The essence of functional programming on semantic data

Martin Leinberger[1], Ralf Lämmel[2], Steffen Staab[1,3]

[1]Institute for Web Science and Technologies, University of Koblenz-Landau, Germany
[2] The Software Languages Team, University of Koblenz-Landau, Germany
[3] Web and Internet Science Research Group, University of Southampton, England

Abstract. Semantic data fuels many different applications, but is still lacking proper integration into programming languages. Untyped access is error-prone. Mapping approaches cannot fully capture the conceptualization of semantic data. In this paper, we present λ_{DL}, a typed λ-calculus with constructs for operating on semantic data. This is achieved by the integration of description logics into the λ-calculus for both typing and data access or querying. The language is centered around several key design principles, in particular: (1) the usage of semantic conceptualizations as types, (2) subtype inference for these types, and (3) type-checked query access to the data by both ensuring the satisfiability of queries as well as typing query results precisely. The paper motivates the use of a designated type system for semantic data and it provides the theoretic foundation for the integration of description logics as well as the core formal definition of λ_{DL} including a proof of type safety.

1 Introduction

Semantic data allows for capturing knowledge in a natural manner. Its characteristics include the representation of conceptualizations inside the data and an entity-relation or graph-like description of data. Both, on their own and together, they allow for precisely specifying the knowledge represented within semantic data. A knowledge system manages semantic data and may infer new facts by logic inference. Different use cases are fueled by the semantic-data approach. The knowledge graphs of Google and Microsoft enhance Internet search. Wikidata [36] is an open source knowledge graph that stores structured data for Wikipedia. It consists of one billion statements and contains 1,148,230 different concepts and 2515 relations. The ontology defined by Schema.org[1] provides structure for data. This data is then used in search as well as personal assistants such as Google Now and Cortana. Google stores more than 3 trillion semantic statements crawled from the web. In the field of Life Sciences, semantic data was applied in the form of Bio2RDF[2], providing 11 billion triples. Semantic data has also interlinked large, varied data sources, such as provided by Fokus[3]

[1] `https://schema.org/`
[2] `http://bio2rdf.org/`
[3] `https://www.fokus.fraunhofer.de/en`

containing more than 200,000 different data sets. These examples demonstrate that semantic data models (e.g., RDF or OWL) are important for representing knowledge in complex use cases. In order to fully exploit the advantages of these data models, it is also necessary to facilitate programmatic access and operating over such data in programs.

As the running example, consider semantic data about music artists formalized in the description logic *ALCOI(D)*. Listing 1 shows that everyone, or rather every object, for which a `recorded` relation that connects the object to another entity of type `Song`, exists is considered to be a `MusicArtist` (Line 2). The object `beatles` is of type `MusicArtist` (Line 4) and `machineGun` is a `Song` (Line 5). The object `hendrix` has recorded the song `machineGun` (Line 6) and was influenced by the object `beatles` (Line 7).

```
1 // Conceptualization
2 ∃recorded.Song ⊑ MusicArtist
3 // Graph data
4 beatles : MusicArtist
5 machineGun : Song
6 (hendrix, machineGun) : recorded
7 (hendrix,beatles) : influencedBy
```

Listing 1: Initial example of semantic data.

The example shows several challenges we need to deal with when working with semantic data in a programming language. (1) Conceptualizations rely on a mixture of nominal (`MusicArtist`) and structural typing (`∃recorded.Song`). (2) It is also not uncommon to have a very general or no conceptualization at all, as exemplified by the `influencedBy` role that expresses that `hendrix` has been influenced by the `beatles`. (3) Additional, implicit statements may be derived by logical reasoning, e.g., in our running example `hendrix:MusicArtist` can be inferred. Another challenge is not illustrated: (4) In real data sources, the sheer size of potential types may become a problem. It is practically infeasible to explicitly convert all 1,148,230 different concepts of Wikidata into types of a programming language.

Integration of data models into programming languages can be achieved in different ways. The three most important are (1) via generic types, (2) via a mapping to the type system of a programming language, or (3) by using a custom type system. A generic approach (1) can represent semantic data using types such as GraphNode or Axiom (cf. [19]). While this approach can represent anything the data can model, it does not leverage static typing: such generic representations are not error-checked. Mapping approaches (2) such as [21] aim at mapping the data model to the type system of the programming language so that static typing is leveraged. However, the mixing of structural and nominal typing, inferred statements, and a high number of concepts worth mapping are problematic. We therefore propose a third, a novel approach: A type system designed for semantic data (3).

In this paper, we present λ_{DL}, a functional language for working with knowledge systems. λ_{DL} uses concept expressions such as $\exists$`recorded.Song` as types. This ensures that every conceptualization can be represented in the language and allows for typing values precisely. It avoids pitfalls of other approaches by forwarding typing and subtyping judgments to the knowledge system, thereby allowing facts to be considered only if required. Lastly, the language contains a simple querying mechanism based on description logics. The querying mechanism allows for checking of satisfiability of queries as well as for typing the query results in the programming language. As a result, λ_{DL} provides a type-safe method of working with semantic data.

To highlight a simple kind of error that type checking can catch, consider a function `f` that takes $\exists$`influencedBy.`$\top$ as input. In other words, the functions accepts all entities for which an `influencedBy` relation exists.Using a query-operator that searches for entities in the data, a developer might simply query for music artists because he has seen that `hendrix` has an influence. Applying any value of the result set to the function `f` can cause runtime errors, as not all music artists have a known influence. Typing in λ_{DL} is precise enough to detect such errors (see Listing 2).

```
1 let f = λ(x:∃influencedBy.⊤) . x.influencedBy in
2   f (head (query MusicArtist))
```

Listing 2: Rejected code—music artist is not a subtype of $\exists$`influencedBy.`$\top$.

In summary, the main contributions of the paper are as follows:

1. We motivate and describe λ_{DL}, a language containing constructs for working with semantic data. In particular, we provide typing, querying constructs and a typecase. Semantics of these constructs rely on description logics, the theoretical foundations of semantic data.
2. We present a formal proof of type safety for λ_{DL}. We highlight how design decisions in λ_{DL} solve many of the problems that occur when dealing with semantic data and allow for a straightforward proof.

As we extend a standard λ-calculus, large parts of the semantics and proof of type safety are routine. We therefore focus on cases particular to our language. The full rules and complete proof can be found in the technical report[4]. Along with the technical report, we also provide a prototypical implementation to show the feasibility of the presented theories in practice.

Road-map of the paper The remaining paper is organized as follows. In Section 2, we introduce description logics as the theoretic foundation of semantic data. In Section 3, we illustrate λ_{DL} with an extension of the running example and an informal view on the calculus. In Section 4, we describe the core language and its evaluation rules. In Section 5, we describe the type system. In Section 6, we provide a proof of type soundness. In Section 7, we examine related work. In Section 8, we conclude the paper including a discussion of future work.

[4] `https://west.uni-koblenz.de/lambda-dl`

2 Description Logics

Semantic data is often formalized in the RDF data model or in the more expressive Web Ontology Language (OWL[5]). Formal theories about the latter are grounded in research on description logics. Description logics is a family of logical languages for describing conceptual knowledge and graph data. All description logic languages are sub-languages of first-order predicate logic. They are defined to allow for decidable or even PTIME decision procedures. Their usefulness for modeling semantic data has been shown with such diverse use cases as reasoning on UML class diagrams [6], semantic query optimization on object-oriented database systems [4], or improving database access through abstraction [10].

Syntax and Semantics Semantic data, also called a knowledge base, comprises of a set of description logics axioms that are composed using a signature $Sig(\mathcal{K})$ and a set of logical and concept operators and comparisons. A signature Sig of a knowledge base $\mathcal{K}$ is a triple $Sig(\mathcal{K}) = (\mathcal{A}, \mathcal{Q}, \mathcal{O})$ where $\mathcal{A}$ is a set of concept names, $\mathcal{Q}$ is a set of role names, and $\mathcal{O}$ is a set of object names. DL uses Tarskian-style, interpretation-based semantics. An interpretation $\mathcal{I}$ is a pair consisting of a non-empty universe $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that maps each object $a, b \in \mathcal{O}$ to an element of the universe. Furthermore, it assigns each concept name $A \in \mathcal{A}$ a set $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ and each role name $Q \in \mathcal{Q}$ to a binary relation $Q^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. In our running example, the signature of Listing 1 contains the concepts[6] `MusicArtist` and `Song`, the roles `recorded` and `influencedBy` as well as the objects `beatles`, `hendrix`, and `machineGun`. An interpretation $\mathcal{I}$ could map objects like `hendrix` to their real-life counterparts, e.g., the artist Jimi Hendrix. Furthermore, the interpretation of concept `MusicArtist` might be $\texttt{MusicArtist}^{\mathcal{I}} = \{\texttt{hendrix}, \texttt{beatles}\}$, and the interpretation of `Song` might be $\texttt{Song}^{\mathcal{I}} = \{\texttt{machineGun}\}$. The interpretation of the `recorded` role might be $\texttt{recorded}^{\mathcal{I}} = \{(\texttt{hendrix}, \texttt{machineGun})\}$ and $\texttt{influencedBy}^{\mathcal{I}} = \{(\texttt{hendrix}, \texttt{beatles})\}$.

Given these element names, complex expressions, e.g. as highlighted by Listing 1, can be built. For the course of the paper, the specific description logics dialect needed to cover all necessary constructs is *ALCOI*, consisting of the most commonly used *Attributive Language with Complements* plus the addition of nominal concept expressions and inverse role expressions. Table 1 summarizes syntax and semantics of role expressions represented through the metavariable R. A role expression is either an atomic role or the inverse of a role expression.

Concept expressions are composed from other concept expressions and may also include role expressions. Concept expressions, represented through the metavariables C and D, are either atomic concepts, $\top$, $\bot$ or the negation of a concept.

[5] `https://www.w3.org/OWL/`

[6] As common in description logics research, we use "concept C" to refer to both the concept name C and the interpretation of this concept name C^I, unless the distinction between the two is explicitly required. Likewise, we do for role names and object names.

Role Expression	Syntax	Semantics
Atomic Role	Q	$Q^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Inverse	R^{-}	$\{(b,a) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}$

Table 1: Role expressions and associated semantics.

Concept Expression	Syntax	Semantics
Nominal concept	$\{ a \}$	$\{a^{\mathcal{I}}\}$
Atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C$
Intersection	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Union	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Existential Quantification	$\exists R.C$	$\{a^{\mathcal{I}} \in \Delta^{\mathcal{I}} \mid \exists b^{\mathcal{I}} : (a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}} \wedge b^{\mathcal{I}} \in C^{\mathcal{I}}\}$
Universal Quantification	$\forall R.C$	$\{a^{\mathcal{I}} \in \Delta^{\mathcal{I}} \mid \forall b^{\mathcal{I}} : (a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}} \wedge b^{\mathcal{I}} \in C^{\mathcal{I}}\}$

Table 2: Concept expressions and associated semantics.

Concept expressions can also be composed from intersection or through existential and universal quantification on a role expression. An example of such a concept expression from Listing 1 is the concept $\exists$`recorded.Song` that describes the set of objects, which have recorded at least one song. Lastly, it is also possible to define a concept by enumerating its objects. This constitutes a nominal type in description logics and allows the description of sets such as the one only containing `hendrix` and the `beatles` through the expression {`hendrix`} $\sqcup$ {`beatles`}. Table 2 summarizes the syntax and semantics of concept expressions.

Furthermore, in the context of programming with semantic data, it makes sense to add additional data types such as string or integer. We then arrive at the language *ALCIO(D)*, the language *ALCIO* plus the addition of data types for constructing knowledge bases. In the OWL standard, the use of XSD[7] data types is common. We therefore also include XSD data types wherever it is appropriate. As an example, consider the concept expression $\exists$`artistName.xsd:string` describing the set of all objects having an artist name that is a string. As the integration of such smaller, closed set of data types can be achieved via mappings to appropriate types in the programming language, we do not go into details about them in the remainder of the paper.

Given such concept (and datatype) expressions, we may now define semantic statements, also called a knowledge base, as pointed out before. A knowledge base $\mathcal{K}$ is a pair $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consisting of the set of terminological axioms $\mathcal{T}$,

[7] `https://www.w3.org/TR/xmlschema-2/`

Name	Syntax	Semantics
Concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Concept equality	$C \equiv D$	$C^{\mathcal{I}} = D^{\mathcal{I}}$
Concept assertion	$a : C$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
Role assertion	$(a, b) : R$	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
Object equivalence	$a \equiv b$	$a^{\mathcal{I}} = b^{\mathcal{I}}$

Table 3: Terminological and assertional axioms.

$$\text{Domain}(R, C) \stackrel{\text{def}}{=} \exists R.\top \sqsubseteq C$$
$$\text{Range(R,C)} \stackrel{\text{def}}{=} \top \sqsubseteq \forall R.C$$

Fig. 1: Syntactical abbreviations for DL.

the conceptualization of the data and the set of assertional axioms $\mathcal{A}$, the actual data. Schematically, a knowledge base can express that two concepts are either equivalent or that two concepts are in a subsumptive relationship. In terms of actual data, objects can either express that belong to a certain concept or that they are related to another object via a role. Furthermore, it is possible to axiomatize that two objects are equivalent. Table 3 summarizes syntax and semantics of possible axioms in the knowledge base.

Even weak axiomatizations such as RDFS[8] allow for the definition of domains and ranges of roles used in the ontology. As shown in Fig. 1, Domain and Range definition can be defined as abbreviations of axioms built according to Table 3.

Using our running example, we can now define a more sophisticated knowledge base (Listing 3). We assume everyone who has recorded a song to be a music artist, but not all music artists have recorded one (Line 2). Music artists who have been played at a radio station however must have recorded a song (Line 3–4). Music groups are a special kind of music artists (Line 5). Every music artist has an artist name, which is always of type xsd:string (Line 6 and 7). As might happen when semantic data is crawled from the Web, a role like `influenceBy` might not be defined in the schema. Thus, it remains a role that is not restricted by any terminological axiom. The actual data includes descriptions of the `beatles` which are a music group (Line 9), `machineGun` which is a song (Line 10) `coolFm` which is a radio station (Line 11). `machineGun` has been recorded by `hendrix` (Line 12), who has been `influencedBy` the `beatles` (Line 13). Lastly, we know that both, `hendrix` and `beatles` have been played by `coolFm` (Line 14–15). It is not explicitly stated that `hendrix` is a music artist. Furthermore, even though we know that the music group `beatles` has been played at `coolFm`, we do not know any song that they recorded.

```
1 // Conceptualization
```

[8] RDF Schema, one of the weakest forms of terminological axioms.

```
2  ∃recorded.Song ⊑ MusicArtist
3  MusicArtist ⊓ ∃playedAt.RadioStation ⊑
4    ∃recorded.Song
5  MusicGroup ⊑ MusicArtist
6  MusicArtist ⊑ ∃artistName.⊤
7  Range(artistName, xsd:String)
8  // Graph data
9  beatles : MusicGroup
10 machineGun : Song
11 coolFm : RadioStation
12 (hendrix, machineGun) : recorded
13 (hendrix, beatles) : influencedBy
14 (hendrix, coolFm) : playedAt
15 (beatles, coolFm) : playedAt
16 (hendrix, "Jimmy Hendrix") : artistName
17 (beatles, "The Beatles") : artistName
```

Listing 3: Advanced example of semantic data.

As illustrated by the example, ALCIO(D) is a description logics language which is already rather expressive to describe complex concept and object relationships. As we want to focus on the "essence of programming with semantic data", we refrain from using more powerful languages, such as OWL2DL, as this would distract from the core contributions of this paper without significantly changing its methods.

Inference In terms of inference, interpretations have to be reconsidered. Axioms built according to Table 3 may or may not be true in a given interpretation. An interpretation I is said to satisfy an axiom F, if its considered to be true in the interpretation. The notation $I \models F$ is used to indicate this. An interpretation I satisfies a set of axioms $\mathcal{F}$, if $\forall F \in \mathcal{F} : I \models F$. An interpretation that satisfies a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$, written $I \models \mathcal{K}$ if $I \models \mathcal{T}$ and $I \models \mathcal{A}$, is also called a model. For an axiom to be inferred from the given facts, the axiom needs to be true in all models of the knowledge base (see Def. 1).

Definition 1 (Inference). *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a knowledge base, F an axiom and $\mathcal{I}$ the set of all interpretations. F is inferred, written $\mathcal{K} \models F$, if $\forall I \in \mathcal{I} : I \models \mathcal{K}$ then $I \models F$.*

An example of this is the axiom `hendrix:MusicArtist`. `hendrix` has recorded a song and must therefore be element of `∃recorded.Song`. As `∃recorded.Song ⊑ MusicArtist` must be true in all models, `hendrix` must also be element of `MusicArtist`. A knowledge system might introduce anonymous objects to fulfill the explicitly given axioms. Take the object `beatles` as an example. The object is a music artist and has been played in the radio. Therefore, according to lines 3–4 in the example, they must have recorded a song. However, the knowledge system does not know any song recorded by them. It will therefore introduce at least one anonymous object representing this song in order to satisfy the axioms.

Queries Interaction between the programming language and the knowledge system can be realized via querying. Two basic forms of queries can be distinguished. Queries that check whether an axiom is true have already been introduced in the previous paragraph ($\mathcal{K} \models F$). A more expressive form of querying introduces variables, to which the knowledge system responds with unifications for which the axiom is true. Querying introduces variables, to which the knowledge system responds with unifications for which the axiom is known to be true (see Def. 2).

Definition 2 (Querying with variables). *Let $\mathcal{K}$ be a knowledge base and C a concept expression. The set of all objects a such that $a : C$ is true is then $\{?X | \mathcal{K} \models ?X : C\}$, where $?X$ represents a variable being unified with said objects.*

As an example, consider the query $\mathcal{K} \models ?X$: `MusicArtist`, the variable $?X$ is unified with all objects that belong to the concept `MusicArtist`. However, this form of query can be problematic as, depending on the knowledge system, an infinite number of unifications might exist. Consider the knowledge base in Listing 4. A person is someone who has a father who is again a person (Line 1). An object `someone` is defined to be a person (Line 2).

```
1 Person ⊑ ∃hasFather.Person
2 someone : Person
```

Listing 4: Infinitely large knowledge system.

If `someone` is a person, then he or she must have an father which is a anonymous object and a person himself, again implying that this anonymous object has a father. A query $\mathcal{K} \models ?X$: `Person` therefore yields an infinite number of unifications. We therefore use a simple form of so called DL-safe queries (cf. [27]), which restrict unifications to objects defined in the signature (see Def. 3).

Definition 3 (DL-safe queries). *Let $\mathcal{K}$ be a knowledge base, $Sig(\mathcal{K}) = (\mathcal{A}, \mathcal{Q}, \mathcal{O})$ its signature and C a concept expression. The set of all objects for which $a : C$ is true and that are not anonymous can be queried by $\{?X | \mathcal{K} \models ?X : C \wedge ?X \in \mathcal{O}\}$.*

In case of the example shown in Listing 4, only the object `someone` would be returned, even though anonymous objects are considered for inferencing.

Open World and No Unique Name assumption Semantic data employs an open world semantics. Axioms are *true* if they are true in all models of the knowledge base. Likewise, an axiom is *false* if they are false in all models of the knowledge. Contrary to a closed world, axioms that are true in some models, but false in others are not false but rather *unknown*. This allows the modeling of incomplete data without inconsistencies. Furthermore, there is no unique name assumption. Two syntactically different objects might be equivalent. As an example, consider the two objects `prince` and `theArtistFormerlyKnownAsPrince`. While they are syntactically different, they might be semantically equivalent.

3 $\lambda_{\mathbf{DL}}$ in a nutshell

Developing applications for knowledge systems, as introduced in the previous section, is difficult and error-prone. λ_{DL} has been created to achieve a type-safe way of programming with such data sources.

3.1 Key design principles

Concepts as types Type safety can only be achieved if terms are typed precisely. This is only possible if the conceptualizations of semantic data are usable in the programming language. Therefore, concept expressions must be seen as types in the language.

Subtype inferences The facts about subsumptive relationships between concepts must be added to the system during the type checking process by forwarding these checks to the knowledge system. This allows for taking inferred statements into account and avoids problems with large number of conceptualizations.

Typing of queries To avoid runtime errors, queries must be properly type-checked. Queries can be checked in two ways: First, unsatisfiable queries must be rejected. This means that queries for which no possible A-Box instance can produce a result are detected and treated as an error. Second, usage of queries must be type-safe, meaning that the query result must be properly typed. Queries always return lists in λ_{DL}.

DL-safe queries A knowledge system might introduce anonymous objects to satisfy axioms. In the worst case, this can lead to infinitely large query results. However, very little information can be gained of such objects aside from their existence. As shown in Def. 3, λ_{DL} relies on a simplified form of DL-safe queries. Queries are enforced to be finite by only allowing unifications with known objects, even though this might lead to empty result sets in some cases.

Open-world querying When looking at inferencing, axioms may be *true*, *false* or *unkown*. For simplicity, λ_{DL} considers axioms to be true only if the axiom is *true* in all models. In other cases, the axiom is considered false. While this view is close to a developers expectation, it also introduces the side effect that union of two queries such as `query` C and `query` $\neg C$ does not yield all objects. For some objects, it is simply unknown whether they belong to either C or $\neg C$.

3.2 Example use case

Consider an application that works on the knowledge system defined in Listing 3. Four necessary functions should be implemented: First, the application should query for all music artists that have recorded a song. Second, the application should provide a mapping from a music artist to the list of their songs. Third, a mapping from a music artist to his artist name must be created. Fourth, the

application should display all influences of an artist—therefore a mapping from a music artist to his influences is needed. However, these influences should also be human-readable, meaning that they should also be mapped to their name.

The first requirement is implemented by the querying mechanism in λ_{DL}. The necessary list of music artists that have recorded at least one song can be queried using `MusicArtist` $\sqcap$ $\exists$`recorded.Song` (see Listing 5). Applied to a knowledge system working on the facts in Listing 3, this yields a list containing both `hendrix` and `beatles`. This expression is typed by the concept expression used in the querying, assigning a type of (`MusicArtist` $\sqcap$ $\exists$`recorded.Song`) list to the evaluation result.

```
1 query MusicArtist ⊓ ∃recorded.Song
```

Listing 5: Querying for music artists that have recorded a song.

Mapping a member of this list to his or her recorded songs can be done using role projections. The input type for such a mapping function is $\exists$`recorded.Song` which is a super type of `MusicArtist` $\sqcap$ $\exists$`recorded.Song`. Listing 6 shows the code for the mapping function. As mentioned before, for the object `beatles`, the semantic data does not contain any recorded songs, even though such a song must exist. The anonymous object introduced by the knowledge system is removed and an empty list is returned. Yet, the developer knows that an anonymous object must exist and that the knowledge system might know this song at some point in the future—otherwise typing would have rejected the function application.

```
1 let getRecordings = λ(a:∃recorded.Song).
2   a.recorded
```

Listing 6: Mapping to the recordings.

A function mapping a music artist to his name is again built by role projections. As our knowledge systems claims that every music artist has an artist name (Listing 3, line 5), the input type for this function can be the music artist concept. Additionally, the knowledge system states that the returned list of values are all of type string. We can therefore simply take the head of the returned list. Listing 7 shows the code of the mapping function. However, this code also shows a problem λ_{DL} still faces—if the knowledge system would not know the name of an artist, the resulting list would be empty and the code would still produce a runtime error.

```
1 let getArtistName = λ(a:∃artistName.xsd:string).
2   head (a.artistName)
```

Listing 7: Mapping a artist to his name.

The last requirement, mapping a music artist to his influences introduces casting, as music artists are not in a direct subtype relation to `influencedBy.`$\top$. This casting is important, as simply allowing the projection could cause runtime errors if, e.g., used on the object `beatles`. λ_{DL} provides a typecase for this use case. Listing 8 shows the code for such a mapping from a `MusicArtist` to an influence.

In case that the argument of the function is of type `influencedBy.⊤`, the actual mapping function is applied to the value—otherwise, an empty list is returned.

```
let getArtistInfluences = λ(artist:MusicArtist).
  case artist of
    type ∃influencedBy.⊤ as x -> getInfluences x
    default nil[∃influencedBy⁻.MusicArtist]
```

Listing 8: Casting a music artist to `influencedBy.⊤`.

The function computing the actual influences can use a projection and then apply a function that converts influences to their human-readable name However, this getting the name of an influence is problematic due to the weak schematic restrictions of the `influencedBy` role. The code must therefore proceed on a case by case basis. If the influence is a music artist, the projection to the human-readable string is known. Otherwise, the influence cannot be converted. Listing 9 shows the complete code for the function.

```
let getInfluences = λ(obj:∃influencedBy.⊤).
  let toName = λ(x:∃influencedBy⁻.⊤).
    case x of
      type MusicArtist as y -> getName y
      default "cannot convert to name"
  in letrec getNames:(∃influencedBy⁻.⊤ list -> string list) =
    λ(source:∃influencedBy⁻.⊤ list) .
      if (null source)
        then nil[string]
        else cons (toName (head source)) (getNames (tail source))
  in
    getNames obj.influencedBy
```

Listing 9: Mapping influences to their human-readable representations.

4 Core language

Syntax Our core language λ_{DL} (Fig. 2) is a simply typed call-by-value λ-calculus. Terms of the language include let-statements, a fixed point operator for recursion, function application and if-then-else expressions. Constructs for lists are included in the language: **cons**, **nil** including a type parameter, **null**, **head** and **tail**. Specific to our language is the querying construct for selecting data in the knowledge system based on a concept expression and projections from an object to a set of objects using role expressions. We use a typecase constructs that provides branch control based on types. It contains an arbitrary number of cases plus a default case. If a branch matches, the object is considered to be of the matched type inside the case itself. It therefore acts as a type-safe casting construct.

$$
\begin{array}{lr}
t ::= & (\textit{terms}) \\
\quad \textbf{let } x = t \textbf{ in } t & \text{(let binding)} \\
\quad \mid \textbf{fix } t & \text{(fixed point of } t\text{)} \\
\quad \mid t\ t & \text{(application)} \\
\quad \mid \textbf{if } t \textbf{ then } t \textbf{ else } t & \text{(if-then-else)} \\
\quad \mid \textbf{cons } t\ t & \text{(list constructor)} \\
\quad \mid \textbf{null } t & \text{(test for empty list)} \\
\quad \mid \textbf{head } t & \text{(head of a list)} \\
\quad \mid \textbf{tail } t & \text{(tail of a list)} \\
\quad \mid \textbf{query } C & \text{(query)} \\
\quad \mid t.R & \text{(projection)} \\
\quad \mid \textbf{case } t \textbf{ of} & \text{(typecase)} \\
\quad\ \ \overline{\textit{case}} & \text{(typecases)} \\
\quad\ \ \textbf{default } t & \text{(default case)} \\
\quad \mid t = t & \text{(equivalence)} \\
\quad \mid x & \text{(identifier)} \\
\quad \mid v & \text{(value)}
\end{array}
$$

$$
\begin{array}{lr}
v ::= & (\textit{values}) \\
\quad a & \text{(object)} \\
\quad \mid \textbf{nil[}T\textbf{]} & \text{(empty list)} \\
\quad \mid \textbf{cons } v\ v & \text{(list constructor)} \\
\quad \mid \lambda(x:T).t & \text{(abstraction)} \\
\quad \mid p & \text{(primitive value)}
\end{array}
$$

$$
\begin{array}{lr}
p ::= & (\textit{primitive values}) \\
\quad \text{true} & \text{(true)} \\
\quad \mid \text{false} & \text{(false)}
\end{array}
$$

$$
\textit{case} ::= \textbf{type } C \textbf{ as } x \text{ -> } t \qquad (\textit{typecase})
$$

$$
\begin{array}{lr}
T ::= & (\textit{types}) \\
\quad C & \text{(concept type)} \\
\quad \mid T \to T & \text{(function type)} \\
\quad \mid T \text{ list} & \text{(list type)} \\
\quad \mid \Pi & \text{(primitive types)}
\end{array}
$$

$$
\begin{array}{lr}
\Pi ::= & (\textit{primitive types}) \\
\quad \text{bool} & \text{(boolean)}
\end{array}
$$

$$
\begin{array}{lr}
\Gamma ::= & (\textit{context}) \\
\quad \emptyset & \text{(empty context)} \\
\quad \mid \Gamma, x:T & \text{(type binding)}
\end{array}
$$

Fig. 2: Syntax (terms, values, types) of λ^{DL}.

$$
\textbf{letrec } x : T_1 = t_1 \textbf{ in } t_2 \stackrel{\text{def}}{=} \textbf{let } x = \textbf{ fix } (\lambda x : T_1.t_1) \textbf{ in } t_2
$$

Fig. 3: Syntactical abbreviations of λ^{DL}.

We use an overbar notation to represent sequences of syntactical elements. That is, $\overline{a}$ stands for $a_1, a_2, ..., a_n$. As DL has no unique name assumption, objects can be syntactically different but semantically equivalent. Therefore, we also included the equality operator in our representation. Values (v) include objects defined in the knowledge base, nil and cons to represent lists, λ-abstractions and primitive values. λ-abstractions indicate the type of their variable. In terms of primitive values, we assume data types such as integers and strings, but omit routine details. To illustrate them, we usually just include booleans in our syntax. Types (T) consist of concept expressions built according to Table 3, type constructors for function and list types and primitive types. Additionally, we use a typing context to store type bindings for λ-abstractions. To simplify recursion, we also define a letrec as an abbreviation of the fixpoint operator (see Fig. 3).

Semantics The operational semantics is defined using a reduction relation, which extends the standard ones. Reduction of lists and terms not related to the knowledge bears no significant difference from rules as, e.g., defined in [32]. We there-

$$\textbf{query}\ C \to \sigma(\{?X \mid ?X \in \mathcal{O} \wedge \mathcal{K} \models ?X : C\}) \qquad \text{[E-QUERY]}$$

$$a.R \to \sigma(\{?X \mid ?X \in \mathcal{O} \wedge \mathcal{K} \models (a, ?X) : R\}) \qquad \text{[E-PROJV]}$$

$$\frac{t_1 \to t_1'}{t_1.R \to t_1'.R} \qquad \text{[E-PROJ]}$$

$$\frac{\mathcal{K} \models a \equiv b}{a{=}b \to \text{true}} \qquad \text{[EQ-NOMINAL-TRUE]}$$

$$\frac{\mathcal{K} \not\models a \equiv b}{a{=}b \to \text{false}} \qquad \text{[EQ-NOMINAL-FALSE]}$$

$$p_1{=}p_1 \to \text{true} \qquad \text{[EQ-PRIM-TRUE]}$$

$$\frac{p_1 \neq p_2}{p_1{=}p_2 \to \text{false}} \qquad \text{[EQ-PRIM-FALSE]}$$

$$\frac{t_1 \to t_1'}{t_1 = t_2 \to t_1' = t_2} \qquad \text{[E-EQ1]}$$

$$\frac{t_2 \to t_2'}{v_1 = t_2 \to v_1 = t_2'} \qquad \text{[E-EQ2]}$$

Fig. 4: Reduction rules related to KB.

fore omit these rules and focus on the constructs specific to λ_{DL} (see Fig. 4 and 5). The full semantics can be found in the technical report.

A term representing a query can be directly evaluated to a list of objects (E-QUERY). The query reduction rule queries the knowledge system for all $?X$ for which the axiom $\mathcal{K} \models ?X : C$ is true. As λ_{DL} relies on DL-safe queries, only objects actually defined in the signature are allowed. For simplicity's sake, we consider the result to be a list and introduce a σ-operator that takes care of communication between the knowledge system and λ_{DL}. As queries yield sets of objects, this operator essentially works by concatenating every object of the query result into a list. Projections (E-PROJ and E-PROJV) behave similarly. Once the term has been reduced to a object a, the knowledge system is queried for all $?X$ for which $\mathcal{K} \models (a, ?X) : R$. Again, anonymous objects are not considered and the result is converted into a list by the σ-operator.

In case of equivalence, both terms must first be reduced to values (E-EQ1 and E-EQ2). Once both terms are values, equivalence can be computed. Equivalence is distinguished into equivalence for objects (EQ-NOMINAL-TRUE and EQ-NOMINAL-FALSE) and equivalence for primitive values (EQ-PRIM-TRUE and EQ-PRIM-FALSE). λ_{DL} considers two primitive values only equivalent if they

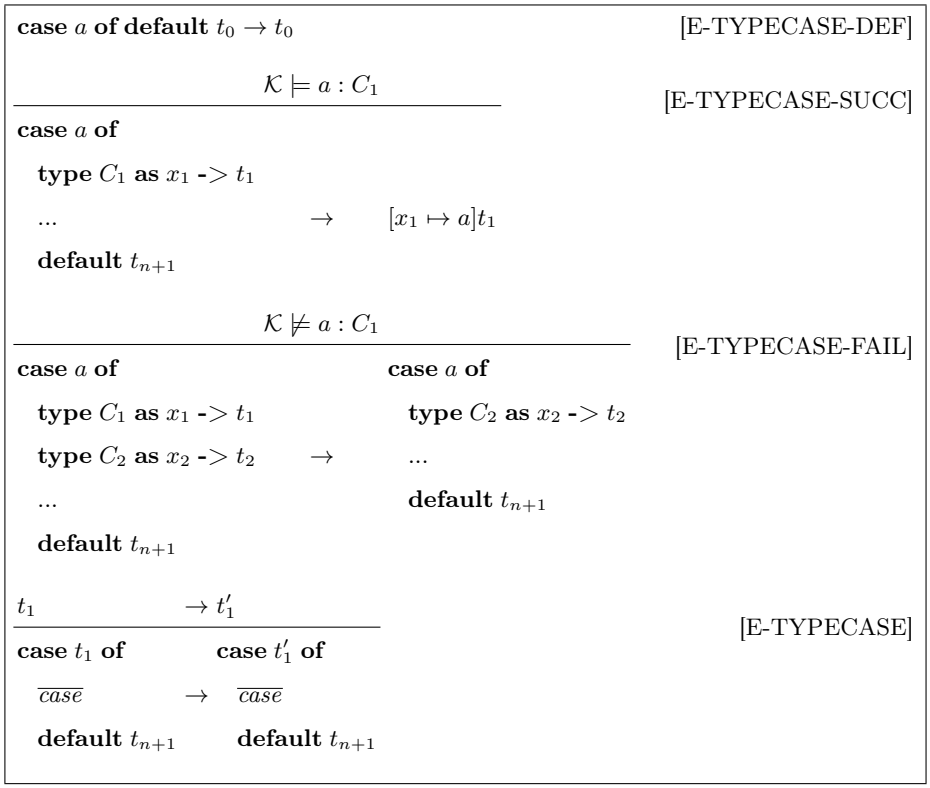

Fig. 5: Reduction rules for typecase terms.

are syntactically equal. In case of objects, the knowledge base is queried. If the knowledge system can unambiguously prove that a is equivalent to b, the two objects are considered to be equal. Due to the open-world querying, objects are considered to be different if the knowledge system is unsure or if it can actually prove that the two objects are not equivalent. We do not consider equivalence for lists or λ-abstractions.

Evaluation of typecase terms (see Fig. 5) is somewhat special. The terms are first reduced to an object (E-TYPECASE). The semantics can then test the object, case by case, until one of them matches (E-TYPECASE-SUCC and E-TYPECASE-FAIL). For each case the knowledge system is queried whether the axiom $\mathcal{K} \models a : C$ is true. Due to the open-world querying, it might happen that the knowledge system cannot compute such a membership. In this case, the typecase is reduced to its default.

5 Type system

The most distinguishing feature of the type system for λ_{DL} is the addition of concept expressions, built according to the rules of Table 2, as types in the language. For constructs unrelated to the knowledge system, this has little impact.

$$lub(\pi_1, \pi_1) \Rightarrow \pi_1 \qquad \text{[LUB-PRIMITIVE]}$$

$$lub(C, D) \Rightarrow C \sqcup D \qquad \text{[LUB-CONCEPT]}$$

$$\frac{lub(S, T) \Rightarrow W}{lub(S \text{ list}, T \text{ list}) \Rightarrow W \text{ list}} \qquad \text{[LUB-LIST]}$$

$$\frac{glb(S_1, T_1) \Rightarrow W_1 \qquad lub(S_2, T_2) \Rightarrow W_2}{lub(S_1 \to S_2, T_1 \to T_2) \Rightarrow W_1 \to W_2} \qquad \text{[LUB-FUNC]}$$

Fig. 6: Least upper bound of types.

Least Upper Bound and Greatest Lower Bound In the typing rules for a few constructs, e.g., for for typing if-then-else expressions, the least upper bound of two types S and T has to be determined; see the designated judgment *lub* in Fig. 6. In case of a least upper bound for primitive types, we simply assume the types to be equal (LUB-PRIMITIVE). For two concepts C and D, a new concept $C \sqcup D$ is constructed (LUB-CONCEPT). For lists of the form S list and T list, we compute the least upper bound of S and T as a new element type for the list. For two functions, $S_1 \to S_2$ and $T_1 \to T_2$, the greatest-lower bound of the argument types S_1 and T_1 ('contra-variance') as well as the least upper bound of S_2 and T_2 ('co-variance') are computed.

The greatest-lower bound of two types S and T is defined analogously. For instance, the greatest lower bound of two concepts C and D is the concept $C \sqcap D$. The complete definition of the designated judgment *glb* can be found in the technical report.

Typing knowledge-base unrelated constructs The typing rules for constructs unrelated to the knowledge base are mainly the standard ones as in common simple (applied) lambda calculi. We only include rules here for constructs that need special attention due to λ_{DL}.

The typing rule for if-then-else expressions needs to be adjusted in a manner similar to type systems with subtyping; see the use of the *lub*-judgment in Fig. 7.

$$\frac{\Gamma \vdash t_1 : \text{bool} \qquad \Gamma \vdash t_2 : S \qquad \Gamma \vdash t_3 : T \qquad lub(S, T) \Rightarrow W}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : W} \qquad \text{[T-IF]}$$

Fig. 7: Typing rules for constructs unrelated to the KB.

Fig. 8 shows the typing rules for list-related forms of terms. The empty list constructor has a type parameter (T-NIL). A cons function (T-CONS) is typed using the least upper bound judgment. The remaining typing rules for functions on lists are the standard ones. For instance, a null function takes a well-typed list and returns a boolean value.

$$\Gamma \vdash \mathbf{nil}[T_1] \ : T_1 \text{ list} \qquad \text{[T-NIL]}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \text{ list} \qquad lub(T_1, T_2) \Rightarrow T_3}{\Gamma \vdash \mathbf{cons}\ t_1\ t_2 : T_3 \text{ list}} \qquad \text{[T-CONS]}$$

$$\frac{\Gamma \vdash t_1 : T \text{ list}}{\Gamma \vdash \mathbf{null}\ t_1 : \text{Bool}} \qquad \text{[T-NULL]}$$

$$\frac{\Gamma \vdash t_1 : T \text{ list}}{\Gamma \vdash \mathbf{head}\ t_1 : T} \qquad \text{[T-HEAD]}$$

$$\frac{\Gamma \vdash t_1 : T \text{ list}}{\Gamma \vdash \mathbf{tail}\ t_1 : T \text{ list}} \qquad \text{[T-TAIL]}$$

Fig. 8: Typing rules for lists

$$\frac{\mathcal{K} \not\models C \equiv \bot}{\Gamma \vdash \mathbf{query}\ C : C \text{ list}} \qquad \text{[T-QUERY]}$$

$$\frac{\Gamma \vdash t_1 : C}{\Gamma \vdash t_1.R : (\exists R^{-}.C) \text{ list}} \qquad \text{[T-PROJ]}$$

$$\frac{\Gamma \vdash t_1 : C \qquad \Gamma \vdash t_2 : D \qquad \mathcal{K} \not\models C \sqcap D \equiv \bot}{\Gamma \vdash t_1 = t_2 : \text{bool}} \qquad \text{[T-EQN]}$$

$$\frac{\Gamma \vdash t_1 : \Pi_1 \qquad \Gamma \vdash t_2 : \Pi_1}{\Gamma \vdash t_1 = t_2 : \text{bool}} \qquad \text{[T-EQP]}$$

$$\Gamma \vdash a : \{\ a\ \} \qquad \text{[T-OBJECT]}$$

Fig. 9: Typing rules for constructs related to the KB.

Typing of knowledge-base related constructs Typing of terms related to the knowledge base is summarized in Fig. 9. Queries (T-QUERY) have a concept C; thus, the result is of type C list. Unsatisfiable queries are rejected by the type system on the grounds of querying the knowledge system on whether C is equivalent to $\bot$. Projections (T-PROJ) require a term of type C and can then be typed by the inverse of the relation used for the projection. This may seem suprising at first sight, but it is actually the most precise type that can be assigned to this term. Range-definitions of roles may be very general (e.g., the range definition for `influencedBy` in the running example). Equivalence requires two well-typed operands with either a non-empty intersection of the

$$\frac{\begin{array}{ll}\Gamma \vdash t_0 : D & \Gamma, x_i : C_i \vdash t_i : T_i \text{ for i=1,..}n \\ \mathcal{K} \not\models C_i \sqsubseteq C_j \text{ for } i < j & \mathcal{K} \not\models C_i \sqcap D \equiv \bot \text{ for i} = 1,..,n \\ \Gamma \vdash t_{n+1} : T_{n+1} & \overline{lub}(T_1, ..., T_{n+1}) \Rightarrow W \end{array}}{\begin{array}{l}\Gamma \vdash \textbf{case } t_0 \textbf{ of} \\ \quad \textbf{type } C_1 \textbf{ as } x_1 \textbf{ -> } t_1 \\ \quad ... \qquad\qquad : \qquad\qquad W \\ \quad \textbf{type } C_n \textbf{ as } x_n \textbf{ -> } t_n \\ \quad \textbf{default } t_{n+1}\end{array}} \text{[T-TYPECASE]}$$

Fig. 10: Typing rule for typecase

associated concepts (T-EQN) or the same primitive type (T-EQ-P); the result is of type bool. Lastly, single objects can be typed using a nominal concept—a concept expression created through enumerating its members.

Consider the typing rule for typecase in Fig. 10. The term to be dispatched on, t_0, is of type D, i.e., a concept. The types of the non-default cases are determined in a context where the variable x_i for each case is bound to the type C_i of the case. The idea is here that t_0 is casted to C_i type-safely and to be accessed as x_i within t_i. The result type of typecase is the least upper bound of the types of all cases including the default case. (We use $\overline{lub}$ as a shortcut for the repeated application of the *lub*-judgment.) There are additional premises to ensure meaningful cases. That is, the intersection between all the C_i and D should not be equivalent to $\bot$, as it would then be impossible for a case to ever match. Also, a case should never be subsumed by a preceding case, as cases are tried sequentially.

Subtyping Subtyping rules are summarized in Fig. 11. We rely on a standard subtyping relation. A term t of type S is also of type T, if $S <: T$ is true (T-SUB). Any type is always a subtype of itself (S-RELF). Subtyping for concepts is handled by the knowledge system. A concept C is a subtype of concept D if the knowledge base can infer that $\mathcal{K} \models C \sqsubseteq D$ (S-CONCEPT). The forwarding of this decision to the knowledge system is important because the knowledge system can take inferred facts into account before making the conclusion. Subtyping for list and function types is reduced to subtyping checks for their associated types. A list S list is a subtype of T list if $S <: T$ is true (S-LIST). Function types are in a subtyping relationship (S-FUNC) if their domains are in a flipped subtyping relationship ('contra-variance') and their co-domains are in a subtyping relationship ('co-variance').

Algorithmic type checking We mention in passing that the type system is more or less directly suited for algorithmic type checking. That is, the rules are completely syntax driven with the routine exception of the rule for adding subtyping

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad [\text{T-SUB}]$$

$$S <: S \quad [\mathsf{S-RELF}]$$

$$\frac{\mathcal{K} \models C \sqsubseteq D}{C <: D} \quad [\mathsf{S-CONCEPT}]$$

$$\frac{S <: T}{S \text{ list} <: T \text{ list}} \quad [\mathsf{S-LIST}]$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad [\mathsf{S-FUNC}]$$

Fig. 11: Subtyping rules.

for terms (T-SUB). There is no problematic rule like transitivity of subtyping, as concept subtyping is taken care of by the knowledge systems.

6 Type soundness

We show the soundness of λ_{DL} by proving that, given the design choices of λ_{DL}, well-typed programs do not get stuck. As with many other languages, there are exceptions to this rule, e.g., down-casting in object-oriented languages, cf. [2]. One may expect that typecases of λ_{DL} may constitute an exception, but the default case avoids this problem. Thus, the only exception concerns lists.

We show that if a program is well-typed, then the only way it can get stuck is by reaching a point where it tries to compute **head nil** or **tail nil**. We proceed in two steps, by showing that a well-typed term is either a value or it can take a step (progress) and by showing that if that term takes a step, the result is also well-typed (preservation). We start by providing some forms about the possible well-typed values (canonical forms) for each type.

Lemma 1 (Canonical Forms Lemma). *Let v be a well-typed value. Then the following observations can be made:*

1. *If v is a value of type C, then v is of the form a.*
2. *If v is a value of type $T_1 \to T_2$, then v is of the form $\lambda(x : S_1).t_2$ with $T_1 <: S_1$.*
3. *If v is a value of type C list, then v is either of the form $(\textbf{\textit{cons}}\ v_1 ...)$ or **nil**.*
4. *If v is a value of type bool, then either v is either true or false.*

Proof. Immediate from the typing relation.

Given Lemma 1, we can show show that a well-typed term is either a value or it can take a step. Given the design decisions of λ_{DL}, this is straightforward.

In particular, we rely on the interpretation of *unknown* facts as false (open-world querying). We also foresee that no case of typecase fits to the runtime value and thus insist on default case. Further, progress for querying relies on the restriction to DL-safe queries, as this leads to finite query results that can be transformed into lists of objects in one step.

Theorem 1 (Progress). *Let t be a well-typed closed term. If t is not a value, then there exists a term t' such that $t \to t'$. If $\Gamma \vdash t : T$, then t is either a value, a term containing the forms **head nil** and **tail nil**, or there is some t' with $t \to t'$.*

Proof. By induction on the derivation of $\Gamma \vdash t : T$. As large parts of the proof are standard cases, we focus on the part specific to our language. The remaining standard cases can be found in the technical report.

(T-QUERY) $t =$ **query** C, $\Gamma \vdash t : C$ list. Immediate since rule E-QUERY applies (see Fig. 4).

(T-PROJ) $t = t_1.R$, $\Gamma \vdash t_1 : C$, $\Gamma \vdash t : (\exists R^-.C)$. By hypothesis, either t_1 is a value or it can take a step. If it can take a step, rule E-PROJ applies. If its a value, then by Lemma 1 $t_1 = a$, therefore rule E-PROJV applies.

(T-TYPECASE)
$t =$ **case** t_0 **of**
$\overline{case}$
default t_{n+1}
$\Gamma \vdash t_0 : D$, $\Gamma \vdash t : W$
By hypothesis, t_0 is either a value or it can take a step. If it can take a step, rule E-TYPECASE applies. If its a value, by Lemma 1, $t_0 = a$. If $\overline{case}$ is non-empty, either rules E-TYPECASE-SUCC or E-TYPECASE-FAIL apply. Otherwise, rule E-TYPECASE-DEF applies (see Fig. 5).

(T-EQN) $t_1 = t_2$, $\Gamma \vdash t_1 : C$, $\Gamma \vdash t_2 : D$. Either t_1 and t_2 are values or they can take a step. If they can take a step, rules E-EQ1 and E-EQ2 apply. If both are values, by Lemma 1, $t_1 = a$, $t_2 = b$. Therefore, either rule EQ-NOMINAL-TRUE or EQ-NOMINAL-FALSE applies.

(T-EQP) $t_1 = t_2$, $\Gamma \vdash t_1 : \Pi_1$, $\Gamma \vdash t_2 : \Pi_1$. Either t_1 and t_2 are values or they can take a step. If they can take a step, rules E-EQ1 and E-EQ2 apply. If both are values, them they are either syntactically equal or not. Therefore either EQ-PRIM-TRUE or EQ-PRIM-FALSE applies.

(T-OBJ) Immediate, since $t = a$ is a value.

For proving preservation, two additional Lemmas are required. One, that substitution preserves the type and two, that the least upper bound judgment computes a type that is really a supertype of its two input types.

Lemma 2 (Substitution). *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.*

Proof. Substitution in λ_{DL} does not differ from standard approaches, e.g., as described in [32]. Therefore, the proof is omitted.

Lemma 3 (Least Upper Bound). *Let S, T and W be types. If $lub(S,T) \Rightarrow W$, then $S <: W$ and $T <: W$.*

Proof. Four cases must be considered: S and T are either primitives, concepts, lists or functions.

Primitives: Result is immediate since $S = T = W$. By subtyping rule S-REFL, $S <: W$ and $T <: W$ holds.
Concepts: $S = C$, $T = D$, $W = C \sqcup D$. Since $\mathcal{K} \models C \sqsubseteq C \sqcup D$ and $\mathcal{K} \models D \sqsubseteq C \sqcup D$, $S <: W$ and $T <: W$ hold via subtyping rule S-CONCEPT.
Lists Immediate through the induction hypothesis and subtyping rules for lists.
Functions Immediate through induction hypothesis and subtyping rules for functions.

Given these lemmas, we can now continue to show that if a term takes a step by the evaluation rules, its type is preserved. A problematic case for preservation are projections. Existing approaches have problems assigning the most specific type to such terms (e.g., projections involving `influencedBy`). They resolve this by using assigning rather general types, which is ultimately not very helpful. The usage of concept expressions as types on the other hand allows for assigning the most specific type.

Theorem 2 (Preservation). *Let t be a term and T a type. If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.*

Proof. By induction on the derivation of $\Gamma \vdash t : T$. Again, we examine only the specific cases while the full proof can be found in the technical report.

(T-QUERY) t = **query** C, $\Gamma \vdash t : C$ list. By applying rule E-QUERY, t' = **cons** a_1 .... However, for each a, it is known that $\mathcal{K} \models a : C$, therefore $\{ a \} <: C$ holds for each a and $\{ a_1 \} \sqcup ... <: C$ list.
(T-PROJ) $t = t_1.R$, $\Gamma \vdash t_1 : C$, $\Gamma \vdash t : (\exists R^{-}.C)$. There are two rules by which t' can be computed: E-PROJ and E-PROJV:
- **(1)** $t' = t'_1.R$. By induction hypothesis, typing is preserved for t_1. Therefore, by T-PROJ, $t' : (\exists R^{-}.C)$ list.
- **(2)** $t' = \sigma(\{?X \mid ?X \in \mathcal{O} \wedge \mathcal{K} \models (a, ?X) : R\})$ = **cons** b_1 .... For a, it is known that $\mathcal{K} \models a : C$ and for each b is known that $\mathcal{K} \models (a,b) : R$ holds. Therefore, $\mathcal{K} \models b : (\exists R^{-}.C)$ must hold for each b. Thereby, $\{ b_1 \} \sqcup ... <: (\exists R^{-}.C)$ and by S-LIST $(\{ b_1 \} \sqcup ...)$ list $<: (\exists R^{-}.C)$ list

(T-TYPECASE)
t = **case** t_0 **of**
  type C_1 **as** x_1 **->** t_1
  ...
  type C_n **as** x_n **->** t_n
  default t_{n+1}
$\Gamma \vdash t_0 : D$, $\Gamma \vdash t_1 : T_1$, ..., $\Gamma \vdash t_n : T_n$, $\Gamma \vdash t_{n+1} : T_{n+1}$,
$\overline{lub}(T_1, ..., T_{n+1}) \Rightarrow W$, $\Gamma \vdash t : W$

There are four rules by which t' can be computed: E-TYPECASE, E-TYPECASE-SUCC, E-TYPECASE-FAIL and E-TYPECASE-DEF.

(1)

$t' =$ **case** t'_0 **of**
type C_1 **as** x_1 **->** t_1
...
type C_n **as** x_n **->** t_n
default t_{n+1}

By induction hypothesis, $t_1 \rightarrow t'_1$ preserves the type. Therefore, by T-TYPECASE, $t' : W$.

(2) $t' = [x_1 \mapsto a]t_1,\ \Gamma \vdash t_1 : T_1$. By Lemma 2, substitution does not change the type of t_1. By Lemma 3, $T_1 <: W$ and therefore by rule T-SUB $t_1 : W$.

(3)

$t' =$ **case** a **of**
type C_2 **as**x_2 **->** t_2
...
type C_n **as**x_n **->** t_n
default t_{n+1}

$\Gamma \vdash t_2 : T_1,\ ...,\ \Gamma \vdash t_n : T_n,\ \Gamma \vdash t_{n+1} : T_{n+1},$
$\overline{lub}(T_2, ..., T_{n+1}) \Rightarrow W',\ \Gamma \vdash t' : W'$
The removal of the first case causes T-TYPECASE to assign type $t' : W'$. Removal of T_1 makes W' more specific then W, but $W' <: W$ holds. Therefore by, T-SUB $t' : W$.

(4) $t' = t_{n+1}\ \Gamma \vdash t_{n+1} : T_{n+1}$. By Lemma 3, $T_{n+1} <: W$, therefore by T-SUB $t' : W$.

(T-EQN) $t_1 = t_2,\ \Gamma \vdash t_1 : C,\ \Gamma \vdash t_2 : D,\ \Gamma \vdash t :$ bool. There are 6 different rules by which t' can be computed: E-NOMINAL-TRUE, E-NOMINAL-FALSE, E-PRIM-TRUE, E-PRIM-FALSE, E-EQ1 and E-EQ2.

(1) $t' =$ true. Immediate by rule T-TRUE.
(2) $t' =$ false. Immediate by rule T-FALSE.
(3) $t' =$ true. Immediate by rule T-TRUE.
(4) $t' =$ false. Immediate by rule T-FALSE.
(5) $t' = t'_1 = t_2$. By induction hypothesis, $t_1 \rightarrow t'_1$. preserves the type. Therefore, by rule T-EQN, $t' :$ bool.
(6) $t' = v_1 = t'_2$. By induction hypothesis, $t_2 \rightarrow t'_2$. preserves the type. Therefore, by rule T-EQN, $t' :$ bool.

(T-EQP) $t_1 = t_2,\ \Gamma \vdash t_1 : \Pi_1,\ \Gamma \vdash t_2 : \Pi_1$. Same as T-EQN.

(T-OBJ) Vacuously fulfilled since $t = a$ is a value.

As a direct consequence of Theorem 1 and 2, a well-typed closed term does not get stuck during evaluation. The only exception concerns the handling of lists, which can get stuck if **head** or **tail** is applied to an empty list. Empty lists might be produced by queries with empty result sets.

To a certain degree, type safety holds even when the knowledge system is evolving. Additional axioms are unproblematic, as DL is a monotonous logic—they do not invalidate existing inferences. Deletion and modification of the actual

data (A-Box) is unproblematic unless the program contains statements explicitly referencing the objects under modification. Of course, type safety cannot be guaranteed if schematic parts (T-Box) of the knowledge system are altered.

7 Related work

λ_{DL} is generally related to the integration of data models into programming languages. We consider four different ways of integrating a data model: by using generic representations, by mappings into the target language, through a preprocessing step before compilation, or through language extensions or custom languages.

Generic representations Generic representations offer easy integration into programming languages and have the advantage that they can represent anything the data can model, e.g., generic representations (such as DOM[9]) for XML [37]. This approach has also been applied to semantic data. Representations can vary, however the most popular ones include axiom-based approaches (e.g., [19]), graph-based ones (e.g., [11]) or statement-based ones (e.g., RDF4J[10]). All these approaches are error-prone in so far that code on the generic representations is not type-checked in terms of the involved conceptualizations.

Mappings Mapping approaches on the other hand use schematic information of the data model to create types in the target language. Type checking can be used thus to check the valid use of the derived types in programs. This approach has been successfully used for SQL [28], XML [37, 25, 3], and more generally [24, 35]. Naturally, mappings have been studied in a semantic data context, too. The focus is on transforming conceptual statements into types of the programming language. Frameworks include ActiveRDF [29], Alibaba[11], Owl2Java [21], Jastor[12], RDFReactor[13], OntologyBeanGenerator[14], Àgogo [31] and LITEQ [26]. However, mapping approaches are problematic for semantic data. For one, the transformation of statements such as those shown in line 1 of Listing 3 is not trivial due to the mixture of nominal and structural typing. Extremely general information on domains and ranges of roles such as `influencedBy` occurs frequently. The question arises what types support such a role. Frameworks usually resolve the situation by assigning the role to every type they create. In terms of the range of the role, they usually assign the most general available type and leave it to the developer to cast the values to their correct types—this is an error-prone approach. Lastly, all mapping frameworks have problems with the large number of potential types in semantic data sources.

[9] `https://www.w3.org/DOM/`
[10] `http://rdf4j.org/`
[11] `https://bitbucket.org/openrdf/alibaba`
[12] `http://jastor.sourceforge.net/`
[13] `http://semanticweb.org/wiki/RDFReactor`
[14] `http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator`

Precompilation A separate precompilation step, where the source code is statically analyzed and then transformed is another way to solve the problem of integrating data models into programming languages. Especially queries embedded in programming languages can be verified in this manner. This approach has been applied to, for example, SQL queries [38]. The approach has been applied to semantic data in a limited manner [16]—for queries that can be typed with primitive types such as integer.

Language extensions and custom type systems The most powerful approaches extend existing languages or create new type systems to accommodate the specific requirements of the data model. Examples for such extensions are concerned with relationships between objects [7] and easy data access to relational and XML data [8]. Another example concerns programming language support for the XML data model specifically in terms of regular expression type, as in the languages CDuce [5] and XDuce [20]. While semantic data can be seen as somewhat semi-structured and is often serialized in XML, the XML-focused approaches do not address the logics-based challenges regarding semantic data. Another related approach is the idea of functional logic programming [17]. However, λ_{DL} emphasizes type-checking on data axiomatized in logic over the integration of the logic programming paradigm into a language.

The typecase construct of λ_{DL} is inspired by other other forms of typecase such as those in the context of dynamic typing [1], intensional polymorphism [13], and generic functional programming [23]. None of these forms are concerned with semantic data or description logics.

Language extensions and custom approaches have also been implemented for semantic data. In one approach [30], the C# compiler was extended to allow for OWL and XSD types in C#. The main technical difference to λ_{DL} is that λ_{DL} makes use of the knowledge system for typing and subtyping judgments. λ_{DL} can therefore make use of inferred data and has a strong typing mechanism. There is also work on custom languages that use static type-checking for querying and light scripting in order to avoid runtime errors [12, 14]. However, the types are again limited in these cases, as they only consider explicitly given statements. Furthermore, they face the same difficulties as mapping approaches when it comes to schema information—they rely on domain and range specifications for predicates to assign types.

8 Summary and future work

In this paper, we have motivated, introduced, and studied λ_{DL}: a typed λ-calculus for semantic data that is built around concept expressions as types as well as queries. We have shown that by using conceptualizations as they are defined in the knowledge system itself, type safety can be achieved. This helps in writing less error-prone programs, even when facing knowledge systems that evolve or lack role definitions. There are these directions for future work.

Fixed-Domain Reasoning While description logics usually employs an open-world assumption that allows for the modeling of unknown facts, in some cases, a closed-world assumption might be preferable. The semantics as presented in Section 2 could be replaced by a fixed-domain semantics, e.g., as described by [15]. Future work aims to examine how expressiveness and the type safety property of λ_{DL} are affected by such a semantics.

Contracts Type-safety has been achieved in λ_{DL} by some rather harsh restrictions, e.g., by requiring a default case in typecase constructs. Additionally, it is still possible to get stuck, e.g., when taking the head of an empty query response. A possible improvement could be the introduction of contracts, as they are applicable to functional programming [18]. Contracts have been applied already to semantic data [22] while focusing on constraints regarding existence and cardinality. We envision a form of contracts that also covers anonymous objects.

$\lambda_{\mathbf{DL}}$ *and System F* The presented calculus essentially combines 'simple' types and concepts with subtyping. Parametric polymorphism à la *System F* would be needed to arrive at a sufficiently expressive language for purposes of actual programming. Further, the subtyping aspects of $\lambda_{\mathbf{DL}}$ may also call for a comprehensive integration of description logics and polymorphism with subtyping à la *System* $F_{<:}$ [34]. Such an integration is not straightforward.

Modification of the semantic data It is clearly desirable that semantic data can also be modified. A corresponding extension of λ_{DL} is non-trivial because of the aspect that facts are inferred by the knowledge system. Consider the facts about music artists in Listing 3 and let us assume that we want to remove the (implicit) fact that the `beatles` have made a song. The fact cannot be removed directly. Instead, either the fact that the `beatles` are of type `MusicArtist` or the fact that they have been played by `coolFm` must be removed. In order to integrate modification of knowledge systems into λ_{DL}, the theory of knowledge revision based on the AGM theory [33] can be considered and integrated into the language.

Enhanced querying Queries, as they are currently implemented, are limited in their expressive power. A simple extension are queries for roles, such as `influencedBy` that result in sets of pairs. Typing such queries is possible via the addition of tuples to λ_{DL}. The addition of query languages closer to the power of SQL is also possible. The biggest challenge in this regard is query subsumption. When such queries are typed in the programming language, subsumption checks are necessary to determine whether a function can be applied to query results. Therefore only query languages with decidable query subsumption are to be considered, e.g., [9].

References

1. M. Abadi, L. Cardelli, B. C. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *J. Funct. Program.*, 5(1):111–130, 1995.

2. A.Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
3. S. Alagic and P. A. Bernstein. Mapping XSD to OO schemas. In *Proc. of Object Databases*, volume 5936 of *LNCS*, pages 149–166. Springer, 2009.
4. D. Beneventano, S. Bergamaschi, and C. Sartori. Description logics for semantic query optimization in object-oriented database systems. *Trans. Database Syst.*, 28(1):1–50, 2003.
5. V. Benzaken, G. Castagna, and A. Frisch. Cduce: an xml-centric general-purpose language. *SIGPLAN Notices*, 38(9):51–63, 2003.
6. D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artif. Intell.*, 168(1-2):70–118, 2005.
7. G. Bierman and A. Wren. First-Class Relationships in an Object-Oriented Language. In *Proc. of ECOOP*, pages 262–286. Springer, 2005.
8. G. M. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Comega. In *Proc. of ECOOP*, LNCS, pages 287–311. Springer, 2005.
9. P. Bourhis, M. Krötzsch, and S. Rudolph. Reasonable highly expressive query languages. In *Proc. of International Joint Conference on Artificial Intelligence*, pages 2826–2832. AAAI Press, 2015.
10. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, A. Poggi, and R. Rosati. Ontology-based database access. In *Proc. of Advanced Database Systems*, pages 324–331, 2007.
11. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proc. of WWW - Alternate Track Papers & Posters*, pages 74–83. ACM, 2004.
12. G. Ciobanu, R. Horne, and V. Sassone. Descriptive types for linked data resources. In *Perspectives of System Informatics*, LNCS, pages 1–25. Springer, 2014.
13. K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. *J. Funct. Program.*, 12(6):567–600, 2002.
14. Ciobanu G, R. Horne, and V. Sassone. Minimal type inference for linked data consumers. *J. Log. Algebr. Meth. Program.*, 84(4):485–504, 2015.
15. S. A. Gaggl, S. Rudolph, and L. Schweizer. Fixed-domain reasoning for description logics. In *ECAI 2016*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, pages 819–827. IOS Press, 2016.
16. S. Groppe, J. Neumann, and V. Linnemann. SWOBE - embedding the semantic web languages rdf, SPARQL and SPARUL into java for guaranteeing type safety, for checking the satisfiability of queries and for the determination of query result types. In *Proc. of Symposium on Applied Computing*, pages 1239–1246. ACM, 2009.
17. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
18. R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *Proc. of Functional and Logic Programming*, LNCS, pages 208–225. Springer, 2006.
19. M. Horridge and S. Bechhofer. The OWL API: A java API for OWL ontologies. *Semantic Web*, 2(1):11–21, 2011.
20. H. Hosoya and B. C. Pierce. Xduce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
21. A. Kalyanpur, D. J. Pastor, S. Battle, and J. A. Padget. Automatic mapping of OWL ontologies into java. In *Proc. of International Conference on Software Engineering & Knowledge Engineering*, pages 98–103, 2004.

22. Petr Kremen and Zdenek Kouba. Ontology-driven information system design. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 42(3):334–344, 2012.
23. R. Lämmel and S. L. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proc. of TLDI'03*, pages 26–37. ACM, 2003.
24. R. Lämmel and E. Meijer. Mappings make data processing go 'round. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE)*, LNCS, pages 169–218. Springer, 2005.
25. R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch - (changing lead into gold). In *Datatype-Generic Programming - International Spring School, SSDGP*, LNCS, pages 285–367. Springer, 2006.
26. M. Leinberger, S. Scheglmann, R. Lämmel, S. Staab, M. Thimm, and E. Viegas. Semantic web application development with LITEQ. In *Proc. of ISWC*, LNCS, pages 212–227. Springer, 2014.
27. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. *J. Web Sem.*, 3(1):41–60, 2005.
28. E. J. O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proc. of International Conference on Management of Data*, pages 1351–1356. ACM, 2008.
29. E. Oren, B. Heitmann, and S. Decker. Activerdf: Embedding semantic web data into object-oriented languages. *Web Semant.*, 6(3):191–202, 2008.
30. A. Paar and D. Vrandecic. Zhi# - OWL aware compilation. In G. Antoniou, M. Grobelnik, E. P. Bontas Simperl, B. Parsia, D. Plexousakis, P. De Leenheer, and J. Z. Pan, editors, *Proc. of Extended Semantic Web Conference*, LNCS, pages 315–329. Springer, 2011.
31. F. S. Parreiras, C. Saathoff, T. Walter, T. Franz, and S. Staab. 'a gogo: Automatic Generation of Ontology APIs. In *ICSC2009*. IEEE, 2009.
32. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
33. G. Qi, W. Liu, and D. A. Bell. Knowledge base revision in description logics. In *Proc. of Logics in Artificial Intelligence*, pages 386–398. Springer, 2006.
34. J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.
35. D. Syme, K. Battocchi, K. Takeda, D. Malayeri, and T. Petricek. Themes in information-rich functional programming for internet-scale data sources. In *Proc. of DDFP*, pages 1–4. ACM, 2013.
36. D. Vrandecic and M. Krötzsch. Wikidata: a free collaborative knowledgebase. *Commun. ACM*, 57(10):78–85, 2014.
37. M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proc. of International Conference on Functional Programming*, pages 148–159. ACM, 1999.
38. G. Wassermann, C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16(4), 2007.