` or `implements` specifications.

The data source propagates lexically and do not participate in inheritance. Inner classes inherit their data source from the first outer class that specifies one.

```
public class Outer extends Object implements Cloneable from "music.rdf" {
  // data source is music.rdf
  static class Inner {
    // still music.rdf
    static class Shadow from "wine.rdf" {
      // now it is wine.rdf
    }
    // back to music.rdf
  }
  // still music.rdf
}

class Child extends Outer {
  // no data source, propagation is only lexical
}
```

Listing 6.1 Data source specification and propagation.

A `from` declaration of an inner class shadows those of its outer classes (see listing 6.1).

It is legal to use multiple different data sources in the same program, but they behave as if they were separate instances of the λ_{DL} model. See the following sections on the effects this entails.

6.2.1 Syntax

As `from` is a valid identifier in normal Java code, turning it into a reserved word would break any existing code that uses it as a name. To prevent this, instead of recognizing it as a token in the scanner, the word is parsed as an identifier and manually compared to the expected keyword.

As the normal Java grammar does not normally allow an identifier to be placed in that position, this setup does not cause any conflicts with existing code. No new reserved words are introduced and `from` remains a valid identifier.

6.2.2 Frontend

The data source is used every time communication with the knowledge base is performed, as the context of a database is necessary for it to perform any of its functions. A missing data source specification causes a compile-time error, while an invalid data source will cause an error as soon as any actual communication with the knowledge base is attempted.

6.2.3 Backend

While the data source is used in code generation of other features to communicate with the knowledge base, the `from` declaration itself is not needed at run-time. Therefore no code is generated for it.

6.3 Semantic Concepts as Types

Expression	**Syntax**	**Texas Version**[2]	λ_{DL}
Everything (Top Concept)	`⊤`	`#T`	$\top$
Nothing (Bottom Concept)	`⊥`	`#F`	$\bot$
Full (Top Role)	`▽`	`#t`	—
Empty (Bottom Role)	`△`	`#f`	—
Atom	`«A»`	`<<<A>>>`	A
Nominal Concept	`⦃«a»⦄`	`{|<<<a>>>|}`	$\{a\}$
Negation	`¬C`	`-.C`	$\neg C$
Inversion	`R⁻`	`R^-`	R^-
Intersection	`C ⊓ D`	`C &&& D`	$C \sqcap D$
Union	`C ⊔ D`	`C ||| D`	$C \sqcup D$
Existential Quantification	`∃R·C`	`#ER...C`	$\exists R.C$
Universal Quantification	`∀R·C`	`#AR...C`	$\forall R.C$
Grouping	`[…]`		(...)

Table 6.1 Semantic concept type syntax.

Semantic concept type specifications look very similar to how they do in λ_{DL}, as can be seen in table 6.1. The reason they are not identical is to avoid conflict with existing Java syntax. These types can be used in any place that regular Java types can be used, including variable declarations, method arguments, generics parameters, wildcard parameters, casts and `instanceof` checks (see listing 6.2 for an example).

In case a user has trouble inputting the Unicode characters used in the type syntax, Texas variants[2] that do not require non-ASCII characters to be input are provided. There is merely a visual difference, both variants provide the exact same functionality.

[2]"Texas Operator: The ASCII variant of a non-ASCII Unicode operator or symbol. So described because "Everything's bigger in Texas."" – the Perl 6 glossary, `https://docs.perl6.org/language/glossary#Texas_operator`, accessed 2017-08-18.

All semantic concept types are subtypes of `semantics.model.Individual`, as if they extend the class. Subtype relationships between semantic concept types are resolved by the knowledge base. A pair of semantic concept types is considered equivalent if they are subtypes of each other, regardless of their structure.

Type-checking uses the data source of the enclosing class for reasoning about subtypes (see section 6.2). If no data source is in scope, a compile-time error is raised. Attempting to pass a concept-typed individual between classes that specify different data sources is a compile-time error. To intermix individuals from different ontologies, it is instead necessary to combine them ahead of time, for example by performing an ontology mapping [KS03, ES04], and specify the new, combined data source for both classes.

Implicit assignment of semantic concept types is possible if the appropriate subtype relationship exists. Otherwise, a cast or type case (see section 6.4) is necessary. This works analogously to regular polymorphic Java types.

Casts with semantic concept types work as expected: a run-time type check is performed and in case of failure an exception is thrown. The thrown object is an instance of `java.lang.ClassCastException`.

Similarly, `instanceof` tests with semantic types perform a run-time type check. If the given value is an `Individual` and the knowledge base determines it to be a member of the given type, the expression evaluates to true. Otherwise, it evaluates to false.

Semantic concept types undergo type erasure, they are turned into specifications of `semantics.model.Individual` during bytecode generation. The same limitations on method overloading apply as do with generic types.

```
public static Set<? extends ⊤> example(«:RedWine» ⊓ «:DryWine» redAndDry) {
  Set<«:Wine»> wines = new HashSet<>();
  wines.add(redAndDry);
  return wines;
}
```

Listing 6.2 Concept type arguments, wildcards, generics and implicit assignment.

6.3.1 Syntax

As the entire syntax of the semantic concept types uses tokens that do not exist in the regular Java types, there is no conflicts to worry about. The scanner was simply extended to recognize the new tokens and the parser to recognize semantic concept types where any other type access is allowed.

Both the Unicode and Texas variants of the various symbols used in the type specifications are resolved to the same token in the scanner. The parser sees no difference between them, making the implementation of these aliases trivial. The only exception is the `...` Texas operator, which Java already uses for variadic arguments. However, in the place where it is used, namely in the middle of a quantifier separating the role and concept arguments, a variadic argument operator would not make sense, as it can only appear at the end of a type parameter. As such, there is no ambiguity as to the meaning of the operator.

There is however potential conflict between semantic concept types and description logic expressions (see section 6.5). Especially type casts may look very similar to parenthesized DL statements. This issue is alleviated by using square brackets to group type expressions instead of parentheses and giving concept and role literals special delimiters instead of using, for instance, string literals or bare identifiers.

In a few cases where the parser would commit to an incorrect interpretation of `⊤`, `⊥` and quantifiers involving `▿` or `▵` as a partial DL expression instead of a concept type and does not backtrack out of them, the scanner must disambiguate the tokens beforehand. For that, tokens are stored in a lookahead buffer until a definitive decision can be made. This is analogous to how ExtendJ's Java 8 disambiguates, for instance, intersection casts from bit-and expressions [Hog14].

6.3.2 Frontend

In the ExtendJ compiler, semantic concept type declarations are defined as a subclass of class declarations. This means that only the relevant behaviors need to be specified: structural type identity, subtype relations using the knowledge base, access control (to make them accessible from anywhere) and type erasure.

As the compiler itself uses this interface to implement generic types [EH07a], this extension is simple and works in all contexts types may be used. Features such as type checking, wildcard types and method overload resolution simply fall out of the existing facilities of the compiler.

Semantic concept types are never actually declared like a class would, they are only reified upon access. However, due to JastAdd using the abstract syntax tree to hold all information [HM03], the declarations still need to be attached somewhere in the tree. To solve this, a similar approach to the ExtendJ non-null type extension [EH07b] is used: the declarations are attached to the class declaration of `semantics.model.Individual` as a non-terminal attribute.

6.3.3 Backend

As semantic concept type specifications undergo type erasure, there is little code generation work done with them. They are simply resolved to be specifications of `semantics.model.Individual` instead.

Casts and `instanceof` tests however do require some code to be generated so that their run-time checks can be performed during execution. A similar technique as described in [ÖH13] is used for code generation: instead of manually generating bytecode or rewriting the tree when it is first accessed, a new abstract syntax

```
if (x instanceof «:Wine») {
  getWineColor((«:Wine») x);
}
// ...is transformed into:
if (KnowBase.of("wine.rdf").instanceOf(new Concept(":Wine"), x)) {
  getInfluences(KnowBase.of("wine.rdf").cast(new Concept(":Wine"), x));
}
```

Listing 6.3 Tree transformation of casts and `instanceof` tests.

tree as a non-terminal attribute, containing only plain Java expressions. Bytecode generation is then delegated to that generated tree.

In this case, the expressions are simply transformed into method calls that implement the appropriate run-time behavior, an example of which can be seen in listing 6.3. The semantic concept types are transformed into an equivalent run-time object model compatible with the knowledge base.

Note that this transformation *only* happens during code generation, all static checks still operate on the proper, untransformed tree. This allows for errors to refer to their proper origin, as well as for full de-parsing of the tree with the original syntax intact [ÖH13].

6.4 Type Casing

```
switch-type (individual) {
  ∃«:hasMaker»⁻·⊤ producer {
    System.out.println(getWines(producer));
  }
  «:Winery» winery {
    System.out.println(winery.getName() + " is a winery without wines!");
  }
  default {
    System.out.println("There's only sour grapes here.");
  }
}
```

Listing 6.4 Type case example.

Type cases allow for cleaner run-time checks of several semantic concept types than a chain of `instanceof` tests and casts would. It also provides additional compile-time checking to ensure the cases are valid and ordered properly.

The keyword `switch-type` is used to introduce type casing, followed by the expression whose type is supposed to be checked and a block that contains the cases themselves. Each type case consists of a semantic concept type, an identifier to bind to and a block to execute when the type matches. The last block must be a `default` case, which takes neither a type nor an identifier. See listing 6.4 for an example.

There may be zero or more type cases, which are checked in order they are declared. If an earlier case's type subsumes a later case's type, a compile-time error is raised, as the later block would be unreachable.

When a type case matches the result of the given expression, its value is bound to a `final` variable with the given identifier and the block is executed. If none of the cases match, the `default` block is executed. There must be exactly one `default` case, and no other cases may be declared after it.

Only one case is ever executed, there is no fall-through like there is with a regular `switch` statement. Blocks cannot be exited by using `break`, as is the case with any non-loop block in Java.

Checking against `T` or any equivalent value would always match, thereby masking the default case. It is therefore disallowed. Checking against a type that is unsatisfiable would never match, and the associated block would never be executed. It is therefore also disallowed, which is consistent with the way Java handles dead code in other places.

6.4.1 Syntax

As opposed to the `from` keyword in section 6.2, the sequence `switch-type` is simply a keyword. As the only thing that can legally follow a `switch` keyword in regular Java code is an open parenthesis, no existing code could make use of this sequence in any other way. Therefore, full compatibility is retained.

Note that the word `type` is not a keyword on its own, the scanner recognizes it only as part of the `switch-type` token.

Anything inside the block of the type casing statement is simply new syntax that only exists in that context. Therefore extending the parser for it is simple, as it does not need to touch existing rules.

6.4.2 Frontend

While type casing introduces several new node types to the grammar, the static checks on them are very much straightforward and simply follow the rules given by the usage overview above.

Each type case carries a `final` variable declaration with it that is only visible to its associated block. The type check for binding this variable happens at run-time.

A type casing statement performs completability analysis similar to how a chain of if-else statements would: a type casing statement can complete normally if any of its blocks can complete normally. This information is also used in reachability analysis by the ExtendJ compiler.

6.4.3 Backend

```
switch-type (wine) {
  «:RedWine»   red   { return "pair " + red  .getName() + " with meat"; }
  «:WhiteWine» white { return "pair " + white.getName() + " with fish"; }
  «:RoseWine»  rose  { return "pair " + rose .getName() + " with rice"; }
  default            { return "don't drink colorless wine at all!";     }
}
// ...is transformed into:
{
  Individual #topic = wine;
  KnowBase   #kb    = KnowBase.of("wine.rdf");

  if (#kb.isMember(new Concept(":RedWine"), #topic)) {
    final Individual red = #topic;
    return "pair " + red.getName() + " with meat";
  }
  else if (#kb.isMember(new Concept(":WhiteWine"), #topic)) {
    final Individual white = #topic;
    return "pair " + white.getName() + " with fish";
  }
  else if (#kb.isMember(new Concept(":RoseWine"), #topic)) {
    final Individual rose = #topic;
    return "pair " + rose.getName() + " with rice";
  }
  else {
    return "don't drink colorless wine at all!";
  }
}
```

Listing 6.5 Tree transformation of type cases.

Code generation uses the same tree transformation as described in section 6.3.3. An example can be seen in listing 6.5.

To hold the value of the result of the given expression and the knowledge base connection, variables with names that cannot appear in regular Java code are used. This ensures they do not shadow any user-defined identifiers in the surrounding scope. This technique is used in a similar fashion in the Java 8 extension for ExtendJ [Hog14].

Otherwise, code generation is simply a matter of building a chain of if-else statements whose tests communicate with the knowledge base. As a degenerate case, a type casing statement that contains only a `default` block will simply evaluate the topic and then execute that block unconditionally.

6.5 Queries

```
Set<? extends «:Wine»> wines = query-for(":RedWine" ⊓ ":DryWine");
```

Listing 6.6 Query and description logic syntax.

Querying the knowledge base involves building a description logic expression first. The syntax is analogous to the description logic syntax for types from section 6.3, but in place of concept and role literals enclosed in «», any expression that evaluates to a `String` can be used. For convenience, instances of `semantics.model.Individual` used in DL expressions are turned into nominal concepts. Texas variants of the symbols may be used as well (see table 6.1).

Description logic operators are part of the normal set of Java operators. This means description logic expressions can appear anywhere any other expression can appear and dynamic queries can be constructed incrementally. As with any other expression, parentheses are used for grouping.

The intersection operator ⊓ has higher precedence than the union operator ⊔. Both these operator's precedence is just lower than shift operators. In particular, this means that `+` has higher precedence, allowing for string concatenation within an intersection or union expression.

The negation operator ¬ has the same precedence as the logical not operator `!`. The inversion operator ⁻ has the same precedence as postfix increment `++`. ⊤, ⊥, ▿ and ▵ are treated as literals.

Quantification operators ∃ and ∀ apply to the entire expression that is to their right. Parentheses around the entire expression may be used to delimit them. Atomic concept delimiters ⦃⦄ simply act as a bracketing construct, similar to parentheses.

All concept expressions extend from `semantics.model.Conceptual` and all role expressions extend from `semantics.model.Roleish`. The types for all description logic operations can be seen in table 6.2. Literal concepts and roles may be constructed by using `semantics.model.Concept` and `semantics.model.Role`, but this should generally not be necessary, as `String`s are automatically wrapped at compile-time.

The `query-for` operator is perform the actual query. It receives a concept expression as its argument in parentheses and returns a `java.util.Set` that contains the found elements as instances of `semantics.model.Individual`. See listing 6.6 for an example.

The knowledge base uses the ontology from the data source of the current scope as context for the query operation. See section 6.2 for details on that.

The actual type of the query operation depends on the expression it is given. Constant-foldable expressions, such as `":RedWine" ⊓ ":WhiteWine"`, will return a set of the equivalent semantic concept type, `Set<«:RedWine» ⊓ «:WhiteWine>»` in this case. Non-constant parts of the query are given an upper bound as described

Expression	Resulting Type
⊤	`semantics.model.Everything`
⊥	`semantics.model.Nothing`
▽	`semantics.model.Full`
△	`semantics.model.Empty`
⦃C⦄	`semantics.model.One`
¬C	`semantics.model.Negation`
R⁻	`semantics.model.Inversion`
C ⊓ D	`semantics.model.Intersection`
C ⊔ D	`semantics.model.Union`
∃R·C	`semantics.model.Existence`
∀R·C	`semantics.model.Universal`
`query-for(C)`	`java.util.Set<? extends semantics.model.Individual>`

Table 6.2 Types of description logic expressions.

in table 4.1. For example, a `query-for(":Wine" ⊓ ⦃wineId⦄)`, where `wineId` is some variable of type `String`, will have a type of `Set<«:Wine» ⊓ ⊤>`[3].

If a query is detected to be unsatisfiable at compile-time, an error is raised, as it would not be useful. At run-time when the entire query argument is known, satisfiability is *not* checked again. It is not considered an exceptional circumstance that dynamic queries may be unsatisfiable, they are simply executed.

6.5.1 Syntax

The description logic syntax re-uses the same tokens from semantic concept types (see section 6.3.1), just in expression context, rather than as type specifiers. There are no syntax limitations where the operators may be used, invalid usage is detected during type checking instead.

As the ExtendJ parser has only limited extensibility [ÖH13, Hog14], some patches are necessary for the operators to have the correct precedence. These adjust the operator precedence table, as well parser rules to fit union and intersection operations just before shift operations.

The `query-for` keyword works the same as the `switch-type` keyword from section 6.4.1: the entire sequence is recognized as a new token by the scanner. As the `for` keyword could never occur in a subtraction expression and `query` on its own is not a keyword, there are no conflicts with existing Java code.

[3]Which can be written as just `Set<«:Wine»>`, as the types are equivalent.

6.5.2 Frontend

As they are simply expressions, type-checking of description logic expressions and query arguments is straightforward and functions much like other operators in ExtendJ.

Recognizing constant parts of a query also simply fits into the existing compiler facilities: ExtendJ allows declaration of synthetic attributes to recognize constant expressions, as well as retrieving the constant value from such an expression.

Partial query types are given an upper bound by replacing non-constant parts of the query with special unknown concept or unknown role instances and later reducing them according to the rules set out in table 4.1.

This type is then used by the knowledge base to check if the query is unsatisfiable, and an error is raised if that is the case. The expression's resulting type becomes a `java.util.Set` parametrized with the partial, upper-bounded type.

6.5.3 Backend

```
«:Winery» winery = ...;
Set<? extends «:Wine»> wines = query-for(":Wine" ⊓ ∃":hasMaker"·winery);
// ...is transformed into:
Individual winery = ...;
Set<? extends Individual> wines = KnowBase.of("wine.rdf").query(
    new Intersection(new Concept(":Wine"), new Existence(
        new Role("hasMaker"), new One(winery.getIri()))));
```

Listing 6.7 Tree transformation of query and description logic expressions.

Once again, code generation uses the same kind of tree transformation as can be seen in section 6.3.3. An example for queries is shown in listing 6.7.

The generated code is simple: description logic expressions turn into constructors of the appropriate types. Expressions that would evaluate to `String`s are wrapped into `semantics.model.Concept` and `semantics.model.Role` respectively. Expressions evaluating to `semantics.model.Individual` have their IRI retrieved by calling `getIri` on them, which is then wrapped into `semantics.model.One`.

Operators turn into a construction of the appropriate model class, as listed in table 6.2. The arguments to these operators are not otherwise modified, as they are simply normal expressions.

The `query-for` operator itself is turned into a `query` method call against the knowledge base specified by the data source that is in scope (see section 6.2).

```
public static Set<∃«:hasMaker»·«:Winery»> getWines(«:Winery» winery) {
  return winery.(":hasMaker"⁻);
}
```

Listing 6.8 Projection with description logic syntax.

6.6 Projection

Projections are applied by using a dot operator, followed by an expression in parentheses. The expression must either be of type `semantics.model.Roleish` or a `String`, which will be implicitly wrapped into a role analogous to how it is done in queries. See listing 6.8 for an example.

A projection of the form `i.(e)`, where `i` is an individual of concept type `C` and `e` is an expression evaluating to a role `R`, has a result type of `java.util.Set<∃R⁻·C>`. To ensure the projection is valid, `C` must be a subtype of `∃R·⊤`. If necessary, this restriction can be circumvented by using an equivalent query instead of a projection (see listing 6.9).

```
private static Set<? extends Individual> getWinesFor(«:Winery» winery) {
  // Fails at compile-time, as «:Winery» is not a subtype of ∃":hasMaker"⁻·⊤:
  return winery.(":hasMaker"⁻);
  // An equivalent query may be used instead:
  return query-for(∃":hasMaker"·winery);
}
```

Listing 6.9 Circumventing projection check with an equivalent query.

Inference of upper bounds for non-constant role expressions works as described in table 4.1. As with queries, satisfiability of projections is checked at compile-time in that the expression `∃R⁻·C` must not be unsatisfiable.

The data source currently in scope (see section 6.2) is used by the knowledge base for performing the projection. No data source being in scope causes a compile-time error to be raised.

6.6.1 Syntax

Parsing a projection simply involves adding the option of a parenthesized expression to the right-hand side productions of the dot operator. As normally only identifiers are allowed here, there is naturally no conflict with any existing Java code.

6.6.2 Frontend

The static checking of projections is simple: their left-hand side is checked to be an instance of the appropriate concept type and the result type of is inferred via the same mechanism as is done for queries. The result type of the expression is simply filled in from those two types as described above.

6.6.3 Backend

Code generation is yet another case of tree transformation from section 6.3.3. See listing 6.10 for the way a projection is transformed.

The generated code simply involves a method call on the knowledge base with the data source that is currently in scope (see section 6.2). The description logic expressions are turned into their equivalent constructors (see section 6.5.3), with strings wrapped into `semantics.model.Role` constructors.

```
return wine.(":hasColor");
// ... is transformed into:
return KnowBase.of("wine.rdf").project(wine, new Role(":hasColor"));
```

Listing 6.10 Tree transformation of projection expressions.

Chapter 7

Applying the Language Extension

After the last chapter explained the inner workings, this one shows the Semantics4J language extension in action. It presents a sample application that makes use of several of the extension's features of an interactive environment, lays out a set of common errors relating to this application that static checking is able to detect at compile-time, and finally validates that backward compatibility with existing Java code is retained.

7.1 Sample Application

To test out the Semantics4J language extension in the context of a program interacting with a user and other systems, the wineseλrch application has been created[1]. It is a single-page web application, implemented using Aurelia[2] on the frontend and Spark[3] on the backend.

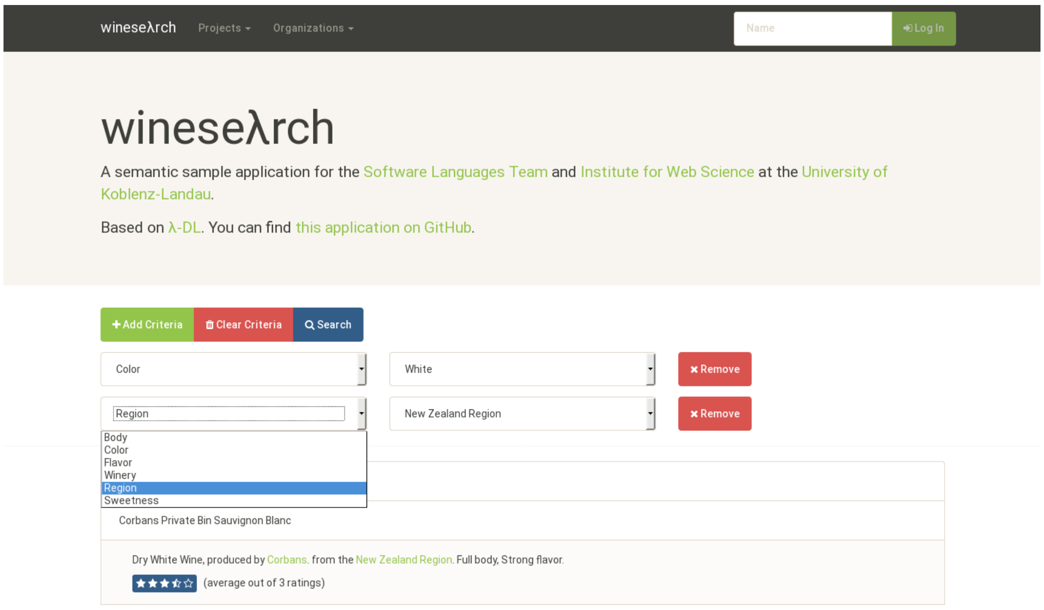
Figure 7.1 shows wineseλrch in use. The application allows users to find wines and information about them. Detailed information about a wine is given by pressing its entry in the list of results. Users can specify which wines they want to search by selecting a set of criteria that the wines should match, such as color, sweetness and the region they come from. For example, a selection of *Color: Red*, *Color: White* and *Region: Italian Region* will find all red and white wines from Italy.

As with all previous examples, the application is based on the W3C Wine Ontology. Semantic data is accessed via a Semantics4J-based model, which makes pervasive use of semantic concept types, queries and projections. Queries for wines

[1]The source code for wineseλrch can be found at https://github.com/hartenfels/winesearch, accessed 2017-08-18.

[2]http://aurelia.io/, accessed 2017-08-18.

[3]http://sparkjava.com/, accessed 2017-08-18.



wineseλrch

A semantic sample application for the Software Languages Team and Institute for Web Science at the University of Koblenz-Landau.

Based on λ-DL. You can find this application on GitHub.

Corbans Private Bin Sauvignon Blanc

Dry White Wine, produced by Corbans. from the New Zealand Region. Full body, Strong flavor.

(average out of 3 ratings)

Figure 7.1 The wineseλrch application.

are dynamically constructed based on user input, as a combination of unions, intersections and nominal concepts.

To show the interaction of semantic data with other data sources, wineseλrch also implements two additional features. One of them is retrieving additional information about wineries and regions via the Wikipedia REST API[4]. While there is also DBpedia[5], which provides a semantic ontology over Wikipedia's data, the REST API allows for fuzzy search and retrieving a simple summary of the most fitting page. This is used to show the user information about the winery or region they selected without knowing the exact title of the page.

The other feature is rating and reviews of wines. Users can log in by specifying their name and assign a score to wine and optionally leave a text reviewing it. The ratings are aggregated and their average shown with each wine. For this, a typical relational SQL database is used for storing and querying the data. When the application retrieves details about a wine, the information from the Wine Ontology and the SQL database are simply put together into the response.

For comparison, a SPARQL-based model has been constructed as well. It uses the Stardog 4.2.4 SNARL interface[6]. Stardog supports the necessary reasoning capabilities and is able to process OWL ontologies.

As there are no semantic concept types in regular Java code, this interface does not enable any static type checking, instead passing generic `String` and `Value` in-

[4]`https://en.wikipedia.org/w/api.php`, accessed 2017-08-18.
[5]`http://wiki.dbpedia.org/`, accessed 2017-08-18.
[6]`http://www.stardog.com/docs/4.0.5/#_java_programming`, accessed 2017-08-18.

stances to methods. Code size is notably different: the Semantics4J-based model's source[7] consists of 91 non-whitespace, non-comment lines of code, whereas the SPARQL-based model's source[8] takes up 172. The Stardog implementation also proved *significantly* slower, taking up to 15 minutes to load details for a single wine, whereas the Semantics4J implementation takes only around 3 seconds. Queries are cached by both Stardog and Semserv, so subsequent access is within a few seconds for both implementations.

7.2 Feature Comparison

This section shows, via a set of examples based on the wineseλrch application[9], how the Semantics4J language extension is able to catch several common errors at compile-time, while conventional semantic data interfaces in programming language do not provide any such assistance.

7.2.1 Typed Individuals

```
public static Map<String, ∃▽·«:Wine»> getWineInfo(«:Wine» wine) {
  Map<String, ∃▽·«:Wine»> info = new HashMap<>();
  info.put("body",   head(wine.(":hasBody"  )));
  info.put("color",  head(wine.(":hasColor" )));
  info.put("flavor", head(wine.(":hasFlavor")));
  info.put("maker",  head(wine.(":hasMaker" )));
  info.put("region", head(wine.(":locatedIn")));
  info.put("sugar",  head(wine.(":hasSugar" )));
  return info;
}
```

Listing 7.1 Method to retrieve properties of a wine, such as its body and color.

As one of wineseλrch's capabilities, users must be able to retrieve information about a specific wine, such as its color, flavor or region of origin. The users specifies the wine that they wish to receive the information about and the model is expected to return this information as a map. If no appropriate wine can be found, the model should return `null`.

Listing 7.1 shows a method `getWineInfo` for the Semantics4J interface, which retrieves the properties of a wine. Both its parameter and return value are as-

[7]`https://github.com/hartenfels/winesearch/blob/0c928ab26d/backend/src/main/java/semantics/example/SemanticsModel.java`, accessed 2017-08-18.

[8]`https://github.com/hartenfels/winesearch/blob/d6374884ab/backend/src/main/java/semantics/example/SparqlModel.java`, accessed 2017-08-18.

[9]...and actual errors the author made developing it.

signed the appropriate semantic concept types. The properties are retrieved via projections on the given wine.

Assuming that user input is given as a `String` in some variable `wineId`. Naively, one would do a `query-for(⦃wineId⦄)`, returning `null` if there is no result and otherwise passing the element to `getWineInfo`. However, this is erroneous: the user input may be any IRI, and therefore the query could return an element that is not actually a wine. Instead of returning the appropriate `null` value, the function would instead return a map containing useless wine-related information for something that is not a wine.

Semantics4J catches this mistake at compile-time: the return type of the query is a `java.util.Set<T>`, and `T` is not a subtype of `«:Wine»`. To fix this mistake, one should `query-for(":Wine" ⊓ ⦃wineId⦄)`, which will correctly return an empty set if the given IRI does not represent a wine. The return type of the query is a `Set<«:Wine» ⊓ T>`, whose elements are correctly subtypes of `«:Wine»` and therefore pass the type check.

For the SPARQL interface, one would perform a set of queries[10] of the format `SELECT ?body WHERE { ?wine :hasBody ?body }`, where `?wine` is bound to the `wineId`. Inputs and outputs are not typed, they consist of `String`s and generic semantic data `Value`s.

In this case, a missing check of `?wine rdf:type :Wine` is not detected at all. While defensive checks before executing the query could ensure that the input is a wine as a pre-condition, it would still defer the programmer error to run-time, rather than detecting the mistake at compile-time.

7.2.2 Injection Safety

SPARQL interfaces in general-purpose programming languages have potential for injection [OAA+10], analogously to the common issue of SQL injection. The implementor must take care that untrusted input is correctly sanitized, usually via parametrized queries or escaping mechanisms. This way, data can safely cross the language barrier from the programming language (such as Java) to SPARQL without changing its meaning to something potentially erroneous or even malicious.

Semantics4J on the other hand does not require any explicit parametrization or escaping. As the description logic syntax is *integrated* into Java, there is no language barrier to cross in the first place. Queries and projections are constructed directly using operators of the programming language. Injection safety naturally falls out of this, as variables in the code cannot expand and change the meaning of surrounding code like it happens in a SPARQL injection situation.

An example for this are the wine properties from the previous chapter. The SPARQL query `SELECT ?body { ?wine :hasBody ?body }` requires parametriza-

[10] A single query would require accounting for wines with missing properties, which do exist in the Wine Ontology. It causes Stardog to cancel the query due to an exceeded GC limit though, which is why the query is separated in the wineseλrch implementation.

tion of the `?wine` with the `wineId`. Simply concatenating the `wineId` into the query would *seemingly* work as well, but as it is untrusted, uncontrolled input to the program, it presents an injection vulnerability. For instance, concatenating a string with a value such as `"#"` into the query string would comment out the rest of the line and surely trigger a syntax error.

With Semantics4J on the other hand, the `query-for(":Wine" ⊓ wineId)` is already safe from injection by default, as `wineId` cannot somehow transform the syntax of the query expression, it is always just a variable. Subsequent projections on the retrieved wine, such as `wine.(":hasBody")` are safe as well, for the same syntactical reason.

7.2.3 Signature Validation

Semantics4J will automatically warn the user about concept atoms, role atoms and nominal concepts that are not part of the ontology's signature. Warnings are given at compile-time if possible, and always at run-time when they occur during execution.

This feature is able to catch various typos. For instance, take the `query-for("Wine" ⊓ ⦃wineId⦄)` – it is missing the prefixing colon on the wine concept IRI, it should be `":Wine"` instead. An appropriate warning will be emitted, telling the user that the concept `Wine` is not in the signature of `wine.rdf`. When the query is executed, an appropriate warning will be logged as well.

Conveniently, this feature compounds semantic type checking nicely as well. When presented with two almost identical-looking types in an error message that only differ in a forgotten colon, having a warning point to the mistake aids in finding the source of the error.

With the SPARQL interface on the other hand, no such signature checking occurs, the query will simply not find any matching elements and silently return an empty result set. This is compounded by the lack of static type safety: while Semantics4J would detect an attempted assignment of `«Wine»` to `«:Wine»`, a generic `Value` type will simply allow such an operation.

Signature validation could potentially be implemented in a plain SPARQL interface without any additional compile-time support, but the warnings would of course occur at run-time, even if the IRIs are actually constant in the query strings themselves.

7.2.4 Dynamically Constructed Queries

To search for the wines that match the user-specified criteria, wineseλrch must construct a query dynamically, based on user input. The searching function takes a list of criteria categories, each associated with a list of values for the category. The function is expected to return a set of wines that match the intersection of the

```
// From the following user input (formatted as JSON):
{":hasBody": [":Full"], ":hasColor": [":Red", ":White"]}
// The following query should be constructed:
query-for(":Wine" ⊓ (∃":hasBody"·":Full") ⊓ (∃":hasColor"·":Red" ⊔ ":White"))
```

Listing 7.2 Sample for a query constructed from user input.

union of each category's values. Listing 7.2 shows a sample input and expected resulting query.

As Semantics4J simply uses Java expressions to construct its queries, building a dynamic query works much like one would construct a dynamic numeric calculation. Partial queries can be stored in variables and incrementally built by amending to them. As queries are never translated into a query language string, no explicit parameter escaping is necessary. Listing 7.3 shows how one could build a query like that, while still retaining enough type information to ensure a return type of a `Set<«:Wine»>`.

For the SPARQL interface, query construction is much more complex, as can be seen in listing 7.4. To construct the query itself, the SPARQL query must be built by concatenating its pieces as strings. This can very easily lead to accidentally producing syntactically invalid queries that are only detected at run-time when they actually occur. Additionally, to avoid SPARQL injection from malicious user input, one must also construct a set of parameters to apply to the resulting query string, which are not validated at all.

```
public Set<«:Wine»> findWines(Map<String, String[]> searchArgs) {
  Conceptual dl = ⊤;
  // Iterate over each category.
  for (Map.Entry<String, String[]> arg : searchArgs.entrySet()) {
    String[] values = arg.getValue();
    // Build a union from all given values.
    Conceptual union  = ⦃values[0]⦄;
    for (int i = 1; i < values.length; ++i) {
      union ⊔= ⦃values[i]⦄;
    }
    // Aggregate the category and union.
    dl ⊓= ∃arg.getKey()·union;
  }
  // The result can still be typed by intersecting with a constant concept.
  return query-for(":Wine" ⊓ dl);
}
```

Listing 7.3 Dynamic query construction for wine searching.

A common alternative to manual string concatenation is the use of a query builder, such as the Apache Jena Query Builder[11] or LINQtoSPARQL[12], which hides the string concatenation behind a nicer interface. However, while it is harder to accidentally create syntactically invalid SPARQL queries, one must still take care of user-supplied inputs being escaped correctly.

7.3 Compatibility

The Semantics4J language extension is intended to purely *extend* the Java language, not incompatibly alter its existing semantics. Therefore, backward compatibility is intended to be kept in the sense that *valid* Java code retains its semantics even when using the Semantics4J compiler for it. No restrictions are made on *invalid* code, as no working code base should contain such code anyway.

To validate this goal, ExtendJ's regression test suite is used[13]. As of the current version of ExtendJ[14] and the regression test suite[15], all 1510 tests pass with the Semantics4J compiler. Therefore, despite having made significant modifications to ExtendJ, one can be reasonably sure that it still behaves in its intended way.

A consequence of this is that future updates to ExtendJ can be integrated into Semantics4J simply by rebuilding it on top of the new version. The regression test suites of ExtendJ and Semantics4J[16] can then be used again to validate that the upgrade went smoothly and did not break any functionality.

[11]`https://jena.apache.org/documentation/extras/querybuilder/index.html`, accessed 2017-07-09.

[12]`https://github.com/Efimster/LINQtoSPARQL`, accessed 2017-08-18.

[13]The test suite can be found at `https://bitbucket.org/extendj/regression-tests`, accessed 2017-08-18.

[14]Commit 8900768 from 2017-07-27.

[15]Commit 5b34037 from 2017-07-27.

[16]Semantics4J's test suite can be found at `https://github.com/hartenfels/Semantics4J/tree/master/src/test`, accessed 2017-08-18.

```java
public Set<Value> findWines(Map<String, String[]> searchArgs) {
  // Accumulator for the query string.
  List<String> lines  = new ArrayList<>();
  lines.add("SELECT ?wine WHERE {");
  lines.add("  ?wine rdf:type :Wine .");
  // Accumulator for the parameter values.
  List<String> params = new ArrayList<>();
  // Iterate over each category.
  for (Map.Entry<String, String[]> arg : searchArgs.entrySet()) {
    // Concatenate the union.
    String union = Arrays
      .stream(arg.getValue())
      .map(v -> {
        // Add the necessary parameters.
        int index = params.size();
        params.add(arg.getKey());
        params.add(v);
        // Construct a union entry with numbered parameters.
        return String.format("?wine ?param%d ?param%d", index, index + 1);
      })
      .collect(Collectors.joining(" UNION "));
    // Concatenate the union into the query string.
    lines.add(String.format("  { %s } .", union));
  }
  // Remember to close all delimiters.
  lines.add("}");
  // Prepare the query and then apply all parameters in the correct places.
  SelectQuery sq = prepareSparqlQuery(String.join("\n", lines));
  for (int i = 0; i < params.size(); ++i) {
    sq.parameter("param" + i, Values.iri(params.get(i)));
  }
  // Actually execute the query and extract the retrieved wines.
  return executeSparqlQuery(sq)
    .map(m -> m.get("wine"))
    .distinct()
    .collect(Collectors.toSet());
}
```

Listing 7.4 Dynamic SPARQL query construction.

Chapter 8

Concluding Remarks

In this final chapter, the work presented in this thesis is summarized, its limitations pointed out and possibilities for future work discussed.

8.1 Summary

In this thesis, the core concepts of the λ_{DL} language have been explored: semantic concepts as types, queries, projections and type casing. To integrate these features into a real-world general-purpose programming language, several extension mechanisms were considered, from which the JastAdd-based Java compiler ExtendJ was chosen as the most appropriate base to build upon. An extension has been developed that integrates the λ_{DL} concepts into the Java language, and extends them with additional features, such as signature validation, type casting and dynamic queries. Finally, this extension has been shown off in regards to how its features compare to conventional semantic data access via SPARQL, how it could be used in a practical application and shown to remain compatible with existing Java projects.

8.2 Limitations

Similar to λ_{DL} itself, types in the Semantics4J language extension are directly tied to the state of the ontology at the time of compilation. Therefore, modification of the semantic data requires the program to be re-compiled to ensure type-safety again. In particular, modification of the ontology from *within* the program, while desirable, is not available in the current implementation [LLS16].

Since semantic concept types undergo type erasure (see section 6.3), they cease to exist when the program is compiled into bytecode. Therefore, if the source code is not available to the compiler, semantic data types are not available in that module either. A possible mitigation for this would be to include metadata, such as annotations, into the compiled bytecode, from which the Semantics4J compiler could reconstruct the semantic data type information. A possible implementation would be to apply tree rewrites that transform the annotated metadata to semantic concept type accesses.

As the language extension is based upon the ExtendJ compiler, it inherits its features as well as limitations. As of the time of writing, the ExtendJ issue tracker[1] lists, for instance, several problems with regards to Java 8 type inference and lambda functions. However, when these issues are eventually fixed for ExtendJ, the language extension can simply be re-compiled against that new version as described in section 7.3 and in turn inherit these fixes as well.

Queries and projections make use of ExtendJ's existing constant folding features to determine their resulting type. However, this only distinguishes literal constant expressions and treats most variable references as non-constant, even if they are effectively final and assigned a constant value. This could be improved either by analyzing program flow leading up to query and projection operators, or by amending the ExtendJ constant folding algorithm itself.

8.3 Future Work

Support for using the language extension in an integrated development environment (IDE) could be developed, enabling features such as syntax highlighting and code completion for semantic concept types. A way that this could be accomplished is shown by the JModelica IDE [Mat09]: the compiler can serve as a baseline for an IDE plugin and additional static analysis functionality could be added easily, as JastAdd allows for further extension of the language extension itself [HM03].

As the query language is relatively limited in comparison to, for example, SPARQL, it could be another point of extension. As suggested for λ_{DL}, a simple addition would be allowing queries for roles in addition to concepts, returning a set of predicate-object pairs as a result. More complex query features are possible too, but it must be possible to properly assign a type to their results [LLS16].

Finally, a case study would be useful to explore the practical usability of the language extension. This could either be accomplished by developing a full application with it, or by modifying an existing application that makes use of semantic data via some other interface and replacing that aspect of it.

[1] https://bitbucket.org/extendj/extendj/issues?status=new&status=open&page=1, accessed 2017-08-18.

Bibliography

[Baa03] Franz Baader. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge university press, 2003.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[BCM99] Trevor Bench-Capon and Grant Malcolm. Formalising ontologies and their relations. In *International Conference on Database and Expert Systems Applications*, pages 250–259. Springer, 1999.

[BG00] Dan Brickley and Ramanathan V Guha. Resource description framework (RDF) schema specification 1.0. *W3C candidate recommendation*, March 2000.

[Bra97] Scott Bradner. RFC 2119: Key words for use in RFCs to indicate requirement levels. 1997.

[Bra14] Tim Bray. The JavaScript object notation (JSON) data interchange format. 2014.

[BSDTH16] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. In *European Symposium on Programming Languages and Systems*, pages 68–94. Springer, 2016.

[EH07a] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. *ACM Sigplan Notices*, 42(10):1–18, 2007.

[EH07b] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of nonnull types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[ERKO11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In *ACM SIGPLAN Notices*, volume 46, pages 391–406. ACM, 2011.

[ERRO12] Sebastian Erdweg, Felix Rieger, Tillmann Rendel, and Klaus Ostermann. Layout-sensitive language extensibility with SugarHaskell. In *ACM SIGPLAN Notices*, volume 47, pages 149–160. ACM, 2012.

[ES04] Marc Ehrig and Steffen Staab. QOM – quick ontology mapping. In *International Semantic Web Conference*, volume 3298, pages 683–697. Springer, 2004.

[GJS$^+$15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java language specification – Java SE 8 edition, 2015.

[GZ$^+$13] Francis Galiegue, Kris Zyp, et al. JSON Schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*, 2013.

[HB11] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[HM03] Görel Hedin and Eva Magnusson. JastAdd – an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[Hog14] Erik Hogeman. Extending JastAddJ to Java 8. 2014.

[HRS$^+$05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proceedings of the 14th international conference on World Wide Web*, pages 278–287. ACM, 2005.

[KAK15] Kazumasa Kumamoto, Toshiyuki Amagasa, and Hiroyuki Kitagawa. A system for querying RDF data using LINQ. In *Network-Based Information Systems (NBiS), 2015 18th International Conference on*, pages 452–457. IEEE, 2015.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.

[KPBP04] Aditya Kalyanpur, Daniel Jiménez Pastor, Steve Battle, and Julian A Padget. Automatic mapping of OWL ontologies into Java. In *SEKE*, volume 4, pages 98–103, 2004.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: The state of the art. *The knowledge engineering review*, 18(1):1–31, 2003.

[KSR+09] Georgi Kobilarov, Tom Scott, Yves Raimond, Silver Oliver, Chris Sizemore, Michael Smethurst, Christian Bizer, and Robert Lee. Media meets semantic web – how the BBC uses DBpedia and linked data to make connections. In *European Semantic Web Conference*, pages 723–737. Springer, 2009.

[LLS16] Martin Leinberger, Ralf Lämmel, and Steffen Staab. LambdaDL: Syntax and semantics (preliminary report). *CoRR*, abs/1610.07033, 2016.

[Loa98] Ralph Loader. *Notes on Simply Typed Lambda Calculus*. 1998.

[LSL+14] Martin Leinberger, Stefan Scheglmann, Ralf Lämmel, Steffen Staab, Matthias Thimm, and Evelyne Viegas. Semantic web application development with LITEQ. In *International Semantic Web Conference (2)*, pages 212–227, 2014.

[Mat09] Jesper Mattsson. *The JModelica IDE: Developing an IDE by Reusing a JastAdd Compiler*. Department of Computer Science, Lund University, 2009.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[MJ] Allie Mazzia and James Juett. Support for non-null types in Java.

[MK11] Ravi Bhushan Mishra and Sandeep Kumar. Semantic web reasoners and languages. *Artificial Intelligence Review*, 35(4):339–368, 2011.

[MME+10] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(2):4, 2010.

[MMI14] André Murbach Maidl, Fabio Mascarenhas, and Roberto Ierusalimschy. Typed Lua: An optional type system for Lua. In *Proceedings of the Workshop on Dynamic Languages and Applications*, pages 1–10. ACM, 2014.

[MPSP+09] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. OWL 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27(65):159, 2009.

[MVH$^+$04] Deborah L McGuinness, Frank Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[NCM03] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. Polyglot: An extensible compiler framework for Java. In *International Conference on Compiler Construction*, pages 138–152. Springer, 2003.

[OAA$^+$10] Pablo Orduña, Aitor Almeida, Unai Aguilera, Xabier Laiseca, Diego López-de Ipiña, and Aitor Gómez Goiri. Identifying security issues in the semantic web: Injection attacks in the semantic query languages. *Actas de las VI Jornadas Cientifico-Tecnicas en Servicios Web y SOA*, 51:4529–4542, 2010.

[OAC$^+$04a] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical report, 2004.

[OAC$^+$04b] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[ODG$^+$07] Eyal Oren, Renaud Delbru, Sebastian Gerke, Armin Haller, and Stefan Decker. ActiveRDF: Object-oriented semantic web programming. In *Proceedings of the 16th international conference on World Wide Web*, pages 817–824. ACM, 2007.

[ÖH13] Jesper Öqvist and Görel Hedin. Extending the JastAdd extensible Java compiler to Java 7. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 147–152. ACM, 2013.

[Paa09] Alexander Paar. *Zhi# – Programming Language Inherent Support for Ontologies*. PhD thesis, Karlsruhe Institute of Technology, 2009.

[PC12] Antonio Pisasale and Domenico Cantone. An experiment on the connection between the description logics' family $\mathcal{DL}\langle\forall_0^\pi\rangle$ and the real world. *arXiv preprint arXiv:1211.4957*, 2012.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. *W3C recommendation*, January 2008.

[PSW$^+$09] Fernando Silva Parreiras, Carsten Saathoff, Tobias Walter, Thomas Franz, and Steffen Staab. APIs *à gogo*: Automatic generation of ontology APIs. In *Semantic Computing, 2009. ICSC'09. IEEE International Conference on*, pages 342–348. IEEE, 2009.

[PV11] Alexander Paar and Denny Vrandečić. Zhi# – OWL aware compilation. *The Semantic Web: Research and Applications*, pages 315–329, 2011.

[SBT+13] Donald Syme, Keith Battocchi, Kenji Takeda, Donna Malayeri, and Tomas Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proceedings of the 2013 workshop on Data driven functional programming*, pages 1–4. ACM, 2013.

[SET09] Toby Segaran, Colin Evans, and Jamie Taylor. *Programming the Semantic Web: Build Flexible Applications with Graph Data*. O'Reilly Media, Inc., 2009.

[SJ02] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[SMH08] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A highly-efficient OWL reasoner. In *OWLED*, volume 432, page 91, 2008.

[SÖH14] Friedrich Steimann, Jesper Öqvist, and Görel Hedin. Multitudes of objects: First implementation and case study for Java. *Journal of Object Technology*, 13(5):1–1, 2014.

[SSV+] Klemens Schindlerf Schindlerf, Riccardo Solmim, Vlad Vergui, Eelco Visseri, Kevin van der Vlistk, Guido Wachsmuthi, and Jimi van der Woningl. Evaluating and comparing language workbenches.

[VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

[VS05] Max Völkel and York Sure. RDFReactor – from ontologies to programmatic data access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, page 55, 2005.

[WLHA+98] Lauren Wood, Arnaud Le Hors, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Gavin Nicol, Jonathan Robie, Robert Sutor, et al. Document object model (DOM) level 1 specification. *W3C Recommendation*, 1, 1998.

[WMS04] Chris Welty, Deborah L McGuinness, and Michael K Smith. OWL web ontology language guide. *W3C recommendation, W3C (February 2004)* `http://www.w3.org/TR/2004/REC-owl-guide-20040210`, 2004.

[Zen04] Matthias Zenger. Programming language abstractions for extensible software components. 2004.

[ZO01] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *ACM SIGPLAN Notices*, volume 36, pages 241–252. ACM, 2001.