Preventing Technical Errors in Data Lake Analyses with Type Theory

Alexis Guyot, Éric Leclercq, Annabelle Gillet, Nadine Cullot

Preventing Technical Errors in Data Lake Analyses with Type Theory

Alexis Guyot[0000−0001−5896−7693], Éric Leclercq[0000−0001−6382−2288], Annabelle Gillet[0000−0002−4204−9262], and Nadine Cullot[0000−0003−1307−3287]

LIB, Université de Bourgogne, Dijon, France
{firstname}.{lastname}@u-bourgogne.fr

Abstract. Data analysts compose various operators provided by data lakes to conduct their analyses on big data through complex analytical workflows. In this article, we present a formal framework based on type theory to prevent technical errors in such compositions of operators. This framework uses restrictions on type definitions to transform technical errors into type errors. We show how to use this framework to prevent errors related to schema or model transformations in analytical workflows. We provide an open-source implementation in Scala which can be used to detect errors at compile time.

Keywords: Data Lakes · Type Theory · Big Data Analytics.

1 Introduction

Data analysts use data lakes to conduct their analyses on big data. These platforms provide features to store, manage and analyse big data with a schema-on-read paradigm [2]. In other words, heterogeneous data are ingested and stored as is and only cleansed, structured and integrated when needed. Data lakes associate datasets with insightful metadata to describe and control their content and relationships. Data lakes provide various operators to search, transform, enrich and analyse data at different levels of abstraction (data, schema, model and metadata). For example, they may provide data preparation operators to cleanse data and integrate schemas. They may also provide analytical operators like classifiers or graph miners whose results can be reused as new metadata.

Data analysts compose these various operators to build complex analytical workflows. Composed operators may act on different data models (relational, semi-structured, graph, etc.) and levels of abstraction. They may use different theoretical foundations (relational or linear algebra, graph theory, statistics, etc.) and execution paradigms (stream processing, map-reduce, GPU, etc.). Nevertheless, an invalid composition of operators can be a source of technical errors in analyses.

Therefore, a formal framework based on solid theoretical foundations is required to prevent technical errors in the compositions of operators of data lake analytical workflows. In this article, we propose to use type theory as the foundation of this formal framework. We aim to transform technical errors into type

errors through type definitions. Thus, the absence of errors in a composition can be proven using type theory and verified by the compiler. We show how to use types to specify operators acting on multiple data models and levels of abstraction. We also show how our framework prevents errors in a composition of such operators.

2 Related Works

Introducing type safety in operators and languages to prevent errors and misuse is a recurrent solution. Functional languages like OCaml, Haskell or F# use strong static typing based on different type theories to prevent misuse through type inference. Proof assistants like Agda and Coq implement type theories, *resp.* the Unified Theory of dependent Types and the Calculus of Constructions. Recent analytical frameworks also include type safety. For example, type systems for linear algebra are proposed by Griffioen [6] and by Muranushi *et al.* [12]. The Spark framework [15] provides advanced structures like *Datasets* that add type safety guarantees over *DataFrames*. The Tensor Data Model [5] adds types in tensors to provide type safety and schema inference on tensor operators.

Other approaches to prevent errors and misuse of operators focus more on the data they are dealing with. Firstly, metadata can be used to understand the data better. They provide descriptions and constraints on the datasets and their relationships. Metadata models can be based on various formalisms such as logic [7], graphs [13] or UML [16]. Each formalism provides a different trade-off between expressiveness and restrictiveness. Therefore, only some metadata models can formally prevent errors and misuse through mathematical, structural or reasoning properties. Furthermore, existing metadata models are either tailored for specific use cases or not generic enough to be used in different contexts [13]. Secondly, data can be structured using pivot models with well-established algebras. Several pivot models have been proposed for heterogeneous platforms, such as the nested relational model in [1], the JSON semi-structured model in [8], typed tensors in [5] or the RDF graph model in [4]. However, pivot models may hinder the flexibility required in data lakes by restricting the set of available operators. For example, languages based on the nested relational model cannot express transitive closure and, therefore, several graph operators [1]. Finally, foundations of multi-model languages [10,11,14] use category theory to formally represent and control the effects of operators on data. However, this approach mainly focuses on representing different data models with a single formalism. It does not focus on the errors that can occur by composing different operators.

In contrast to the existing approaches, we tackle the problem of preventing errors in analytical workflows by controlling the heterogeneity of data models and the navigation between different levels of abstraction (data, schemas, models, metadata) in a unified way using type theory. This approach allows us to formally ensure the consistency of data manipulations and transformations throughout the data lake workflows.

3 Type-Theoretical Framework

Type theory is a constructive formalism. It can be used to define type systems through construction rules [3]. It defines the concepts of well-formed types and sub-types. A **well-formed type T** (written $T : Type$) is a type having at least one rule allowing its construction. A well-formed type S can be a **sub-type** of another well-formed type T (written $S <: T$). In this case, type S can be used in every situation in which type T is required.

Type theory also defines type constructors. These allow the construction of various composite types such as function, generic, dependent and product types. A **generic type** $A[B]$ is a type that is parameterised by another type like $List[T]$ or $Matrix[T]$. A **dependent type** $\Pi_{(x:B)}A(x)$ is a type that is parameterised by values of another type. For example, a dependent type $\Pi_{(size:Int)}Vec(size)$ represent vectors of a certain size. A dependent type $\Pi_{(name:String)}T(name)$ represent attributes of type T with a certain name. For the sake of readability, we will refer to the latter dependent type as $n \twoheadrightarrow T$ in the following. A **product type** is a composite type (written $A \times B \times ... \times Z$) whose elements are heterogeneous tuples $(a, b, ..., z)$ with $a : A$, $b : B$ and $z : Z$.

We show how to use type theory to prevent technical errors related to schema or model transformations in a composition of operators. Analytical workflows often require such transformations to connect analytical operators acting on different data models. However, transforming the schema or the model of data may be error-prone due to the differences between models. For example, the relational model does not allow multi-valued or nested attributes, whereas semi-structured models such as JSON do. Inconsistencies between data schemas and models due to transformations may result in a technical error. Therefore, we want to verify that a composition of operators does not contain any erroneous schema or model transformation.

Preventing such errors requires representing with types two levels of abstraction (schema, model) and at least two different data models. We propose to represent: 1) one data model allowing multi-valued and nested attributes like JSON, and; 2) one data model not allowing such attributes like the relational model.

$$\frac{\Gamma \vdash S <: Schema \quad \Gamma \vdash RelationSchema[S] : Type}{\Gamma \vdash Relation[S] <: Model[S]}$$

(a) Relational model.

$$\frac{\Gamma \vdash S <: Schema \quad \Gamma \vdash JsonSchema[S] : Type}{\Gamma \vdash JSON[S] <: Model[S]}$$

(b) JSON model.

Fig. 1: Type definitions for data models.

We use products of dependent types to represent data schemas. More precisely, we use products of $n \twoheadrightarrow T$. In schema types, using a dependent type to combine an attribute name with its type can be useful to prevent other errors. For example, it may allow the inference of a schema for the data returned by operators. Analysts can use this inferred schema to prevent inconsistencies or

$$\frac{}{\Gamma \vdash RelationValue[Base] : Type}$$

(a) Type definition for $RelationValue$.

$$\frac{\begin{array}{c}\Gamma \vdash F <: n \twoheadrightarrow T \\ \Gamma \vdash S <: Schema \\ \Gamma \vdash F \times S <: Schema \\ \Gamma \vdash RelationValue[T] : Type \\ \Gamma \vdash RelationSchema[S] : Type\end{array}}{\Gamma \vdash RelationSchema[F \times S] : Type} \qquad \frac{\begin{array}{c}\Gamma \vdash F <: n \twoheadrightarrow T \\ \Gamma \vdash RelationValue[T] : Type\end{array}}{\Gamma \vdash RelationSchema[F \times SNil] : Type}$$

(b) Type definition for $RelationSchema$.

Fig. 2: Restrictions of the relational model on schemas.

$$\frac{}{\Gamma \vdash JsonValue[Base] : Type} \qquad \frac{\begin{array}{c}\Gamma \vdash T : Type \\ \Gamma \vdash JsonValue[T] : Type \\ \Gamma \vdash List[T] : Type\end{array}}{\Gamma \vdash JsonValue[List[T]] : Type} \qquad \frac{\begin{array}{c}\Gamma \vdash S <: Schema \\ \Gamma \vdash JsonSchema[S] : Type\end{array}}{\Gamma \vdash JsonValue[S] : Type}$$

(a) Type definition for $JsonValue$.

$$\frac{\begin{array}{c}\Gamma \vdash F <: n \twoheadrightarrow T \\ \Gamma \vdash S <: Schema \\ \Gamma \vdash F \times S <: Schema \\ \Gamma \vdash JsonValue[T] : Type \\ \Gamma \vdash JsonSchema[S] : Type\end{array}}{\Gamma \vdash JsonSchema[F \times S] : Type} \qquad \frac{}{\Gamma \vdash JsonSchema[SNil] : Type}$$

(b) Type definition for $JsonSchema$.

Fig. 3: Restrictions of the JSON model on schemas.

misuses in the analytical workflow. We define a super-type $Schema$ and two sub-types $SNil$ and $F \times S$, with $F <: n \twoheadrightarrow T$ and $S <: Schema$. This definition allows the construction of schema types with any number of attributes: $SNil$ is the type of empty schemas, $F \times SNil$ is the type of schemas with one attribute of type F, $F \times S$ is the type of schemas with at least one attribute of type F. In the following, and for the sake of readability, we assume the existence of a super-type $Base$ for the main data types like numbers, strings, booleans, dates, etc.

We use generic types parameterised with schema types to represent data models. We define a super-type $Model[S]$ and two sub-types $Relation[S]$ and $JSON[S]$, with $S <: Schema$. Figure 1 presents type definitions for these three types. For a given schema type S, type $Relation[S]$ (*resp.*, $JSON[S]$) is well-formed if type $RelationSchema[S]$ (*resp.*, $JsonSchema[S]$) is well-formed.

We define generic types $RelationSchema[S]$ and $RelationValue[T]$ (*resp.*, $JsonSchema[S]$ and $JsonValue[T]$), with $S <: Schema$ and $T : Type$ (figures 2 and 3). These types link the two levels of abstraction by specifying

the restrictions that the models apply to schemas. $RelationSchema[S]$ (*resp.*, $JsonSchema[S]$) recursively ensures that all the attributes of the schema conform to the model. $RelationValue[T]$ (*resp.*, $JsonValue[T]$) defines value types that the model allows. The relational model only allows attributes with scalar values (figure 2a). The JSON model allows attributes with scalar, multi-valued and nested values (figure 3a).

$$\frac{\begin{array}{c}\Gamma \vdash S1 <: Schema \\ \Gamma \vdash S2 <: Schema \\ \Gamma \vdash M1[S1] <: Model[S1] \\ \Gamma \vdash M2[S2] <: Model[S2]\end{array}}{\Gamma \vdash M1[S1] \Rightarrow M2[S2] : Type}$$

(a) Operators.

$$\frac{\begin{array}{c}\Gamma \vdash M1[S1] <: Model[S1] \\ \Gamma \vdash M2[S2] <: Model[S2] \\ \Gamma \vdash M3[S3] <: Model[S3] \\ \Gamma \vdash f : M1[S1] \Rightarrow M2[S2] \\ \Gamma \vdash g : M2[S2] \Rightarrow M3[S3]\end{array}}{\Gamma \vdash safeCompo(f, g) : M1[S1] \Rightarrow M3[S3]}$$

(b) Composition of operators.

Fig. 4: Type definitions for operators and compositions.

We represent operators with function types acting on models. Therefore, a composition of operators is a function type taking as inputs two operator types and returning a new operator type if their composition is not erroneous. Figure 4 presents type definitions for operators and compositions. The type of a composition is well-formed if its inputs are well-formed, *i.e.* if a series of construction rules leads to the composition. Any error, for example an erroneous schema or model transformation, breaks the series of rules and results in a type error. According to type theory, a complete series of construction rules leading to a well-formed type is proof of the absence of error.

Compilers can detect type errors. Therefore, we propose an open-source implementation in Scala of the types presented in this article[1]. The implementation uses the rich typing features provided by Scala and the Shapeless library[2]. It can be used to encapsulate operators in analytical workflows and verify the absence of errors related to schema or model transformations at compile time.

4 Conclusion

In this article, we have presented a formal framework to prevent technical errors in data lake analytical workflows by finely controlling the compositions of operators. This framework is based on type theory. It uses types to represent data in different models and levels of abstraction and to express restrictions on operator inputs and outputs. We have proposed an open-source implementation in Scala using the compiler to prevent errors. The current limits of our approach are the lack of representation for constraints that apply to data (*e.g.*, ensuring the uniqueness of values for an attribute) and the lack of representation for metadata. Therefore, as perspectives for future works, we plan to extend our formal

[1] https://github.com/AlexisGuyot/type_safe_compo
[2] https://github.com/milessabin/shapeless

framework to support more constraints on data, schemas, models and metadata. As implied by the Curry-Howard isomorphism [9], there is a correspondence between formulae in intuitionistic first-order logic and types (*propositions-as-types*). Therefore, we should be able to handle more logical constraints with new types.

References

1. Alotaibi, R.B.M.: Semantic Optimizations in Modern Hybrid Stores. Ph.D. thesis, University of California, San Diego (2022)
2. Dixon, J.: Pentaho, hadoop, and data lakes — james dixon's blog (2010)
3. Dybjer, P., Palmgren, E.: Intuitionistic Type Theory. In: Zalta, E.N., Nodelman, U. (eds.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, Spring 2023 edn. (2023)
4. Farid, M., Roatis, A., Ilyas, I.F., Hoffmann, H.F., Chu, X.: Clams: bringing quality to data lakes. In: International Conference on Management of Data, SIGMOD'16. pp. 2089–2092 (2016)
5. Gillet, A., Leclercq, E., Savonnet, M., Cullot, N.: Empowering big data analytics with polystore and strongly typed functional queries. In: International Database Engineering & Applications Symposium, IDEAS'20. pp. 1–10 (2020)
6. Griffioen, P.: Type inference for array programming with dimensioned vector spaces. In: Symposium on the Implementation and Application of Functional Programming Languages, IFL'15. pp. 1–12 (2015)
7. Hai, R., Quix, C.: Rewriting of plain so tgds into nested tgds. Proceedings of the VLDB Endowment **12**(11), 1526–1538 (2019)
8. Hai, R., Quix, C., Zhou, C.: Query rewriting for heterogeneous data lakes. In: Advances in Databases and Information Systems, ADBIS'18, Proceedings 22. pp. 35–49. Springer (2018)
9. Howard, W.A.: The formulae-as-types notion of construction. To HB Curry: essays on combinatory logic, lambda calculus and formalism **44**, 479–490 (1980)
10. Koupil, P., Holubová, I.: A unified representation and transformation of multi-model data using category theory. Journal of Big Data **9**(1), 61 (2022)
11. Koupil, P., Hricko, S., Holubová, I.: A universal approach for multi-model schema inference. Journal of Big Data **9**(1), 1–46 (2022)
12. Muranushi, T., Eisenberg, R.A.: Experience report: Type-checking polymorphic units for astrophysics research in haskell. ACM SIGPLAN Notices **49**(12), 31–38 (2014)
13. Scholly, E., Sawadogo, P., Liu, P., Espinosa-Oviedo, J.A., Favre, C., Loudcher, S., Darmont, J., Noûs, C.: Coining goldmedal: A new contribution to data lake generic metadata modeling. In: 23rd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP@ EDBT/ICDT 2021). vol. 2840, pp. 31–40 (2021)
14. Uotila, V., Lu, J., Gawlick, D., Liu, Z.H., Das, S., Pogossiants, G.: Multicategory: multi-model query processing meets category theory and functional programming. Proceedings of the VLDB Endowment **14**(12), 2663–2666 (2021)
15. Zaharia, M., Chambers, B.: Spark: The definitive guide. O'Reilly Media Sebastopol, CA (2018)
16. Zhao, Y., Megdiche, I., Ravat, F., Dang, V.n.: A zone-based data lake architecture for iot, small and big data. In: International Database Engineering & Applications Symposium, IDEAS'21. pp. 94–102 (2021)