On Requirements for Federated Data Integration as a Compilation Process

Alessandro Adamou and Mathieu d'Aquin

Knowledge Media Institute, The Open University, United Kingdom
{alessandro.adamou, mathieu.daquin}@open.ac.uk

Abstract. Data integration problems are commonly viewed as interoperability issues, where the burden of reaching a common ground for exchanging data is distributed across the peers involved in the process. While apparently an effective approach towards standardization and interoperability, it poses a constraint to data providers who, for a variety of reasons, require backwards compatibility with proprietary or non-standard mechanisms. Publishing a holistic data API is one such use case, where a single peer performs most of the integration work in a many-to-one scenario. Incidentally, this is also the base setting of software compilers, whose operational model is comprised of phases that perform analysis, linkage and assembly of source code and generation of intermediate code. There are several analogies with a data integration process, more so with data that live in the Semantic Web, but what requirements would a data provider need to satisfy, for an integrator to be able to query and transform its data effectively, with no further enforcements on the provider? With this paper, we inquire into what practices and essential prerequisites could turn this intuition into a concrete and exploitable vision, within Linked Data and beyond.

Keywords: Linked Data, Query federation, Compilers

1 Introduction

Open standards play an unquestionable role in the evolution of data interoperability, and an eminent example can undoubtedly be found in Linked Data. This set of principles and standards favors uniform federated querying across multiple data providers at the hands of applications. These applications, in turn, can serve many use cases, one being the exposure of an API that publishes aggregated data from multiple sources. One cannot expect such an API to conform to the same interoperability principles as the sources it draws from, due to possible backwards compatibility with legacy systems and other industrial constraints.

Implementing this process certainly benefits from standardized mechanisms for federated querying such as those offered by SPARQL, however, the translation of query results into the desired specifications relies upon the integrator itself. In Linked Data, the line is drawn on semantic interoperability with reuse

of resources, be they terms of a vocabulary or data items, which leaves some loose ends, for instance as to how data URIs should be transformed if necessary.

Software compilers operate in analogy with use cases like the many-to-one scenario above, as in there, a single program analyses and links multiple files (the source code) into an output that is then transformed into an object that complies with the target specification (the machine-executable program). As the compiler literature is vast and its history long, we look into avenues for capitalizing on it.

With this paper, we intend to discuss the merits of these research questions:

RQ1. Is it possible to formulate a data integration problem based on federated querying as a compilation process?
RQ2. If the answer to RQ1 is yes, what information should a data integration environment expose, for us to treat it like software code to be compiled?

Being able to answer yes to RQ1 would open up a range of possibilities for the principles and practices of data federation. Most of all, it would allow us to bring the craftsmanship of compiler experts into the fields of data integration to research the optimal answer to RQ2. This could help solve specific integration problems or optimize existing solutions, effectively allowing us to discuss 'data compilation' as a discipline by its own right.

In Section 2, we outline the above data API scenario in greater detail. On its basis, in Section 3 we reformulate the associated data integration process in terms of the classic analysis/synthesis model of compilers. Finally, in section 4 we give an insight as to what further research is being carried out on this vision.

2 Scenario

Given a collection of known linked data providers (hereinafter, *sites*) that expose a hierarchy of RDF graphs (*datasets*) through an interface such as SPARQL endpoints and/or dereferenceable CoolURIs, the goal is to produce a *data feed* published on a single endpoint (*integrator*), which selectively reuses data from the sites and encodes them in a custom *target language*. Not uncommonly in industrial and traditional data management, this language must give the impression that their provider is 'in control'. To that end, it satisfies the following:

1. a single represented item appears as an attribute/value map;
2. attributes are named according to an in-house naming convention (i.e. no ontology property names are reused);
3. values are represented as items per (1), up to a fixed level of recursion beyond which they are identified by a reference. These references are URIs resolved by the same API that produces the data feed (i.e. the API is self-contained).

These requirements are in stark contrast with the principles of Linked Data, which dictate that providers should be free to use their own vocabularies and identifiers, and that both should be reused, rather than concealed, by others.

Finally, we assume that *some* sites publish meta-level descriptions of their datasets as VoID or Data Cube manifests. These, combined with other meta-level information computed by the integrator (cf. RQ2), form the *site profile*.

3 Data integration in the front/back end compiler model

A compilation problem can be formulated in terms of requirements of the target machine code, e.g. that it has to be executable by processors of a certain family with certain instruction sets and register layout. Data integration can also be approached in terms of the requirements of the final data feed, i.e. the compiled data in a target language that agents of a certain type, human or machine, must be able to read and interpret. We aim at identifying whether a similar parallel is possible in the operational model of the solutions to these problems.

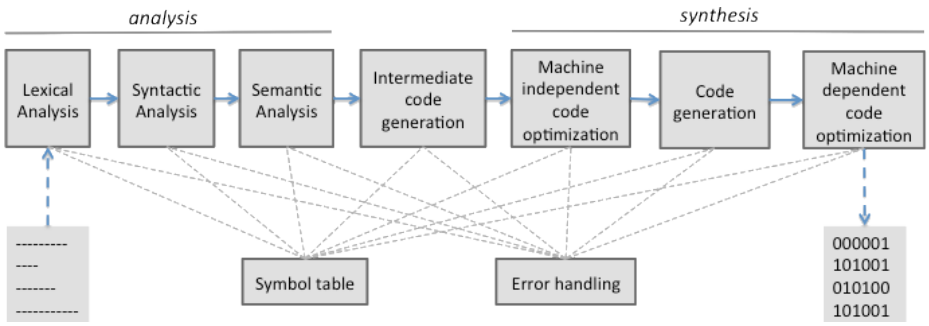

Fig. 1. Compilation phases in the classical front/back end model.

A traditional model of compiler design is depicted in Figure 1. It has as its pivotal phase the generation of code in an intermediate language for the program at hand. This phase is preceded by an *analysis* part, which comprises lexical, syntactic and semantic analysis, and is followed by a *synthesis* part, where the code in the target language is generated and optimizations are performed [5]. Also, it is the compiler itself that has to fulfil the requirements of most phases, especially synthesis ones, whereas source code is mostly required to be correct for the analysis phases not to fail (few programmers will take compiler optimizations into account when writing the code). Synthesis is also called the *back end* of the compiler, and the other phases its *front end*. The following sections break down these operational strands and seek a correspondent for each in the above scenario through query federation, where the burden of performing most of the integration lies on a single peer that we control, and that corresponds to the compiler.

3.1 Intermediate code generation

Intermediate code is generated in a language defined for and used by the compiler alone, in order to satisfy certain optimality conditions. An intermediate language is not necessary, but without one, a full native compiler would be required for each target architecture, instead of only a re-implementation of the synthesis. Porting this notion to our linked data integration scenario, without an intermediate language, all the code present in a *compilation unit*, i.e. an instance of

output of a site (in RDF or SPARQL results) would be rewritten directly into the target language, thus reducing the potential for detecting redundant references and collapsing them in the data feed (cf. Section 2 req. 3, self-containment).

We will assume RDF triples to be the formalism of choice, given their natural inclination to several layers of interoperability, and adapt the analysis and synthesis parts accordingly[1]. Also, there is an interesting parallel with the three-address intermediate code of compilers, which is a serialized form of decision trees on binary operators. The intermediate language itself is the combination of triples and a naming convention for their nodes, e.g. resources and literals, which is entirely up to the integrator. This naming convention is not required to make sense on the outside world, that is, we disregard inherently Linked Data features of RDF such as dereferenceable URIs[2]. We require, however, the following:

1. **globality**. The naming convention *should* be able to rewrite URIs of aligned resources (e.g. via `owl:sameAs` statements) into the same URI.
2. **completeness**. It *must* apply to every possible URI that appears in the data supplied by any site involved in the integration process;

A naming convention *supports* a URI pattern if it satisfies globality for all its occurrences. Completeness can be satisfied even for URIs whose scheme is not known a priori: a function that, for instance, prepends a prefix to the original URI if its pattern is unsupported would be a sufficient naming convention.

3.2 Front end: analysis and assembly

Software compilers perform **lexical, syntactic and semantic analysis** on the source code and derivative data structures, to check if the code is an occurrence of the programming language and respects semantic requirements such as type matching and variable scopes. These phases are usually backed by **symbol tables**, i.e. data structures maintained by the compiler that keep track of the occurrences of entities such as variable names, function signatures and objects.

To begin with, we define the *compilation unit* to be an instance of the output of a site (in RDF or SPARQL results) given a query on it. The role of these analyses in data integration is ambivalent, depending on what elements we choose to be the symbols, syntax and semantics of the process. If we establish that the symbols are the constituents of RDF (URIs, literals, bnodes etc.), then the analysis part coincides with that of an RDF parser; there are no site-specific requirements other than delivering well-formed compilation units, at the price of not being able to perform per-site optimizations. If instead we apply the lexicon-syntax-semantics paradigm differently, then we can expect advantages in translating compilation units to the intermediate language. Here, we will assume that the patterns for constructing URIs in each dataset are part of the lexicon, and their instances are kept track of in the symbol table.

[1] One could also opt for OWL as intermediate language, though we would have to be wary of the caveats of translating RDF triples into OWL axioms appropriately.

[2] The way RDF processors generate blank node IDs can be such an implementation.

Assuming the above in our compiler model, the semantic analysis phase can now include matching of RDF types with URI patterns [2] and heuristics for detecting and collapsing equivalent entities [6]. As part of a process called *linking*, where a single object is built out of multiple compilation units, the results of this analysis (which we can assume to reside in an *assembly* plan maintained by the integrator) can be applied to the generation of unified intermediate code. The question then arises as to what information about the sites and their datasets should the assembly plan contain in order to perform linking effectively. In the present scenario and compilation model, the goal is to avoid query broadcasting and its network and computational overhead: it should be possible to determine the eligibility of a site as a candidate for providing relevant data, therefore worth querying, and the shape of the data it can deliver, so as to determine what ad-hoc query to issue to it. We are currently investigating into how intermediate code that transforms URIs to satisfy globality and completeness can be generated if:

1. all entities are typed, either explicitly or implicitly;
2. the relationship between a URI pattern in a dataset and the types of its instances, or their identifying property values, is explicit;
3. it is known which conventions are employed in the assertions that are materialised in the data, and which are left to inferencing: for instance, which property of an inverse property pair is used in asserted statements.[3]

Related to (1), explicit types can be found in VoID class partitions[4] and Data Cube slicing[5], or by sampling the dataset directly; implicit ones are obtainable through inferencing on the compilation units and the ontologies that describe their vocabularies. Requirement (2) is largely unsatisfied by the existing standards and literature and is mostly left to research. Finally, (3) finds partial fulfilment in VoID property partitioning combined with ontologies.

3.3 Back end: optimization and target encoding

An optimizing compiler modifies generated intermediate and target code in order to improve certain efficiency measures that the compiler supports. What this translates to in a data integration scenario is largely under investigation, but we began by identifying certain tasks; as part of target-independent optimization:

- Consolidation of matching data items and elimination of redundant attributes, through ontology alignment and other means.
- Handling query expansion; identify and construct further queries to be issued to sites in order to perform just-in-time linking.
- Serial and parallel scheduling of these queries built through query expansion.

As part of target-dependent optimization:

[3] Of course some of these limitations can be overcome in SPARQL by adding optional triple patterns and unions to a query, but at a generally impracticable overhead.
[4] Vocabulary of Interlinked Datasets, `http://www.w3.org/TR/void/`
[5] RDF Data Cube vocabulary, `http://www.w3.org/TR/vocab-data-cube/`

- Determining the optimum threshold for including recursively-referenced items in one feed, beyond which their data are replaced with references.
- Rewriting attribute names to avoid name clashing with attributes in other data feeds resulting from a different query to the same sites.

While we do not expect target-dependent optimization to raise significant requirements for site profiles, we hypothesize that target-independent optimization tasks can partly rely on the symbol tables generated in the front end phases.

4 Conclusions

We have made a case of formulating typical legacy data integration problems using the paradigm of software compilers, prognosticating that in so doing compiler optimizations may contribute to this cause. Although we have not identified previous evidence of data integration problems formulated using compilers, there is recent literature on formulating models and challenges for data integration on the Web. Paton et al. have postulated a model for continuously improving integration in a purely Linked Data setting [4], which significantly inspired our work. Hoang et al. have collated scholarly literature on the practices of semantic mashups, which our use case shares several contact points with [3]. As for compiler-like approaches in the Semantic Web, we previously laid out some seminal work in the context of interpreting ontology networks [1]. We are currently elaborating on the position given by this paper as applied to a concrete instantiation of its scenario, now in the process of formalizing back end requirements.

References

1. Alessandro Adamou, Paolo Ciancarini, Aldo Gangemi, and Valentina Presutti. The foundations of virtual ontology networks. In Marta Sabou, Eva Blomqvist, Tommaso Di Noia, Harald Sack, and Tassilo Pellegrini, editors, *I-SEMANTICS 2013 - 9th International Conference on Semantic Systems, Graz, Austria*, pages 49–56. ACM, 2013.
2. Mathieu d'Aquin and Alessandro Adamou. Extracting URI patterns from SPARQL endpoints. Technical report, Knowledge Media Institute, 2014.
3. Hanh Huu Hoang, Tai Nguyen-Phuoc Cung, Duy Khanh Truong, Dosam Hwang, and Jason J. Jung. Semantic information integration with linked data mashups approaches. *IJDSN*, 2014, 2014.
4. Norman W. Paton, Klitos Christodoulou, Alvaro A. A. Fernandes, Bijan Parsia, and Cornelia Hedeler. Pay-as-you-go data integration for linked data: opportunities, challenges and architectures. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Proceedings of the 4th International Workshop on Semantic Web Information Management, SWIM 2012, Scottsdale, AZ, USA*, page 3. ACM, 2012.
5. A.A. Puntambekar. *Compiler Design.* Technical Publications, 2010.
6. Ziqi Zhang, Anna Lisa Gentile, Isabelle Augenstein, Eva Blomqvist, and Fabio Ciravegna. Mining equivalent relations from linked data. In *51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, Sofia, Bulgaria, Volume 2: Short Papers*, pages 289–293. The Association for Computer Linguistics, 2013.