\( \lambda_{DL} \): SYNTAX AND SEMANTICS
(Preliminary Report)

MARTIN LEINBERGER
University of Koblenz-Landau
Institute for Web Science and Technologies
mleinberger@uni-koblenz.de

RALF LÄMMEL
University of Koblenz-Landau
The Software Languages Team
laemmel@uni-koblenz.de

STEFFEN STAAB
University of Koblenz-Landau
Institute for Web Science and Technologies
staab@uni-koblenz.de
& Web and Internet Science Research Group
University of Southampton
s.r.staab@soton.ac.uk
Abstract
Semantic data fuels many different applications, but is still lacking proper integration into programming languages. Untyped access is error-prone while mapping approaches cannot fully capture the conceptualization of semantic data. In this paper, we present \( \lambda_{DL} \), a \( \lambda \)-calculus with a modified type system to provide type-safe integration of semantic data. This is achieved by the integration of description logics into the \( \lambda \)-calculus for typing and data access. It is centered around several key design principles. Among these are (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 in \( \lambda_{DL} \). The paper motivates the use of a modified type system for semantic data and it provides the theoretic foundation for the integration of description logics as well as the core formal specifications of \( \lambda_{DL} \) including a proof of type safety.

Keywords Semantic data, Type systems, Typecase

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 (Vrandecic and Krötzsch 2014) 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 2,515 relations. The ontology defined by Schema.org 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 providing 11 billion triples. Semantic data has also interlinked large, varied data sources, such as provided by Fokus 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 their programmatic access and their integration into programming languages.

As the running example, consider semantic data about music artists formalized in the description logic \( ALCOH(D) \). Listing 1 shows everyone for which a recorded relation, that points to an entity of type Song, exists is considered to be a MusicArtist (Line 2). beats 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. beats : 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 integrating semantic data into 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 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. (Horridge and Bechhofer 2011)). 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 (Kalyanpur et al. 2004) 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.

Contribution of the paper We therefore propose a third, a novel approach: A type system designed for semantic data (3). In this paper, we present \( \lambda_{DL} \), a functional language for working with knowledge systems. \( \lambda_{DL} \) uses concept expressions such as MusicArtist and 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 query mechanism based on description logics. The querying mechanism allows for checking of satisfiability of queries as well as typing the query results in the programming language. As a result, \( \lambda_{DL} \) 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 \( \exists_{\text{influencedBy}}.\top \) as input. In other words, the functions accepts entities for which an influencedBy-relationship exists, disregarding of the type of entity that relation points to. 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 \( \lambda_{DL} \) is precise enough to detect such errors (see Listing 2).

\[
\text{let } f = \lambda(x:\exists_{\text{influencedBy}}.\top) . x.\text{influencedBy in } f(\text{head (query MusicArtist)})
\]

Listing 2: Rejected code — music artist is not a subtype of \( \exists_{\text{influencedBy}}.\top \).

Road-map of the paper The remaining paper is organized as follows. In Section 2, we introduce description logics as the theoretical foundation of semantic data. In Section 3 we illustrate \( \lambda_{DL} \) 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. Additionally, we shortly describe the prototypical implementation of λ_{DL} in the appendix. Further information about λ_{DL} is available at http://west.uni-koblenz.de/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). 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 [Berardi et al. 2003], semantic query optimization on object-oriented database systems [Beneventano et al. 2003], or improving database access through abstraction (Calvanese et al. 2007).

Syntax and Semantics

Semantic data, also called a knowledge base, comprises a set of description logics axioms that are composed using a signature Σ(K) and a set of logical and concept operators and comparisons. A signature Σ(K) of a knowledge base K is a triple Σ(K) = (A, Q, O) where A is a set of concept names, Q is a set of role names, and O is a set of object names. DL uses Tarskian-style, interpretation-based semantics. An interpretation I is a pair consisting of a non-empty universe Δ^I and an interpretation function : Δ^I → Δ^I that maps each object a, b ∈ O to an element of the universe. Furthermore, it assigns each concept name A ∈ A a set A^I ⊆ Δ^I and each role name Q ∈ Q to a binary relation Q^I ⊆ Δ^I × Δ^I. In our running example, the signature of Listing 1 contains the concept MusicArtist and Song, the roles recorded and influencedBy as well as the objects beatles, hendrix, and machineGun. An interpretation 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 MusicArtist^I = {hendrix, beatles}, and the interpretation of Song might be Song^I = {machineGun}. The interpretation of the recorded role might be recorded^I = {(hendrix, machineGun)} and influencedBy^I = {(hendrix, beatles)}.

Given these element names, complex expressions such as shown in Listing 1 can be built. For the course of the paper, the specific description logics dialect needed to cover all necessary constructs is ALCIO, consisting of the most commonly used Atributive 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 a atomic role or the inverse of a role expression.

<table>
<thead>
<tr>
<th>Role Expression</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Role</td>
<td>Q</td>
<td>Q^I ⊆ Δ^I × Δ^I</td>
</tr>
<tr>
<td>Inverse</td>
<td>R^-</td>
<td>{(a, b) ∈ Δ^I × Δ^I</td>
</tr>
</tbody>
</table>

Table 1: Role expressions and associated semantics.

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, ⊤, ⊥ or the negation of a concept. 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 recordedSong 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} ∪ {beatles}. Table 2 summarizes the syntax and semantics of concept expressions.

<table>
<thead>
<tr>
<th>Concept Expression</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal concept</td>
<td>{a}</td>
<td>{a^I}</td>
</tr>
<tr>
<td>Atomic concept</td>
<td>A</td>
<td>A^I ⊆ Δ^I</td>
</tr>
<tr>
<td>Top</td>
<td>⊤</td>
<td>Δ^I</td>
</tr>
<tr>
<td>Bottom</td>
<td>⊥</td>
<td>∅</td>
</tr>
<tr>
<td>Negation</td>
<td>¬C</td>
<td>Δ^I \ C</td>
</tr>
<tr>
<td>Intersection</td>
<td>C ∩ D</td>
<td>C^I ∩ D^I</td>
</tr>
<tr>
<td>Union</td>
<td>C ∪ D</td>
<td>C^I ∪ D^I</td>
</tr>
<tr>
<td>Existential Quantification</td>
<td>∃R.C</td>
<td>{a^I ∈ Δ^I</td>
</tr>
<tr>
<td>Universal Quantification</td>
<td>∀R.C</td>
<td>{a^I ∈ Δ^I</td>
</tr>
</tbody>
</table>

Table 2: Concept expressions and associated semantics.

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 data types is common. We therefore also include XSD data types wherever it is appropriate. As an example, consider the concept expression ⋌artistName.xsd:string describing the set of all objects having an artist name that is a string. However, 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 K is a pair K = (T, A) consisting of the set of terminological axioms T, the conceptualization of the data and the set of assertional axioms 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 allow for the definition of domains and ranges of roles used in the ontology. As shown in Figure 4 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 2). We assume everyone who has recorded a song to be a music artist, but not all music artists have

4 https://www.w3.org/TR/xmlschema-2/}
5 RDF Schema, one of the weakest forms of terminological axioms.
Table 3: Terminological and assertional axioms.

<table>
<thead>
<tr>
<th>Name</th>
<th>Syntax</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept inclusion</td>
<td>$C \subseteq D$</td>
<td>$C^\exists \subseteq D^\exists$</td>
</tr>
<tr>
<td>Concept equality</td>
<td>$C = D$</td>
<td>$C^\exists = D^\exists$</td>
</tr>
<tr>
<td>Concept assertion</td>
<td>$a : C$</td>
<td>$a^\exists \in C^\exists$</td>
</tr>
<tr>
<td>Role equivalence</td>
<td>$(a, b) : R$</td>
<td>$(a^\exists, b^\exists) \in R^\exists$</td>
</tr>
</tbody>
</table>

Figure 1: Syntactical abbreviations for DL.

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 $I$ the set of all interpretations. $F$ is inferred, written $\mathcal{K} \models F$, if $\forall I \in \mathcal{I} : I \models F$.

An example of this is the axiom $\text{hendrix:MusicArtist}$. hendrix has recorded a song and must therefore be element of $\exists \text{recorded.Song}$. As $\exists \text{recorded.Song} \subseteq \text{MusicArtist}$ 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 an 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 for which $a : C$ is true is then $\{ ?X | \mathcal{K} \models ?X : C \}$.

As an example, consider the query $\mathcal{K} \models ?X : \text{MusicArtist}$. 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).

Listing 4: Naturally large knowledge system.

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 $I$ the set of all interpretations. $F$ is inferred, written $\mathcal{K} \models F$, if $\forall I \in \mathcal{I} : I \models F$.

An example of this is the axiom $\text{hendrix:MusicArtist}$. hendrix has recorded a song and must therefore be element of $\exists \text{recorded.Song}$. As $\exists \text{recorded.Song} \subseteq \text{MusicArtist}$ 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 an 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 for which $a : C$ is true is then $\{ ?X | \mathcal{K} \models ?X : C \}$.

As an example, consider the query $\mathcal{K} \models ?X : \text{MusicArtist}$, 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).

Listing 4: Naturally large knowledge system.

If someone is a person, then he must have a father which is an anonymous object and a person himself, again implying that this anonymous object has a father. A query $\mathcal{K} \models ?X : \text{Person}$ therefore yields an infinite number of unifications. We therefore use a simple form of so called DL-safe queries (cf. (Motik et al. 2009)), which restrict unifications to objects defined in the signature (see Def. 3).

Definition 3 (DL-safe queries). Let $\mathcal{K}$ be a knowledge base, $\text{Sig}(\mathcal{K}) = (\mathcal{A}, \mathcal{Q}, \mathcal{O})$ 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 $\{ ?X | \mathcal{K} \models ?X : C \land ?X \in \mathcal{O} \}$.

In this 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. \(\lambda_{DL}\) in a nutshell

Developing applications for knowledge systems, as introduced in the previous section, is difficult and error-prone. \(\lambda_{DL}\) 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 Due to the large number of potential concepts, it is infeasible to compute subtype relations beforehand. Therefore, the facts about subsupervise relationships between concepts must be added to the system during the type checking process by forwarding these checks to the knowledge system.

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. Queries for which no possible A-Box instance can produce a result are therefore detected and rejected. Second, usage of queries must be type safe, meaning that the query result must be properly typed. Queries always return lists in \(\lambda_{DL}\).

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, \(\lambda_{DL}\) relies on a simplified form of DL-safe queries. Queries are enforced to be finite by only allowing unifications with known objects. However, this may also lead to empty result sets for queries.

Open-world querying When looking at inferencing, axioms may be true, false or unknown. For simplicity, \(\lambda_{DL}\) 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 \(\neg C\).

\[
\text{letrec } x : T_1 = t_1 \text{ in } t_2 = \text{let } x = \text{fix } (\lambda x : T_1, t_1) \text{ in } t_2
\]

Figure 3: Syntactical abbreviations of \(\lambda^{DL}\).

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 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 \(\lambda_{DL}\). The necessary list of music artists that have recorded at least one song can be queried using MusicArtist \(\sqsubseteq \exists \text{recorded}.\text{Song}\) (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 \(\sqsubseteq \exists \text{recorded}.\text{Song}\)) to the evaluation result.

\[
\text{query MusicArtist } \sqsubseteq \exists \text{recorded}.\text{Song}
\]

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

Mapping a member of this list to his recorded songs can be done using role projections. The input type for such a mapping function is \(\exists \text{recorded}.\text{Song}\) which is a super type of MusicArtist \(\sqsubseteq \exists \text{recorded}.\text{Song}\). 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.

\[
\text{let getRecordings } = \lambda (a: \exists \text{recorded}.\text{Song}). a.\text{recorded}
\]

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 \(\lambda_{DL}\) 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.

\[
\text{let getArtistName } = \lambda (a: \exists \text{artistName}.\text{xsd:string}). \text{head} (a.\text{artistName})
\]

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. \(\lambda_{DL}\) provides a type dispatch for this use case. Listing 8 shows the code for this function. In case that the argument of the function is of type influencedBy. \(\top\), the actual mapping function is applied to the value — otherwise, an empty list is returned.

\[
\text{let getArtistInfluences } = \lambda (\text{artist: MusicArtist}). \text{case artist of type } \exists \text{influencedBy}.\top \text{ as } x \rightarrow \text{getInfluences } x \text{ default nil}
\]

Listing 8: Casting a music artist to influencedBy. \(\top\).
The function computing the actual influences can use a projection and then a mapping to a human-readable name. However, this human-readable name is problematic. Due to the weak schematic restrictions of the influencedBy role the code must 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 should be ignored. Listing 9 shows the complete code for the function.

```
1 let getInfluences = λ(obj:∃influencedBy.T).
2   let toName = λ(x:∃influencedBy.T).
3     case x of
4       type MusicArtist as y → getName y
5       "no influence known"
6     in letrec
7       getNames:(∃influencedBy.T list → string list) =
8         λ(source:∃influencedBy.T list).
9         if (null source) then nil[primitive value]
10        else cons (toName (head source))
11          (getNames (tail source))
12    in
13       getNames obj.influencedBy
```

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

### 4. Core language

**Syntax** Our core language (Figure 3) 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 statements. Constructs for lists are included in the language: cons, nil, null, head and tail. Based on these, complex operations such as map, fold and filter can be built. For simplicity, we did not include these in our syntax. 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. Casting is done via a type-dispatch construct that contains an arbitrary number of cases plus a default case. We use an overbar notation to represent sequences of syntactical elements. Concretely, \( π \) 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

```
query C → σ(\{?X | ?X ⊓ K |= ?X : C\}) = (cons a_1 ...)
   a.R → σ(\{?X | ?X ⊓ K |= (a, ?X) : R\})
   t_1 → t'_1
   t_1.R → t'_1.R
   K |= a ≡ b
   a = b → true
   K \not|= a ≡ b
   a = b → false
   p_1 = p_1 → true
   p_1 ≠ p_2 → false
   t_1 → t'_1
   t_1 = t_2 → t'_1 = t_2
   t_2 → t'_2
   v_1 = t_2 → v_1 = t'_2
```

Figure 4: Reduction rules related to KB.
In case of equivalence, both terms must first be reduced to val-
uation for primitive values (EQ-PRIM-TRUE and EQ-PRI-
minal objects (EQ-NOMINAL-TRUE and EQ-NOMINAL-FALSE) and
can be computed. Equivalence is distinguished into equiva-
ence for objects, the knowledge base

\[
C = a : C_1
\]

\[
K \not\models a : C_1
\]

\[
C_1 \text{ as } x_1 \to t_1
\]

\[
\ldots \quad \rightarrow \quad [x_1 \mapsto a]t_1
\]

\[
\text{type } C_1 \text{ as } x_1 \to t_1
\]

\[
\text{default } t_{n+1}
\]

\[
K \not\models a : C_1
\]

\[
C_2 \text{ as } x_2 \to t_2
\]

\[
\ldots \quad \rightarrow \quad [x_2 \mapsto a]t_1
\]

\[
\text{type } C_2 \text{ as } x_2 \to t_2
\]

\[
\text{default } t_{n+1}
\]

\[
T_1 \quad \rightarrow \quad T_1'
\]

\[
\text{case } t_1 \text{ of}
\]

\[
\text{case } t_1' \text{ of}
\]

\[
\text{default } t_{n+1}
\]

\[
\text{default } t_{n+1}
\]

Figure 5: Reduction rules for type case terms.

operator in our representation. Values \((v)\) include objects defined in
the knowledge base, nil and cons to represent lists, \(\lambda\)-abstractions
and primitive values. \(\lambda\)-abstractions indicate the type of their var-
iable. 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 \(\lambda\)-abstractions. To sim-
plify recursion, we also define a letrec as an abbreviation of the
fixpoint operator (see Figure 4).

**Semantics**

The operational semantics is defined using a reduc-
tion relation, which extends the standard approaches. Reduction
of lists and terms not related to the knowledge bears no significant dif-
ference from rules as, e.g., defined in (Pierce 2002). We therefore
show these rules in the appendix and focus on the terms related to
the knowledge base (see Figure 4).

A term representing a query can be directly evaluated to a list
of objects (E-QUERY). The query reduction rule queries the knowl-
edge system for all \(?X\) for which the axiom \(K \models ?X : C\) is true.
As \(\lambda_{DL}\) relies on DL-safe queries, only objects actually defined
in the signature are allowed. For simplicities sake, we consider the
result to be a list and introduce a \(\sigma\)-operator that takes care of com-
munication between the knowledge system and \(\lambda_{DL}\). 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 \(K \models (a, ?X) : R\). Again, anonymous objects are not
considered and the result is converted into a list by the \(\sigma\)-operator.
In case of equivalence, both terms must first be reduced to val-
ues (E-EQI 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). \(\lambda_{DL}\) considers two primitive values only equivalent if
they 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 \(\lambda\)-abstractions and avoid these cases during type-checking.

Evaluation of type-dispatch terms (see Fig. 5) is somewhat
special. The terms to be dispatched is first reduced to an object
(E-DISPATCH). The semantics can then test the object by case
until one of them matches (E-DISPATCH-SUCC and E-
DISPATCH-FAIL). For each case the knowledge system is queried
whether the axiom \(K \models a : C\) is true. Due to the open-world
querying, it might happen that the knowledge system cannot com-
pute such a membership. In this case, the type-dispatch uses its
default case to continue evaluation.

5. **Type system**

The most distinguishing feature of the type system for \(\lambda_{DL}\)
is the addition of concept expressions, built according to the rules
described in Table 2 as types in the language. For constructs
unrelated to the knowledge system, this has little impact. However,
computation of upper and lower bounds change due to the addition
of concepts.

**Least-Upper Bound and Greatest-Lower Bound**

Computation of the least-upper bound of two types \(S\) and \(T\), as, e.g., required for
typing if-then-else terms is done by a special judgment dubbed lub
(see Fig. 6). In case of a least-upper bound for primitive types, we
simply assume the types to be equal, making the least-upper bound
the type itself (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 type for the list. For two functions, \(S_1 \rightarrow S_2\) and
\(T_1 \rightarrow T_2\), the greatest-lower bound of the types \(S_1\) and \(T_1\) as well
as the least upper bound of \(S_2\) and \(T_2\) are computed.
The greatest-lower bound of two types \( S \) and \( T \) works analogous to the least-upper bound. Two primitive types must be again equal, making their greatest lower bound the type again. The greatest lower bound of two concepts \( C \) and \( D \) is the concept \( C \sqcap D \). Lists are again reduced to a greatest lower bound of their type. The same is true for functions. The exact rules can be seen in the Fig. 8 in the appendix.

**Typing knowledge-base unrelated constructs** Given the judgment for the least upper bound of two types, the typing rules can now be defined (see Fig. 7). Typing of let, fixpoint operations, applications, abstractions, variables and primitive values does not differ from standard approaches. Typing of if-then-else statements relies on the lub-judgment to create a type \( W \) that combines both branches.

**Typing of knowledge-base related constructs** Typing of terms related to the knowledge base is summarized in Figure 8. Queries (T-QUERY) have a concept associated with them - therefore, the result of the evaluation will be of type \( C \) list. To avoid unsatisfiable queries, the knowledge system is queried whether the concept \( C \) is satisfiable. If it is, typing does not assign a type to the term and type-checking aborts with an error. Projections (T-PROJ) require a term of type \( C \) and can then be typed by the inverse of the relation used for the projection. While this may seem confusing on first sight, it is actually the most precise type that can be assigned to this term. Range-definitions of roles are often extremely general (e.g., the range definition for influencedBy). Equivalence (T-EQP) and T-EQ-P) simply requires two well typed values that are the same type. Fig. 9 summarizes the rules.

The type annotation. A cons function (T-CONS) can be typed if it is applied to a term of type \( T_1 \) and a term of type \( T_2 \) list. The new list can be typed using the least-upper-bound judgment to create the type \( T_3 \) list. The remainder are standard list typing rules: A null function takes a well-typed list and returns a boolean value. Head needs a well-typed list of type \( T \) list and returns a list of the same type. Fig. 8 summarizes the rules.

In terms of lists, we restrict ourselves to lists of objects for demonstration purposes. An empty list (T-NIL) can be typed using the type annotation. A cons function (T-CONS) can be typed if it is applied to a term of type \( T_1 \) and a term of type \( T_2 \) list. The new list can be typed using the least-upper-bound judgment to create the type \( T_3 \) list. The remainder are standard list typing rules: A null function takes a well-typed list and returns a boolean value. Head needs a well-typed list of type \( T \) list and returns a list of the same type. Fig. 8 summarizes the rules.
single objects can be typed using a nominal concept — a concept expression created through enumerating its members.

Typing of a type-dispatch (see Fig. 11) is similar to typing of a if-then-else. Given that the term being dispatched is a well-typed concept \( D \), the type of the term is the least-upper-bound of all branches. We use \( \text{tub} \) as a shortcut for the repeated application of the \( \text{tub} \)-judgment. Additional checks ensure meaningful cases. First of all, the intersection between \( C_i \) and \( D \) should not be equivalent to \( \bot \), as it would then be impossible for the case to ever match. Second, since cases are checked sequentially, it should not happen that a case is subsumed by a case occurring before it.

**Subtyping** Subtyping rules are summarized in Fig. 10. Any type is always a subtype of itself (S-RELFF). 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 \( \mathcal{K} \models C \subseteq D \) (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 \( S \) is a subtype of \( T \) list if \( S <: T \) is true (S-LIST). A function is subsumed by another if its domain is more specific, but its co-domain more general (S-FUNC).

**Algorithmic type-checking** Algorithmic type-checking is completely syntax driven. For instance, transitivity, which could fail to be syntax-driven, is handled by the knowledge system in case of concept expressions, while primitive types do not include any subtype relations.

### 6. Type soundness

In this section, we prove the soundness of \( \lambda_{DL} \): If a program is well-typed, it does not get stuck. As with many other languages, there are exceptions to this rule (e.g., down-casting in object-oriented languages, cf. [Igarashi et al. 2001]). For \( \lambda_{DL} \), these exceptions concern lists. We therefore show that if a program is well-typed, then the only way it can get stuck is if it reaches 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 \( T_1 \rightarrow T_2 \), then \( v \) is of the form \( \lambda(x : S_1)t_1 \) with \( S_1 <: T_1 \).
3. If \( v \) is a value of type \( C \) list, then \( v \) is either of the form (cons \( v_1 \ldots \)) 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. \( \square \)

Given Lemma 1 we can show show that a well-typed term is either a value or it can take a 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 \( t \leadsto t' \). 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 \( t \leadsto t' \).

**Proof.** By induction on the derivation of \( \Gamma \vdash t : T \). We proceed by examining each case individually.

- \((T-LET): t = \text{let } x = t_1 \text{ in } t_2, \Gamma \vdash t_1 : T_1, \Gamma, x : T_1 \vdash t_2 : T_2\). By hypothesis, \( t_1 \) is either a value or it can make a step. If it can, rule E-LET applies. If its a value, E-LETV applies (see Fig. 12).
- \((T-FIX): t = \text{fix } t_1, \Gamma \vdash t_1 : T_1 \rightarrow T_2, \Gamma \vdash t : T_2\). By induction hypothesis, \( t_1 \) is either a value or it can take a step. If it can take a step, rule E-FIXV applies. If its a value, by the canonical forms lemma (Lemma 1), \( t_1 = \lambda(x : T_1)t_2 \). Therefore, rule E-FIXV applies.
- \((T-APP): t = t_1 t_2, \Gamma \vdash t_1 : T_1 \rightarrow T_2, \Gamma \vdash t_2 : T_1, \Gamma \vdash t : T_1\). By hypothesis, \( t_1 \) and \( t_2 \) are either a values or they can take a step. If they can take a step, rules E-APP1 or E-APP2 apply. If both are values, then by the canonical forms lemma (Lemma 1), \( t_1 = \lambda(x : T_1)t_1 \) and rule E-APPABS applies.
- \((T-IF): t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3, \Gamma \vdash t_1 : \text{bool}, \Gamma \vdash t_2 : S, \Gamma \vdash t_3 : T, \Gamma \vdash t : W\). By induction hypothesis, \( t_1 \) is a value or it can take a step. If it can take a step, rule E-IF applies. If its a
value, then by Lemma 1 $t_1 = true$ or $t_1 = false$. In this case, either rules E-IF-TRUE or E-IF-FALSE apply.

(T-ABS): Immediate since $\lambda(x : T_1).t_2$ is a value.

(T-VAR): Impossible since we’re only looking at closed terms.

(T-TRUE): Immediate, since true is a value.

(T-FALSE): Immediate, since false is a value.

(T-SUB): Result follows from induction hypothesis.

(T-NIL): Immediate, since nil is a value.

(T-CONS): $t = cons\ t_1\ t_2$, $\Gamma \vdash t_1 : C$, $\Gamma \vdash t_2 : D$ list. By hypothesis, $t_1$ and $t_2$ are either values or they can take a step. If they can take a step, rules E-CONS1 and E-CONS2 apply (see Fig. 13). Otherwise, the term is a value.

(T-NIL): $t = null\ t_1$, $\Gamma \vdash t_1 : T$ list, $\Gamma \vdash t : bool$. By hypothesis, $t_1$ is either a value or it can take a step. If it can take a step, rule E-NULL applies. If its a value, by Lemma 1 $t_1 = nil$ or $t_1 = (cons\ v_1\ ...)$ Then either E-NULL-TRUE or E-NULL-FALSE apply.

(T-HEAD): $t = head\ t_1$, $\Gamma \vdash t_1 : T$ list, $\Gamma \vdash t : T$. By hypothesis, $t_1$ is either a value or it can take a step. If it can take a step, rule E-HEAD applies. Otherwise, by Lemma 1 $t_1 = nil$ or $t_1 = (cons\ v_1\ ...)$ Then either rule E-HEADV applies or the term is in the accepted normal form $t = head\ nil$.

(T-TAIL): $t = tail\ t_1$, $\Gamma \vdash t_1 : T$ list, $\Gamma \vdash t : T$ list. By hypothesis, $t_1$ is either a value or it can take a step. If it can take a step, rule E-TAIL applies. Otherwise, by Lemma 1 $t_1 = nil$ or $t_1 = (cons\ v_1\ ...)$ Then either rule E-TAILV applies or the term is in the accepted normal form $t = tail\ nil$.

(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-DISPATCH): $t = case\ t_0$ of $\begin{cases} \text{default } t_{n+1} \Gamma \vdash t_0 : D, \Gamma \vdash t : W \end{cases}$ By hypothesis, $t_0$ is either a value or it can take a step. If it can take a step, rule E-DISPATCH applies. If its a value, by Lemma 1 $t_0 = a$. If $\frac{a}{\Gamma}$ is non-empty, either rules E-DISPATCH-SUCCE or E-DISPATCH-FAIL apply. Otherwise, rule E-DISPATCH-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-EQNP): $t_1 \neq t_2$, $\Gamma \vdash t_1 : \Pi_1$, $\Gamma \vdash t_2 : \Pi_1$. 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, then 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 $\lambda_{DL}$ does not differ from standard approaches, e.g., as described in [Pierce 2002]. Therefore, the proof is omitted.

Lemma 3 (Least-Upper-Bound). Let $S, T$ and $W$ be types. If $\text{sub}(S, T) \Rightarrow W$, then $S \sqsubseteq W$ and $T \sqsubset W$.

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., $S \sqsubseteq W$ and $T \sqsubset W$ holds.

Concepts: $S = C$, $T = D$, $W = C \sqcup D$. Since $K \models C \sqsubseteq C \sqcup D$ and $K \models D \sqsubseteq C \sqcup D$, $S \sqsubseteq W$ and $T \subset W$ 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.

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 $\Gamma \vdash t : T$. We proceed by examining each case individually.

(T-LET): $t = let\ x = t_1 in\ t_2$, $\Gamma \vdash t_1 : T_1$, $\Gamma, x : T_1 \vdash t_2 : T_2$. There are two ways $t$ can be reduced: E-LET and E-LETV.

1: $t = let\ x = t_1 in\ t_2$ By induction hypothesis, $t_1 \rightarrow t_1'$ preserves the type. Therefore, by rule T-LET, $t_1' \rightarrow T_2$.

2: $t' = [x \mapsto v_1]t_2$. By Lemma 2 typing is preserved, therefore $t' : T_2$.

(T-FIX): $t = fix\ t_1$, $\Gamma \vdash t_1 : T_1 \rightarrow T_1$, $\Gamma \vdash t : T_1$. There are two rules by which $t$ can be reduced: E-FIX and E-FIXV.

1: $t' = fix\ t_1'$. By induction hypothesis, $t_1 \rightarrow t_1'$ preserves the type. Therefore, by T-FIX, $t_1' \rightarrow T_1$.

2: $t' = [x \mapsto \lambda(x : T_1) v_2] t_2$. By Lemma 2 substitution preserves the type. Therefore, $t' : T_1$.

(T-APP): $t = t_1 t_2$, $\Gamma \vdash t_1 : T_1 \rightarrow T_2$, $\Gamma \vdash t_2 : T_2$. There are three rules by which $t'$ can be computed: E-APP1, E-APP2 and E-APPABS.

1: $t' = t_1 t_2$. By induction hypothesis, $t_1 \rightarrow t_1'$ preserves the type. Therefore, $t' : T_1$.

2: $t' = v_1 t_2 \rightarrow v_1 t_2$. Same case as (1).

3: $t' = (\lambda x : T_1 t_1) v_2 \rightarrow [x \mapsto v_2] t_2$. By Lemma 2 substitution preserves typing. Therefore, $t' : T_1$.

(T-IF): $t = if\ t_1$ then $t_2$ else $t_3$, $\Gamma \vdash t_1 : bool$, $\Gamma \vdash t_2 : S$, $\Gamma \vdash t_3 : T$, $\text{sub}(S, T) \Rightarrow W$, $\Gamma \vdash t : W$. There are three rules by which $t'$ can be computed: E-IF-TRUE, E-IF-FALSE and E-IF.

1: $t' = t_2$. By rule T-IF, $\text{sub}(S, T) \Rightarrow W$ and by Lemma 3 $S \subset W$, therefore $t' : W$ by rule T-SUB.

2: $t' = t_3$. Same case as (1).

3: $t' = if\ t_1$ then $t_2$ else $t_3$. By induction hypothesis, $t_1 \rightarrow t_1'$ preserves the type. Therefore, by rule T-IF, $t' : W$.

(T-ABS): Vacuously fulfilled $\lambda(x : T_1).t_2$ is a value.

(T-VAR): Cannot happen.

(T-TRUE): Vacuously fulfilled since $t = true$ is a value.

(T-FALSE): Vacuously fulfilled since $t = false$ is a value.

(T-NIL): Result follows from induction hypothesis.
(T-NULL): \( t = \text{null} \ \Gamma \vdash t : T \ \text{list}, \ \Gamma \vdash t : \text{bool} \). By hypothesis, \( t \) is either a value or it can take a step. If it can take a step, rule E-NULL applies. If its a value, by Lemma 1 \( t_1 = \text{nil} \) or \( t_1 = (\text{cons} \ v_1 \ ... \) . If \( t_1 = \text{nil} \) then rule E-NULL-TRUE applies. In case of \( t_1 = (\text{cons} \ v_1 \ ...) \), rule E-NULL-FALSE applies.

(T-HEAD): \( t = \text{head} \ t_1 \ \Gamma \vdash t : T \ \text{list}, \ \Gamma \vdash t : T \). There are two rules by which \( t' \) can be computed: E-HEAD and E-HEADV.

(1): \( t' = \text{head} \ t_1' \). By induction hypothesis, \( t_1 \rightarrow t'_1 \) preserves the type. Therefore, by T-HEAD, \( t' \). (2): \( t_1 = \text{cons} \ v_1 \ v_2, \ \Gamma \vdash t : T, \ t' = v_1 \). Result is immediate, since \( v_1 : T \).

(T-TAIL): \( t = \text{tail} \ t_1 \ \Gamma \vdash t : T \ \text{list}, \ \Gamma \vdash t : T \ \text{list} \). There are two rules by which \( t' \) can be computed: E-TAIL and E-TAILV.

(1): \( t' = \text{tail} \ t_1' \). By induction hypothesis, \( t_1 \rightarrow t'_1 \) preserves the type. Therefore, by T-TAIL, \( t' : T \). (2): \( t_1 = \text{cons} \ v_1 \ v_2, \ \Gamma \vdash t : T \ \text{list}, \ t' = v_2 \). Result is immediate, since \( v_2 : T \).

(T-QUERY): \( t = \text{query} \ C, \ \Gamma \vdash t : C \ \text{list} \). By applying rule E-QUERY, \( t' = \text{cons} \ a_1 \ ... \). However, for each \( a \), it is known that \( K \models a : C \), therefore \( \{ a \} <: C \) holds for each \( a \) and \( \{ a \} \sqcup ... <: C \) list.

(T-PROJ): \( t = t_1.R, \ \Gamma \vdash t : C, \ \Gamma \vdash t : (\exists R.C) \). 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): \( t' = \sigma(\{X \mid \exists X \in O \land K \models (a,?X) : R\}) = \text{cons} \ b_1 \ ... \). For \( a \), it is known that \( K \models a : C \) and for each \( b \) is known that \( K \models (a,b) : R \) holds. Therefore, \( K \models b : (\exists R.C) \) must hold for each \( b \). Thereby, \( \{ b \} \sqcup ... <: (\exists R.C) \) and by S-LIST \( \{ b \} \sqcup ... \) list <: (\exists R.C) list.

(T-DISPATCH): \( t = \text{case} \ t_0 \ of \)

- type C_1 as x_1 -> t_1
- type C_n as x_n -> t_n
- default t_{n+1}

\( \Gamma \vdash t_0 : D, \ \Gamma \vdash t_1 : T_1, ... \ \Gamma \vdash t_n : T_n, \ \Gamma \vdash t_{n+1} : T_{n+1}, \ \text{let}(T_1,...,T_{n+1}) = W, \ \Gamma \vdash t : W \)

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

(1): \( t' = \text{case} \ t_0 \ of \)

- 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-DISPATCH, \( t' : W \).

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

(3): \( t' = \text{case} \ a \ of \)

- type C_2 as x_2 -> t_2
- ...
- type C_n as x_n -> t_n
- default t_{n+1}

\( \Gamma \vdash t_2 : T_1, ... \ \Gamma \vdash t_n : T_n, \ \Gamma \vdash t_{n+1} : T_{n+1}, \ \text{let}(T_2,...,T_{n+1}) = W', \ \Gamma \vdash t : W' \)

The removal of the first case causes T-DISPATCH to assign type \( t' : W' \). Removal of \( T_i \) makes \( W' \) more specific then \( W \), but \( W' <: W \) holds. Therefore by T-SUB \( t' : W \).

(T-EQN): \( t_1 = t_2, \ \Gamma \vdash t_1 : C, \ \Gamma \vdash t_2 : D, \ \Gamma \vdash t : \text{bool} \).

There are six 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): \( t' = \text{true} \). Immediate by rule T-TRUE.

(2): \( t' = \text{false} \). Immediate by rule T-FALSE.

(3): \( t' = \text{true} \). Immediate by rule T-TRUE.

(4): \( t' = \text{false} \). 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, \( t' : \text{bool} \).

(6): \( t' = v_1 = v_2 \). By induction hypothesis, \( t_1 \rightarrow t'_1 \) preserves the type. Therefore, by rule T-EQN, \( t' : \text{bool} \).

(T-APP): \( t_1 = t_2, \ \Gamma \vdash t_1 : T_1, \ \Pi_1, \ \Gamma \vdash t_2 : T_2, \ \Pi_2 \). Same as T-EQN.

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

As a direct consequence of Theorems 1 and 2 a well-typed closed term does not get stuck during evaluation.

7. Related work

\( \lambda_{DL} \) is generally related to the integration of data models into programming languages. We consider different ways of integrating such 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\footnote{https://www.w3.org/DOM/} for XML \cite{Wallace99}). This approach has also been applied to semantic data. Representations can vary, however the most popular ones include axiom-based approaches (e.g., \cite{Horridge2011}), graph-based ones (e.g., \cite{Carroll2004}) or statement-based ones (e.g., RDF4\footnote{http://rdf4j.org/}). 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 \cite{O'neil08}, XML \cite{Wallace99}, \cite{Lammel06}, Alagic and Bernstein \cite{2009}, and more generally \cite{Lammel05, Syntse2013}. 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 \cite{Oren05}.
They rely on domain and range specifications for predicates to tie as mapping approaches when it comes to schema information explicitly given statements. Furthermore, they face the same difficulties that use static type-checking for querying and light scripting in order to avoid runtime errors (Ciobanu et al. 2014, 2015). However, all mapping frameworks have problems with the high number of potential types in semantic data sources.

Precompilation A separate precompilation step, where the source code is statically analyzed beforehand for DSL usage and then verified or 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 (Wassermann et al. 2007). The approach has been applied to semantic data in a limited manner (Groppe et al. 2003)—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 (Bierman and Wren 2005) and easy data access to relational and XML data (Bierman et al. 2003). Another example concerns programming language support for the XML data model specifically in terms of regular expression type, as in the languages CDue (Benzen and Wren 2003) and XDuce (Hosoya and Pierce 2003). 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 (Hanus 1994). However, $\lambda_{DL}$ emphasizes type-checking on data axiomatized in logic over the integration of the logic programming paradigm into a language. Given its typecase constructs, $\lambda_{DL}$ is also related to other forms of typecases (Abadi et al. 1993; Cravoy et al. 2002; Lümmel and Jones 2003). However, since semantic data cannot be adequately expressed with existing typing mechanics, these approaches cannot fully solve the problems.

Language extensions and custom approaches have also been implemented for semantic data. In one approach (Paar and Vrandečić 2011), the C# compiler was extended to allow for OWL and XSD types in C#. The main technical difference to $\lambda_{DL}$ is that $\lambda_{DL}$ makes use of the knowledge system for typing and subtyping judgments. $\lambda_{DL}$ 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 (Ciohanu et al. 2014, 2015). 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. Discussion and future work

In this paper, we have motivated, introduced and studied a type system for semantic data that is built around concept expressions as types as well as queries in a simple $\lambda$-calculus. 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. However, the work can be extended in several ways.

Gradual typing A byproduct of achieving type safety are the rather hard restrictions by the schema. This can be seen best for the $\text{influencedBy}$ role as described in the example. The knowledge system could not prove that $\text{MusicArtist} \sqsubseteq \text{influencedBy}$. Therefore, the influences of music artists could not be computed directly. In the example, this was a correct choice as not every music artist was influenced by something. But there are also scenarios where it is reasonable to assume that, for a specific data source, this will be the case even though the schema does not explicitly state so, simply because schemata for semantic data strive to be applicable to different and evolving data sources. Also, the semantic data source may have been created by conversion from more constrained data, e.g., in a SQL database. One way to include such background knowledge of a developer would be to adopt ideas originating from gradual typing (Siek and Tahmasebi 2006) for $\lambda_{DL}$. A `lentient' $\lambda$ could be introduced, which accepts values even though the subtyping relation cannot be proven. However, even in this case, one would still check if the intersection of the functions domain and of the value applied to the function is non-empty in order to avoid grave mistakes.

$\lambda_{DL}$ and System F So far, we have only considered a simply typed $\lambda$-calculus for the integration of semantic data into a functional language. However, programming languages typically feature polymorphic definitions, e.g., for list-processing function combinators. A comprehensive integration of description logics and polymorphism (with System F (Reynolds 1983) as starting point) including aspects of subtyping is not straightforward.

Modification of the semantic data Of course, it is also desirable that semantic data can be modified by an extended $\lambda_{DL}$. However, due to facts inferred by the knowledge system, this is non-trivial. Given the facts about music artists in Listing 3 and the goal to remove the (implicit) fact that the Beatles have made a song, this cannot be removed directly. Instead, either the fact that the Beatles are of type $\text{MusicArtist}$ or the fact that they have been played by coolFm must be removed. In order to integrate modification of knowledge systems into $\lambda_{DL}$, the theory of knowledge revision based on AGM theory (Qui et al. 2006) must be considered and properly integrated into the language.

Enhanced querying Another area of future work concerns the query system. Queries, as they are currently implemented, are limited in their expressive power. A simple extension are queries for roles, such as $\text{influencedBy}$ that result in sets of pairs. Typing such queries is possible via the addition of tuples to $\lambda_{DL}$. 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., (Bourhis et al. 2013)).

References


10 https://bitbucket.org/openrdf/alibaba
http://jastor.sourceforge.net/
http://semanticweb.org/wiki/RDFReactor
http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator
A. Appendix

A.1 Remaining reduction rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>let x = v₁ in t₂ → [x ↦ v₁]t₂</td>
<td>E-LETV</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td></td>
</tr>
<tr>
<td>let x = t₁ in t₂ → let x = t’₁ in t₂</td>
<td>E-LET</td>
</tr>
<tr>
<td>fix (λx : T₁.t₂) → [x ↦ (fix (λx : T₁.t₂))]t₂</td>
<td>E-FIXV</td>
</tr>
<tr>
<td>t₁ → t’₂</td>
<td></td>
</tr>
<tr>
<td>fix t₁ → fix t’₁</td>
<td>E-FIX</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td></td>
</tr>
<tr>
<td>t₁t₂ → t’₁t₂</td>
<td>E-APP1</td>
</tr>
<tr>
<td>t₂ → t’₂</td>
<td></td>
</tr>
<tr>
<td>v₁t₂ → v₁t’₂</td>
<td>E-APP2</td>
</tr>
<tr>
<td>(λx : T₁.v₂) → [x ↦ v₂]t₁</td>
<td>E-APPABS</td>
</tr>
<tr>
<td>if true then t₂ else t₃ → t₂</td>
<td>E-IF-TRUE</td>
</tr>
<tr>
<td>if false then t₂ else t₃ → t₃</td>
<td>E-IF-FALSE</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td></td>
</tr>
<tr>
<td>if t₁ then t₂ else t₃ → if t’₁ then t₂ else t₃</td>
<td>E-IF</td>
</tr>
</tbody>
</table>

Figure 12: Reduction rules for constructs unrelated to KB.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁ → t’₁</td>
<td>E-CONS1</td>
</tr>
<tr>
<td>cons t₁ t₂ → cons t’₁ t₂</td>
<td></td>
</tr>
<tr>
<td>t₂ → t’₂</td>
<td></td>
</tr>
<tr>
<td>cons v₁ t₂ → cons v₁ t’₂</td>
<td>E-CONS2</td>
</tr>
<tr>
<td>null nil → true</td>
<td>E-NULL-TRUE</td>
</tr>
<tr>
<td>null cons v₁ v₂ → false</td>
<td>E-NULL-FALSE</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td>E-NULL</td>
</tr>
<tr>
<td>null t₁ → null t’₁</td>
<td></td>
</tr>
<tr>
<td>head cons v₁ v₂ → v₁</td>
<td>E-HEADV</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td>E-HEAD</td>
</tr>
<tr>
<td>head t₁ → head t’₁</td>
<td></td>
</tr>
<tr>
<td>tail cons v₁ v₂ → v₂</td>
<td>E-TAILV</td>
</tr>
<tr>
<td>t₁ → t’₁</td>
<td>E-TAIL</td>
</tr>
<tr>
<td>tail t₁ → tail t’₁</td>
<td></td>
</tr>
</tbody>
</table>

Figure 13: Reduction rules for lists.
A.2 Greatest-lower bound

\[
\begin{align*}
glb(\pi_1, \pi_1) & \Rightarrow \pi_1 \quad \text{[GLB-PRIMITIVE]} \\
glb(C, D) & \Rightarrow C \cap D \quad \text{[GLB-CONCEPT]} \\
glb(S, T) & \Rightarrow W \quad \text{[GLB-LIST]} \\
glb(S \text{ list}, T \text{ list}) & \Rightarrow W \text{ list} \\
lub(S_1, T_1) & \Rightarrow W_1 \quad glb(S_2, T_2) \Rightarrow W_2 \quad \text{[GLB-FUNC]} \\
glb(S_1 \rightarrow S_2, T_1 \rightarrow T_2) & \Rightarrow W_1 \rightarrow W_2
\end{align*}
\]

Figure 14: Greatest lower bound of types.

A.3 Prototypical implementation

A prototypical implementation, showing the feasibility of $\lambda_{DL}$, is available at [http://west.uni-koblenz.de/de/lambda-dl](http://west.uni-koblenz.de/de/lambda-dl). The interpreter itself is written in F# while relying on a Java-based HermiT reasoner for inferencing. Most of the interpreter is based on the approach shown by [Pierce 2002]. An important difference (besides the actual rules) is that evaluation and typing functions take a knowledge base as an additional parameter. In case of HermiT knowledge bases, a wrapper is passed to those functions. The wrapper serializes queries issued by the evaluation and typing functions and calls a Java program, which then in turn deserializes the queries and calls the reasoner.