A Type-theoretic Approach to Cloud Data Integration

Pavel Shapkin and Gregory Pomadchin
Department of Cybernetics and Information Security,
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute),
31 Kashirskoe shosse, Moscow, Russian Federation

Keywords: Type Theory, Data Integration.

Abstract: We propose an architecture that helps to integrate data accessible via cloud application APIs. The central part of the platform is the domain ontology which is based on type theory. Typing is used to automate the building of integration solutions as well as for automatic verification of program code and compatibility between components.

1 INTRODUCTION

Cloud applications rapidly penetrate into companies of all sizes attracting users with low low costs. Although as side effect data end up "locked" in the cloud and this dramatically complicates migration between different systems and inhibit transition to new solutions. At the same time every SaaS application often has its own API, and a pressure to keep the costs low doesn't allow SaaS developers to implement complete support for different standards and data exchange tools. In modern environment of fast-evolving technologies there is often a need to move to a new system without sacrificing historical data. In this context comprehensive solution to migration problem is needed that will not only simply map data formats but will also preserve data consistency as well as classifier relations. Over 30% of SaaS-applications users use more than one system and require integration.

In this paper we describe architecture of a cloud-based platform for migration, consolidation and integration of data. It is a program complex that facilitate connection of multiple SaaS applications using automatically created integration solution.

Core value of this solution for target users is the capability to untie the data and "free" it from specific cloud apps. Usage of such an environment allows one to smoothly transfer the data between applications, as well up- or download it using different formats and protocols. Users can exploit their data in a whole new way as a self-standing entity that can be freely moved between cloud applications.

The central component of the system is an ontology model of a subject field that accumulates knowledge about object structure and their relative mapping. It is augmented every time a new system is attached. Technology is unique in using ontology for automated search for connections between different systems. Ontology is represented as a conversion system between different object representations united by the means‘ of type theory.

The paper is structured in the following way. Section 2 describes general principles of building addressed architecture. In section 3 the mathematical basis and formalization of architectural components are given. Section 4 covers important features of program implementation. In the conclusion we sum up obtained results and outline the future work.

2 LEVELS OF DATA REPRESENTATION FOR INTEGRATION

Following processes form the basis of integration solution: data extraction, transformation and loading. These processes are very similar to those in ETL systems (extract-transform-load) (Vassiliadis, 2009). Let's consider these steps and data structures required for applying them.

We start from the data transformation as it plays a central role in the whole process. Key requirement for transformation building toolkit is "sense-making" of the results. Formally this means that the resulting object can be safely uploaded into the target system.

Shapkin P. and Pomadchin G..
A Type-theoretic Approach to Cloud Data Integration.
DOI: 10.5220/0005495301640169
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 164-169
ISBN: 978-989-758-106-9

In other words we need an ability to *validate* transformations against some data model. In terms of programming transformations are some kind of functions and thus naturally they can be validated with the aid of *type checking*. Data model is represented by the type structure (or object-oriented classes). Hence the data model is the crucial piece of the integration platform and enables to check the consistency of data transformations.

Now let's consider data extraction and loading. Every time the data is extracted an external system's API is called and the obtained data is transformed into the ontological representation. Data load is a reverse process. So, in our method, data extraction and loading steps can be split into two operations. First — the API call — is specific for every external system or for every standard data exchange protocol. For the most part it can be implemented with standard libraries or SDK if available. Second operation is essentially a transformation of data to or from the ontological data model.

2.1 Core Ontological Representation

According to what we've described above core data model is meant for data transformations consistency check. For greater effectiveness this model has to be composed of most accurate possible object classes descriptions, their dependencies and consistency conditions. This data model combined with a set of transformations we will call an ontology (Gruber et al., 1993): on one hand it provides complete description of subject field semantics, on the other hand using transformations it enables new data "deduction" from available data. The data representation that is consistent with ontology will be referred as *ontological*. One of the main features of ontological representation is that all connections between objects must be represented in explicit form: no intermediate identificators are allowed.

Let's assume that for every domain entity there is a corresponding *type*. Formal definition can be found in 3.1, here we understand types as in programming languages: the simplest way is to interpret them as records with named fields. In their turn these fields can be represented as functions that return corresponding values from the object — "getters".

In turn, external data formats used in systems being integrated can be associated with types used in corresponding software libraries serving certain protocols: SDKs for API access or for certain data formats processing.

In such a manner we've unified ontological representation and external representation — both are collections of types. The main difference is that unified ("ontological") representation explicitly describes interconnections of objects, whereas types of "external" representation usually describe data structure within single message transferred over a communication protocol used in API of a given system. An API message usually describes only a single entity: connections with other objects are represented by their identificators. And since identificators are only meaningful within a single application — each application can use specific identification method or even specific list of entities — in ontological representation identificators are omitted altogether. The transformation of data from external representation into ontological is carried out by *connectors*.

2.2 Representation of External API Messages

Transformations used to exchange data with external systems are different from all other in a way that ontological representation is always only "on one end": it is either the result (data extraction) or the argument (data loading) of the transformation. On the "other end" of transformation is a data model that represents the structure of the API message.

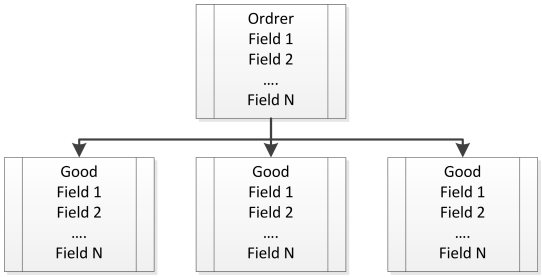
The key difference between the API message structure and the corresponding ontological representation is that in messages links between objects are defined through identificators. in ontological representation all connected objects must be presented in explicit form (see figure 1).

Thus, type assignment will be carried out based on the API semantics and structure of a message and every service will be unambiguously mapped into corresponding class structure.

Acquiring message data structure can also be done in two steps by obtaining a syntactic structure first and then transforming it into a semantic one. For instance, for XML data message a syntactic structure will be a DOM-tree of the XML document while the semantic structure is defined by corresponding XML schema and can be represented with some abstract object.

```
<Order uuid = "376fc72e-..." attrs>
       <Good uuid = "0247db0d-..."
           attrs>
       <Good uuid = "e8c64677-..."
           attrs>
       <Good uuid = "4f9c19aa-..."
           attrs>
</Order>
```

(a) External service XML message



(b) Ontological model

Figure 1: Mapping of external service message into ontological model.

3 FUNCTIONAL REPRESENTATION OF CONCEPTUAL DOMAIN MODELS

3.1 Introduction to Type Theory

Let us introduce some basic definitions of the type theory which we will be using further as the text goes.

Definition 3.1. Type system — is a flexible syntactical method of proving nonexistence of certain kinds of behaviuor in a program using classification of language expressions according to the kinds of in values they compute.

In formal, the Type Theory (TT) studies processes of type inference and type checking in programs. For this purpose, it is necessary to have a formal representation of programs — λ-calculus, where programs are interpreted like the composition of computable functions. We will be giving only major definitions omitting details that can be found in (Harrison, 1997; Wolfengagen and Ismailova, 2003).

Basic construct in λ-calculus — λ-term — is defined as follows.

Definition 3.2. (λ-terms)

- *Variables* are denoted by arbitrary strings of letters and numbers.
- *Constants* are also denoted by strings. We will distinguish them based on a context.
- *Abstraction* of λ-term M by a variable x — $\lambda x.M$ is an unary function of parameter x.
- *Application* — is an application of a function (term) M to an argument N and is denoted as (MN). Braces have left associativity and can be omitted if possible.

Key moment here is a concept of a function as an obect. This, in particular, relieve from the necessity to consider miltiplace functions — they can be regarded as a function of single variable, computational result of wchich is a new function and so on.

Basic rule of computing ("reduction") a value of expression is (β):

$$(\lambda x.M)N = M[x := N],$$

where $M[x := N]$ is a result of substituting all occurances of x for N in M. This rule is also equiped with a set of rules that enable reduction of not only full term but of it's parts as well.

Types are defined as follows:

Definition 3.3. (Types)

- If V is a type variable or constant then V is a type.
- If V and U are types then $V \to U$ is a type.

And finally, the typing rules. Let Γ be some context, then $\Gamma \vdash m : V$ means that term m has type V in a context Γ. For instance for simple terms like variables this is stated explicitly. For the consideration of architecture at the top level, without details of the implementation, it is enough to observe *simply typed λ-calculus* wich has the following system of typing rules:

$$\frac{\Gamma \vdash t : V \to T \quad \Gamma \vdash u : V}{\Gamma \vdash t\,u : T} \quad \text{(Application)}$$

$$\frac{\Gamma, m : V \vdash n : T}{\Gamma \vdash \lambda m.n : (V \to T)} \quad \text{(Abstraction)}$$

We also introduce a special notation for types that have a similar structure. In programming terms, these types called "generics". For example, the type (*class* in object oriented programming) $List[T]$ represents a family of types for lists of elements of type T. Another example is the type "optional value of type T " denoted as $Option[T]$ which allows to represent a value that may be missing. Although, on the one hand, this notation has a complex structure and, in

fact, depends on the type of parameter, we understand it as a name of an atomic simple type.

3.2 Mathematical Basis for Ontology and Inference Formalization

It follows from the above that domain ontology could be represented by a system of abstract types equipped with a set of transformation functions. The set of transformation functions consists of different groups. The first group of transformations are directly represented as functions by the programmer. For instance, function that transforms *Document*-type objects into *Invoice*-type objects. The fact that not every document can be transformed into an invocice can be reflected in function type making it's value optional: $Option[Invoice]$.

Other groups of transformations could be derived implicitly, e.g.:

- Field accessors of objects; for example, function *name* 'transforms' an object of type *Person* into a string, and function *work* transforms an object of type *Employee* into an object of type *Organization*.
- Coercion functions implementing subtyping; for inctance, if types *Person* and *Company* inherit type *Client* that means there are implicit type transformation functions:

  $$Person \to Client \text{ and } Company \to Client,$$

  implementing rules "Each person (company) is a client".
- Inverse coercion functions; for instance, in terms of previous example the assertion "Some clients are persons (companies)" can be represented by the corresponding functions returning the value of an optional type:

  $$Client \to Option[Person] \text{ and}$$
  $$Client \to Option[Company].$$

The rules above a uniform way to represent both specialized transformations and field values as well as taxonomical relations. And vice versa we can see that specialized transformations can interpreted as extensions of the domain taxonomic structure — for instance, from given example it can be inferred that type *Invoice* can be regarded as subtype of the *Document* type[1].

[1]In explicit form this idea is implemented in some programming languages with so called coercive subtyping (Luo, 1999).

4 FUNCTIONAL MODEL OF THE INTEGRATION PLATFORM

Domain entities as well as the structure of their API representation can be presented as types that allows to simulate the system purely with mathematical methods — as a set of functions.

To complete the platform, in addition to the ontology (with transformations), it is required to have special objects to communicate with external systems by API — connectors. Let us consider the structure of these objects.

Each of the integrated applications provides access to different types of objects. Within the data synchronization task such an access means an opportunity to send and receive a set of objects in order to synchronize data between systems.

To structure the connectors the 'repository' pattern was used: each connector is a set of *repositories* — objects of type $Repository[T]$, which provide access to entities of type T. In other words, the repository is a set of two functions:

$$pull : Unit \to List[A] \text{ and } push : List[A] \to Unit.$$

The function *pull* is responsible for loading objects of type A. The function *push* is responsible for uploading objects of type A.

$Unit$ is a special type with only one value. We use the type $Unit$ as input or output value types of functions that do not accept or do not return any values. In other words, the value of type $Unit$ has no information attached to it and is unique because there is no way to distinguish it from other values of this type.

The main problem in the development of connectors is the development of bidirectional conversions between ontology objects and types of API messages.

4.1 Levels of Data Access

As it was described above, due to the presence of explicit relations there is no need for identifiers in ontological representation while the communication with external services predominantly consists in the exchange of identifiers: to upload an object with links we have to upload linked objects first to get their identifiers and so on. For simplicity, we assume that identifiers are strings — type $String$.

For this purpose connectors are equipped with a set of advanced repositories with additional functions:

$$pushId : List[A] \to List[String] \text{ and}$$
$$pullId : List[String] \to List[A].$$

The function *pushId* is responsible for uploading set of objects of type A and returns a list of identifiers

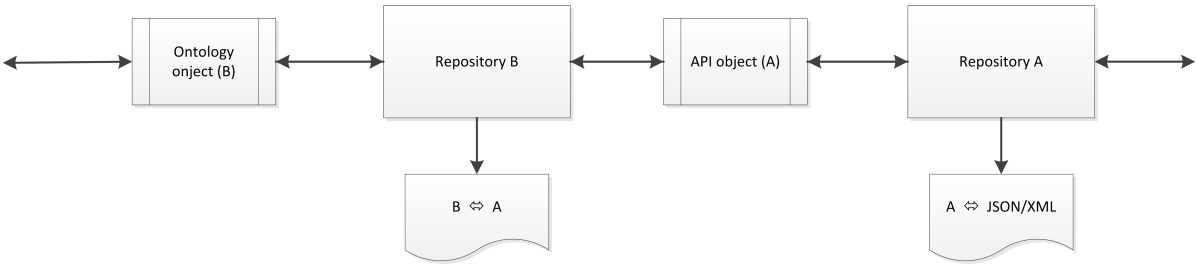


Figure 2: Multilevel repository structure.

of uploaded objects. The function *pullId* is responsible for loading objects by a list of identifiers.

As a result, there are two types of repositories: one operates on ontology types while the other operates with API classes by sending or receiving API messages to or from external services (see figure 2). Repository which operates with ontology classes is responsible for converting ontological representations to external message structure types and vice versa.

There was made an attempt to automate or to semi-automate building of conversions of external API messages to inner classes. As a result, the best decision was to generate classes and conversion rules for external API messages. The same approach may be used to automate building of the internal transformations.

5 IMPLEMENTATION

Software implementation of the proposed architecture is made in the Scala programming language. To solve the problem of automating the composition requires a detailed research of the type system and means of processing type information in Scala (Odersky et al., 2004). In formal, the solution can be expressed as a set of rules or theorems describing the methodology of building the function composition of the original list. Software implementations of such solutions can be divided into two groups:

- expression of all rules with the type system;
- code generation of rules, with reference to the types of processed objects.

The first approach in fact carries all the processes of building function compositions on the compiler. In Scala, this feature is available through the presence of the so called implicit values that are implicitly calculated by the compiler: the programmer specifies only the type of the desired value. On the one hand, this approach is most similar to the direct use of the Type Theory and the Curry-Howard isomorphism. On the other hand, Scala compiler implicits search is often ineffective and fails on a large amounts of calculations.

The second approach is in defining a “macros” — functions, generating in code fragments in compile-time. In this case, macros can use the information about the types. In fact, the macros in Scala implement a sort of static reflection: a developer can describe programs that are driven by type structures, however type-dependent parts of the program are evaluated at a compile-time with a type checking. This ensures type safety of the resulting code. In contrast to the use of implicit values, which are calculated implicitly by the compiler, the programmer must describe the whole algorithm of code-generation explicitly.

6 CONCLUSIONS

We described an approach to integration solutions creation which is based on representing the domain model semantics in form of abstract type systems. It helps to employ the same tools for automatic code verification as well as automatic integration solution generation. Authors implemented the proposed approach in the development of Tylip cloud integration platform[2].

REFERENCES

Gruber, T. R. et al. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5:199–199.

Harrison, J. (1997). Introduction to functional programming. *Lecture Notes, Cam*.

Luo, Z. (1999). Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130. 00118.

Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the Scala programming language. *LAMP-EPFL*.

[2]http://www.tylip.com

Vassiliadis, P. (2009). A Survey of Extract-Transform-Load Technology:. *International Journal of Data Warehousing and Mining*, 5(3):1–27.

Wolfengagen, V. E. and Ismailova, L. Y. (2003). *Combinatory logic in programming*. Citeseer.