Type-safe programming with OWL in Semantics4J (PREPRINT)

Carsten Hartenfels[2], 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. Programming with OWL is error-prone due to the lack of type-safe integration into programming languages. While generic types such as *Resource* can represent anything the data can model, they make it impossible to error-check programs. Mapping ontological concepts into types of the programming language on the other hand often cannot capture the ontology completely and duplicates knowledge that a semantic reasoner already has. Semantics4J, an extended Java compiler, allows for type-safe programming with OWL by integrating DL expressions as types and values into the programming language. This paper presents a prototype that supports our theoretical concept [4] by relying on an extended type-checking process building upon a reasoner as well as class expression queries that are being used as types themselves.

1 Introduction

While OWL allows for capturing knowledge in a natural manner, programming with such a data source is error-prone. There are two common approaches to program with OWL. Using generic types (1) such as *Resource* as exemplified by [1] can represent anything the data can model, but it ignores structural properties or ontological knowledge and therefore makes it impossible for type systems to detect errors in the program based on semantic reasoning with ontological concepts. To allow such an error detection, mapping approaches (2) map ontological concepts into types of the programming language. However, such mappings are limited by the expressiveness of the type system used in the target language and they duplicate knowledge that a reasoner working over such data already has. As an example, consider data about musicians, actors and mutual influences represented in description logics as shown in Listings 1 and 2.

```
1 ∃recorded.Song ⊑ Musician
2 Actor ⊓ Musician
3     ⊑ ActingMusician
4 RockMusician ⊑ Musician
5
```

Listing 1. Example ontology.

```
1 (hendrix,machineGun) : recorded
2 machineGun : Song
3 elvis : Actor
4 elvis : RockMusician
5 (hendrix,elvis) : influencedBy
```

Listing 2. Example data.

In the ontological part (Listing 1), everyone who has recorded a song is defined to be a `Musician` (line 1). `ActingMusician`s are entities who are both `Actor` and `Musician` (lines 2–3). A `RockMusician` is a special form of a `Musician` (lines 4).

In the data part (Listing 2), `hendrix` has recorded the song `machineGun` (lines 1–2). The entity `elvis`, who is both `Actor` and `RockMusician` is said to have influenced `hendrix` (lines 3–5). A mapping that converts such data into types of a programming language faces difficulties because of implicit knowledge, e.g. the fact that `hendrix` is a `Musican` or the lack of multiple inheritance in common programming languages. With regard to the latter, consider `ActingMusician`, which is subsumed by both `Actor` and `Musican`. But it must also handle the intricacies of the `influencedBy` role, which is used very broadly. As a result, many mappings cannot fully capture the ontology for the purpose of type checking the program.

Our system relies on a type system extension. It uses DL expressions as types and extends the type-checking process to benefit from a semantic reasoner. This has been explored from a theoretical point of view in [4] where we showed that this approach does indeed yield a type-safe language. In this paper, we present an backward compatible extension of the Java programming language called Semantics4J that implements the methodology in the Java programming language.

2 Example use case

As an example, consider an application written for the data as defined in Listings 1 and 2. Such an application could query for all `Musicians` and then print their influences. Listing 3 depicts the outline of such a program while leaving out the specifics of a *getInfluences* method for mapping from a `Musician` to his influences.

```
1  public class Influences from "music.rdf" {
2    public static Set<∃«:influencedBy»⁻ · ⊤>
3      getInfluences(... artist) { ... }
4
5    public static void main(String args[]) {
6      for («:Musician» artist : query-for(":Musician")
7        System.out.format("%s was influenced by %s",
8          artist.getName(),
9          String.join(", ", names(getInfluences(artist)))
10       );
11   }
12 }
```

Listing 3. Program querying for musicians and listing their influences.

Line 1 of the program specifies the class name and data source ”music.rdf“ the program is written for. Lines 2–3 contain the signature of the *getInfluences* method. This method returns a set of values where each value is an instance of the DL expression $\exists \texttt{influencedBy}^{-}.\top$. In the *main* method of the program, Line 7 contains a query for all `Musicians` over which the for-loop iterates. For each `Musician`, the *getInfluences* method is called and its results are joined into a single string via the *getName* method which returns the fragment identifier part of the entities IRI. An important feature of Semantics4J is that DL expressions are both types and values. DL expressions as types are used during the type-checking of the program. DL expressions as values, e.g.,

as a parameter to the *query-for* operator, are treated as strings and are implicitly converted to roles and concepts. This allows them to be dynamic, enabling the construction of a query on the fly. Assuming that a concept name, e.g., `Actor`, is being passed to the program via command line, the query in line 6 could also query for the intersection of the two concepts (see Listing 4).

```
6   for («:Musician» artist : query-for(":Musician"⊓args[0])
```

Listing 4. Example for a dynamically constructed query.

Typing of *query-for* is dependent on the given input values. However, only the static parts of such queries can be used for typing as the dynamic part may change during runtime. Therefore, the query displayed in Listing 4 is only typed with the concept `Musician`. This is also true if the dynamic part is not given as a program parameter, but rather as another variable holding a reference to some string.

A simple approach to the *getInfluences* method may be to assume that all `Musician`s have an influence. The method could therefore take an instance of the DL expression $\exists$`influencedBy`.$\top$ and query for influences directly (see Listing 5).

```
2   public static Set<∃«:influencedBy»⁻·⊤>
3       getInfluences(∃«:influencedBy»·⊤ artist) {
4     return artist.(":influencedBy");
5   }
```

Listing 5. Example for problematic code.

However, this code contains a type error—since it unknown whether `Musician` is a subconcept of $\exists$`influencedBy`.$\top$, it is problematic to assume that this relation holds. To enforce type safety, Semantics4J requires the programmer to do a type case instead (see Listing 6). This also requires the presence of a default case which applies if it is either unknown whether the `Musician` has an influence or if it is known that the `Musician` has no influence.

```
2   public static Set<∃«:influencedBy»⁻·⊤>
3     getInfluences(«:Musician» artist) {
4      switch-type (artist) {
5       ∃«:influencedBy»·⊤ influencable {
6         return influencable.(":influencedBy");
7       }
8       default { return Collections.emptySet(); }
9     }
10  }
```

Listing 6. Type-safe example code.

3 Implementation

Semantics4J builds upon ExtendJ, a JastAdd Extensible Java compiler [2]. It can parse Java programs extended with the constructs used in the examples. Type-checking is

done by forwarding all decisions about the relation of ontological concepts to a reasoning service. In case of the current implementation, we rely on HermiT [3]. Semantics4J transforms the extended program into standard JVM bytecode, which may query the reasoning service during runtime to gain access to the data (see Fig. 1). The current implementation can be found at https://github.com/hartenfels/Semantics4J.

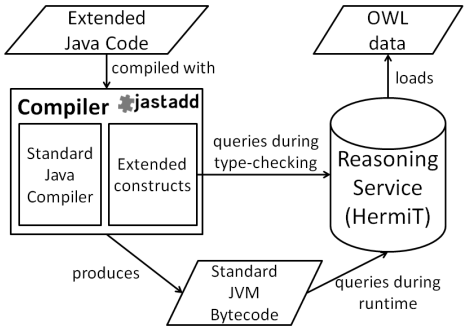

Fig. 1. Overview over the Semantics4J architecture.

4 Summary

In this paper, we present Semantics4J, an extended Java compiler for type-checking programs working with OWL. It allows for using ontological concepts as types by integrating the reasoner into type-checking. Furthermore, it provides typed queries. Typing information allows the system to reject potentially problematic programs that may produce runtime-errors. However, some restrictions have been put onto the system. A pure extension that does not modify existing rules of a compiler requires a strict and unambiguous syntactic separation between existing rules and newly integrated rules. The distinction between types and values also creates some restrictions as values can be created and modified during runtime. Typing of e.g., *query-for* can only consider values given directly to the operator while variable parts are ignored. A deeper knowledge about the dynamics of such variables would be desirable to allow for a more precise typing, but requires an analysis of the flow of the program that is currently not implemented.

References

1. J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proc. WWW - Alternate Track Papers & Posters*, pages 74–83. ACM, 2004.
2. T. Ekman and G. Hedin. The JastAdd Extensible Java Compiler. In *Proc. OOPSLA '07*, pages 1–18. ACM, 2007.
3. B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang. HermiT: An OWL 2 Reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014.
4. M. Leinberger, R. Lämmel, and S. Staab. The Essence of Functional Programming on Semantic Data. In *Proc. ESOP 2017*, LNCS, pages 750–776. Springer, 2017.