What's in a (Data) Type? Meaningful Type Safety for Data Science

Riley Moher, Michael Gruninger, and Scott Sanner

Department of Mechanical and Industrial Engineering, University of Toronto, Ontario, Canada M5S 3G8

Abstract. Data science incorporates a variety of processes, concepts, techniques and domains, to transform data that is representative of real-world phenomena into meaningful insights and to inform decision-making. Data science relies on simple datatypes like strings and integers to represent complex real-world phenomena like time and geospatial regions. This reduction of semantically rich types to simplistic ones creates issues by ignoring common and significant relationships in data science including time, mereology, and provenance. Current solutions to this problem including documentation standards, provenance tracking, and knowledge model integration are opaque, lack standardization, and require manual intervention to validate. We introduce the meaningful type safety framework (MeTS) to ensure meaningful and correct data science through semantically-rich datatypes based on dependent types. Our solution encodes the assumptions and rules of common real-world concepts, such as time, geospatial regions, and populations, and automatically detects violations of these rules and assumptions. Additionally, our type system is provenance-integrated, meaning the type environment is updated with every data operation. To illustrate the effectiveness of our system, we present a case study based on real-world datasets from Statistics Canada (StatCAN). We also include a proof-of-concept implementation of our system in the Idris programming language.

Keywords: data science · dependent types · type safety · data provenance · meaningful types

1 Introduction

Data science is a delicate task involving the analysis and manipulation of hetereogenous datasets that represent real-world phenomena embedded with assumptions, rules, and interpretations. This data, however, is ultimately represented using simple datatypes like strings or integers, which fail to typify the real-world concepts the data represents. Whether comparing net and gross profit, averaging populations of overlapping regions, or summing measurements from different time periods, real-world phenomena are much more than than the simple datatypes that represent them. With data science becoming an increasingly integral part of many industries, important decisions are being made based on results produced by data scientists. These decisions can have significant consequences, the wrong decision made by a hospital administrator puts human lives

at stake, the decisions of policy-makers can have far-reaching impacts to our society; ensuring data science is correct and verifiable will have significant benefits.

To solve datatype issues, existing work supplements simple datatypes with external information in an informal, ad-hoc, and highly manual manner. These approaches reduce the nuanced real-world rules and interpretations of data to scattered informal knowledge and documentation, provenance records, and prohibitively complex knowledge models. This leads to an environment where results become extremely difficult to validate and verify, especially throughout the long and complex process that is the modern data science pipeline.

In this paper, we present the meaningful type safety (MeTS) framework, where type checking equates to validating the rules and assumptions of the real-world phenonema the data represents. Rather than use strings or floats, MeTS relies on semantically-rich types like population, time intervals, and geospatial regions. We do this through powerful, expressive, and decidable dependently-typed programming, based on axioms of formal knowledge models. Our approach, contrary to the current state, is formal, automatic, and easily verifiable.

This paper consists of a detailed presentation of the issue of datatypes and how our framework can be applied to solve it, including a case study of StatCAN census data to make our framework concrete. To provide salient motivation, we identify specific scenarios where errors of real world rules and interpretations frequently occur in data science, and identify how current approaches fail to address these scenarios. The most significant portion of this paper is the specification of MeTS, presented in a formal syntax based on Martin-Löf type theory [18], and also instantiated for the StatCAN census data with a proof-of-concept implementation in the Idris programming language. Finally, we discuss how our work fits into a broader research program including a correspondence theorem and a focus on tractability.

2 Datatypes Fail to Typify Data

This section will detail three prominent and significant classes of datatype problems including those associated with time, mereology, and data provenance. Datatype errors are a result of the fact that one simple datatype may be used to represent multiple very different real-world concepts. Both a person's age and the population of a city could be represented by an integer, despite ages and city populations having very different rules and interpretations. This leads to many different kinds of problems, from something seemingly simple like adding feet to metres, to more complex errors like geospatial mereology changing as a function of time.

2.1 Time

Time is an especially important concept, it is an ever-present factor in data which contains many implicit rules and assumptions. For example, training a machine learning model on observations that occurred later than test-set observations would be predicting the past from the future, producing a non-sensical, anti-causal model. Furthermore, the real-world interpretation of data may change as a function of time, while labels do

not. Consider comparing the price of a home in 1967 to that of a home in 2021, the relationship of purchasing power and time must be reconciled.

Time issues can be further exacerbated through the layering of time with other metadata like units or labels. Consider financial analysis on the Frankfurt stock exchange; even ignoring inflation, a quote from 1960 would be given in Marks, while one from 2004 would be given in Euros. The real-world rules and assumptions of time in a given dataset, even if well-understood, lack a formal means of integration or validation, a human still must interpret column labels, consult documentation, or manually review code. So long as manual intervention is necessary to verify and validate results, errors are bound to occur, and negative consequences are bound to follow.

2.2 Mereology

One of the most fundamental relationships that exists in data is mereology, the relationship of parts to a whole. The seminal work of Winston et al [22] and the myriad of work that built upon it, especially recent developments in the knowledge modelling community [4, 12], illustrates that mereology is a complex concept. In data science, mereology manifests itself in many ways, from categories and sub-categories to overlapping time intervals, data scientists must be aware of these relationships to preserve the integrity of interpretation. For example, consider a data scientist working with COVID-19 vaccination data: they must understand that "individuals with two vaccine doses" is a subset of "individuals who have received a single dose", and that both are part of the larger whole of "eligible individuals". Understanding these relationships is crucial to avoiding errors like double-counting and preserving the integrity of results. Furthermore, time may complicates these issues further, "eligible population" may change to encompass new age groups, booster shots and a new 3+ dose category could alter the definition of "fully vaccinated", etc.

Underlying mereological relationships are not formally modelled or integrated in data science pipelines, so catching these kinds of errors requires careful and laborious manual review. In the data science pipeline, simple datatypes reduce concepts with mereological relationships to formats that cannot capture this reality.

2.3 Provenance

Just as time alters the rules and interpretation of data, so do the operations data scientists perform in their analyses. Even assuming a complete formal understanding of a dataset, the interpretation of that dataset will throughout the data pipeline. For example, the average of city populations is no longer itself a "city population"; a new entity has been derived, with its own real-world rules and interpretation. Additionally, just like time, provenance adds a layer of complexity in conjunction with the previously mentioned issues.

Consider a relevant example of provenance issues involving physical units, whereby types consist of physical units; going beyond floats and integers to kilograms or feet. Even with this more complex representation, ensuring the integrity of the data's real-world interpretation is more than ensuring: `Unit X == Unit Y`. Take for example data consisting of individual's height measurements: if Bob and Alice's heights are

both measured in centimetres, $height_{bob} + height_{alice}$ is completely type-safe from a unit perspective, but the result of this computation has no meaningful interpretation; it is no longer a height. Alternatively, $population_{regionA} + population_{regionB}$ does have a meaningful interpretation: it makes sense to speak of "total populations", but not "total heights". Even with more robust datatypes, it is not enough to enforce only equal types for any given operation, operations each have their own preconditions, and may produce entirely new datatypes as a result.

3 Current Approaches

Data scientists utilize many tools and methodologies to address the problems of datatypes, but none of them sufficiently address datatype issues like time, mereology, or provenance. These limitations result from various factors inherent in existing methods, like the necessity of manual effort, informal representations, poor standardization, and ad-hoc applications. This section will critically examine these approaches in three categories: documentation, provenance tracking, and knowledge representation techniques.

3.1 Documentation Standards

Documentation is a tool intended to ensure quality, reduce risk, save time, and encourage knowledge sharing. However, effective utilization of documentation for data science is a challenging task. The most significant issues of using documentation to address datatype issues is their informal nature, their inability to properly address provenance, and their lack of standardization.

One of the most significant recent developments for data science documentation is Datasheets for Datasets [7], an effort to create a gold-standard for machine learning dataset documentation. These datasheets consist of answers to curated questions about a wide range of dataset information, like their motivation, collection process, pre-processing steps, and maintenance. While encouraging ML practitioners to be well-informed about their data is a positive intent, this work cannot avoid the inherent limitations of documentation. Even with a complete datasheet, there is no guarantee that the answers supplied to the set of questions will be of sufficient detail, contain sufficient contextual information, or be unambiguous. The ambiguity of natural langauge is problematic for data science, without any formal specification beyond natural language, the issues of semantic heterogeneity are unavoidable. Moreover, while the authors do provide example datasheets for well-known datasets, they do not evaluate these examples or provide any means of evaluation; there are no quality standards for datasheets.

An additional issue with documentation-based approaches is a failure to sufficiently address provenance. With the interpretation of data changing as the result of data transformations, the information contained in the documentation must change accordingly. This will require an unrealistic degree of manual effort to update the documentation and maintain effective versioning. The last major issue of documentation approaches is a lack of standardization. Even for Datasheets for Datasets, the authors acknowledge that enterprises use unique implementations of their work, like Google's Model Cards [19] or IBM's factsheets [15]. This lack of standardization places unnecessary burdens on

data scientists which hinders knowledge-sharing and interoperability. With no formal semantics, quality criteria, provenance-awareness, or standardization, documentation-based approaches cannot adequately address the datatype issues.

3.2 Provenance Tracking

Provenance tracking provides information about the origins, history and derivation of data. For an excellent survey on data provenance work, we refer the reader to [14]. While provenance tracking is mature, formal, and well-understood within the data modelling and engineering community, provenance tracking alone does not sufficiently address the nuances of datatype issues. One example of a formal representation of provenance is in Buneman et. al's Graph Model of Data and Workflow provenance [1], which models workflow provenance through directed acyclic graphs called provenance graphs. As opposed to documentation-based approaches, these approaches have formal semantics and are machine-readable. However, despite this formality, this provenance information still must be interpreted with respect to the specific real-world phenomena modelled in the data. Provenance tracking along does not integrate real-world knowledge with provenance information, and fails to automatically detect datatype errors. While provenance-based approaches are preferable to manually reviewing code, it still lacks the additional semantics necessary to enforce meaningful data science.

3.3 Knowledge Representation

Knowledge Representation is a rich discipline containing many models which represent complex rules and interpretations from the real-world in a very formal way. While there are many widely accepted and useful knowledge models, such as ontologies for units of measure [11, 21], time [13], and processes [10], current work does not sufficiently integrate and apply them for general-purpose data science. Even given an agreed-upon ontological representation of real-world concepts, there is no consensus on how to integrate this model into the data science pipeline, and no guarantee on the tractability or scalability of these methods applied to data science.

Context interchange technology (COIN) [6, 8, 17] identifies a similar problem identified in this paper, that of semantic heterogeneity between datasets, and addresses it using a general knowledge model to translate between more specific models, or "contexts". The issue with this approach is that COIN accounts only for a single data science operation: projection. The user queries some data and it is converted into their specific context for them; it does not support the interpretation of addition, multiplication, averages, etc. Furthermore, COINs contexts are specified weakly, with just two relations: `is_a` and `attribute`. COIN's shortcomings are inherent to description logic-based (DL) approaches, as even with additional properties, DL lacks the expressiveness to integrate real-world rules into various operations.

Foundational Ontologies for Units of Measure (FOUnt) [11] is a collection of ontologies that model rules for combining units of measure with respect to the physical objects and processes they describe. While these ontologies are more expressive than those of COIN, they also provide limited coverage of data science operations; only

specific instances of addition, subtraction, and division. In addition, FOUnt lacks consideration of provenance; the axioms of units do not change as specific operations are performed unless an entirely new unit is derived (like density from mass and volume). The other issue with higher-order logic-based approaches like FOUnt is complexity; reasoning in a data science application would require an enormous volume of instantiations and undecidable reasoning with a theorem prover.

Real-World Types (RWT) [23, 24] is a framework with a motivation very similar to that of COIN and this paper; it identifies a mismatch between machine representation and the real-world entities they represent. Unlike COIN and FOUnt, however, RWT is much more informal in specifying types, they are essentially annotations of OOP objects, consisting of a name, a natural language definition, possible values, and references. Furthermore, the RWT framework relies on OOP naming conventions to identify entities and generate candidates for real-world types, and requires manual review to identify errors. This approach does not rely on any formal model or logic to detect errors or enforce rules, being more akin to a low-level and object-oriented version of an approach like datasheets for datasets [7]. RWT's treatment of real-world rules as add-ons to objects speaks to the larger issue of a reliance on objects as an alternative to simple datatypes. The object-oriented-paradigm can perform many run-time checks to verify properties of data and attempt to enforce real-world rules. The specification of real-world rules themselves, however, must be formally specified in some other format, as annotations, an ontology, etc. Furthermore, reliance on object implementations to ensure the integrity of real-world rules entangles the hides the ontological commitments of programmers behind implementation decisions.

Other knowledge modelling approaches involve utilizing upper ontologies, those that model real-world concepts generally, in conjunction with conceptual modelling languages like UML. For instance, in [2], Albuquerque et al employ semantic reference spaces to ontologically ground UML datatypes. The reference structures used to specify the meaning of datatypes consist of a taxonomy for measurement dimensions like ordinal dimensions or interval dimensions. Consider the datatype for the temperature in Toronto, 10°C is an integer interval dimensions which is associated to the measurable quality Outside Temperature, of the Toronto city. Knowing that this quantity is an interval dimension, I know I can compare it to other temperatures through relations like hotterThan, and knowing it is a characterization of a place, I could limit these comparisons to other cities, to answer questions like "Is Chicago hotter than Toronto?". However, when one considers the data science application, this approach provides no semantic information to address which operations could be performed on this quantity; can I add together the temperature of Toronto and Chicago? The application of these taxonomy-centric solutions often focus on providing explanations for how quantities may be associated to concepts, but do not provide solutions for the complex issues encountered when combining and manipulating datatypes in the data science pipeline.

An additional form of knowledge modelling is conceptual modelling in the database community, in approaches like the entity relationship (ER) model [5], temporal ER models [9], and semantic data models (SDM) [20]. While these approaches provide a more formal means of understanding and documenting database schema, they, like knowledge modelling approaches in general, lack a focus on data science specifically.

This kind of conceptual modelling focuses primarily on specifying the relationship which exists between the entities represented in a database. These conceptual models, however, are not designed specifically for the data science application, where provenance becomes a significant concern. The transformative nature of data science operations, especially in high volumes, create complex semantic requirements which conceptual models lack the expressiveness to enforce. While knowledge models do have steps in the right direction in terms of formality, their lack of a data science-focused approach means they cannot adequately solve datatype issues.

4 Case Study: StatCAN Census Data

Now having a comprehensive understanding of datatype issues including how current work fails to sufficiently address them, we provide further, concrete motivation drawn from Statistics Canada (StatCAN) census data. These datasets model a variety of demographic information for various geographic areas throughout Canada, and are made publicly available [1]. We base our analysis on population data divided by geographic divisions of various levels [2] from 2011 and 2016. These datasets allow us to provide specific and concrete examples of the aforementioned datatype problem classes and can also be used to evaluate our framework. Furthermore, concepts like geospatial regions, time, and mereology are not only integral to these datasets, but are commonly represented across many diverse datasets.

4.1 Example Operations

In order to provide specific and concrete instances of how the problems of datatypes occur within data science, we exemplify datatype errors through operations a data scientist may perform on census data. These operations may be addition, subtraction, or averages, among other common operations, and are performed on population quantities defined for a specific geospatial region and timepoint. While this case study is on population data, the embedded concepts like time, geospatial regions, and mereology are widely applicable to general datasets. Each of these example computations violates real-world assumptions or rules associated with census data, and would produce a result that, while a quantity could be computed, would have no meaningful interpretation. These examples form a foundation for the problem of providing precise explanations for how and why real-world rules are violated through datatype errors.

Disjointedness The following two example computations violate real-world rules of temporal and geospatial disjointedness, respectively.

- Computing the median of non-disjoint populations (campbelton and each of its parts in bordering provinces)

[1] StatCAN's full range of Census data can be accessed at `https://www12.statcan.gc.ca/census-recensement/2021/dp-pd/index-eng.cfm`

[2] The specific datasets we referenced can be accessed by their StatCAN catalogue numbers: 98-401-X2016041, 98-401-X2016043, 98-401-X2016066

```
median(Toronto, ... Campbelton, Campbelton (NB Part), Campbelton (Quebec Part))
```

– Computing the sum of populations over disjoint time periods
```
sum(Toronto2016, Hamilton2016, ... Guelph2011)
```

Each of these errors stems from disjointedness, with the first example, adding Campbelton's population together with that of its parts, is a case of double-counting overlapping regions. The second example contains addition over populations which vary over both time *and* geospatial regions, producing a nonsensical result.

Provenance The following two examples illustrate how data science operations produce new kinds of data, forming new rules.

– Computing the average of two average populations
```
avg(avg(Toronto, Hamilton), avg(Guelph, Kitchener))
```
– Computing the average of a population change over time, and a population difference in regions
```
avg((Toronto2016 - Toronto2011), (Guelph 2011 - Hamilton2011))
```

For the first example, while an average of averages could have an interpretation, it is not the correct means of obtaining an average over the four regions (which is true not just of demographics data but for any data). With the second example, an average of differences seems like an average over the same kind of quantity. However, the key distinction is what these differences are aggregated 'over': change over time is not comparable to a difference over geospatial regions.

5 Meaningful Type Safety Framework (MeTS)

In this section, we provide a high-level overview of the framework designed to model real-world concepts and rules like those contained in StatCAN Census data and detect errors like those given in the example operations. The Meaningful Type Safety framework (MeTS) rejects simple datatypes and elevates type safety to a meaningful result, one which is derived from specific representations of real-world concepts and their associated rules and interpretations. The complete and production-ready implementation of MeTs would include several components, including a method of integration with existing data science tools, and a complete correspondence theorem, both of which fit into the broader research program of MeTS (discussed in section 8). The focus of this chapter, however, is the technical heart of MeTS, namely the type-checking mechanism. In keeping with a focused scope, we present the MeTS architecture as two primary components: the interface component, and the typing component, depicted in Figure 1.

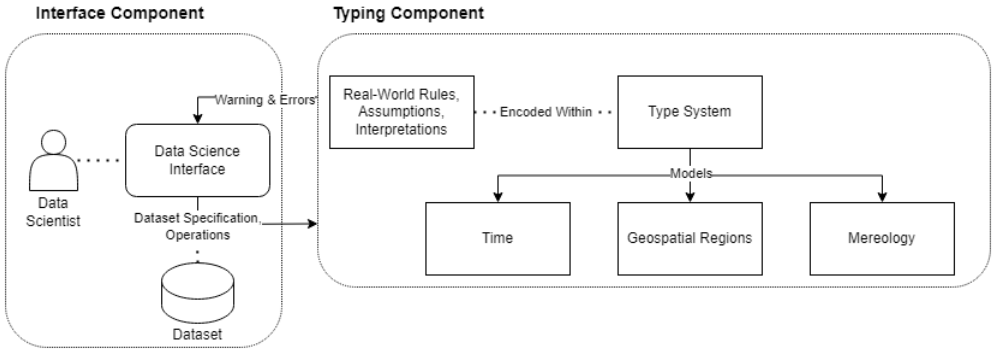

Fig. 1. Architecture for the Meaningful Type Safety Framework (MeTS)

5.1 Interface Component

The interface component is how the data scientist interacts with MeTS, by performing operations on some dataset(s) and receiving corresponding feedback from the type system. In the context of this paper, the interface component exists within our proof-of-concept implementation (further elaborated in section 6.4) using the Idris programming language [3]. However, it should be noted that *we do not expect data scientists to learn dependently-typed functional programmming*, a production-ready interface would be implemented through common data science tools like pandas, R, or Tableau for familiarity and ease of use, with type-checking still performed in a dependently-typed language. Operations that a data scientist performs would be done the same as they currently are, while real-world concepts would be specified at the beginning stage of the data pipeline. This initial specification would associate a dataset with concepts formally modeled in the type system, such as time, geospatial regions, or mereological relationships, thereby pairing the dataset with its real-world interpretation and rules. This pairing will then enable the automatic detection of real-world rule violations for any downstream operations involving the dataset. This pairing is trivial in our proof-of-concept implementation since the specification of the dataset(s) is already in Idris.

5.2 Typing Component

The typing component is the primary reasoning mechanism of the framework, based on a dependently-typed representation of the real-world rules and assumptions of the dataset(s) being modelled. The type system has the goal of implementing the kinds of real-world checks that would normally be done manually, through type-checking. This elevates the notion of type-safety from a trivial check to guaranteeing a level of interpretability. It should be noted that the real-world rules and assumptions encoded within this typing component are not arbitrary and are rooted in formal logic models, namely first order logic ontologies. The specific relationship between these ontologies and the typing system is outside the scope of this paper and encompassed in the correspondence

theorem, as described in section 8.1. The specifics of the dependently-typed program component will be presented in complete detail in section 6.

6 Type System

The type system of MeTS uses dependent type theory and expression tree representations to construct preconditions for a wide range of data science operations. For the purpose of this paper, MeTS is presented in a simplified syntax based on Martin-Lof's intuitionistic type theory [18], and basic functional programming. MeTS is presented here with specific type-checking functionality given for time, geospatial entities, and data provenance. Additionally, the type system is evaluated with respect to the previously presented example operations (with a corresponding proof-of concept implementation in the Idris programming language [3]).

6.1 Syntax

Since dependent type theory is not typically associated with data science, our presentation of the MeTS type system assumes little to no prior experience with dependent types or functional programming (for a more thorough introduction to these topics, we refer the reader to [16]).

To illustrate our function syntax, consider the basic factorial function:

```
factorial : Integer → Integer
factorial(0) = 1
factorial(n) = n*(factorial(n - 1))
```

Three syntactic constructs should be observed here: typing declarations, function notation, and pattern matching. We use the `:` operator to denote typing declarations, it can be read as "has the type". The $\rightarrow$ operator denotes a function type, a function taking one argument will have one $\rightarrow$, infixed between the input type and the output type, a function taking n arguments will have $n \rightarrow$ symbols. Lastly, the actual function definition utilizes pattern matching, where specific patterns for the arguments of the function are given as a 'blueprint' for the function to follow. In our example, the first pattern is given as a base-case, and all other possible inputs to our function can be computed recursively. It should also be noted that MeTS utilizes only total functions, that is, functions will have patterns for all possible value of the input type(s).

Dependent Type Theory The central idea of dependent types is that types may depend on values or other types. Dependent type theory has two new type constructs, the dependent function type and the dependent pair type. The dependent function type constructs a type from some parameter, the canonical example of which is vectors of length `n`, with `n` being a type parameter of the natural number type, denoted $n : \mathbb{N}$. In the MeTS framework, we utilize the dependent pair type, since it provides a method of enforcing operation preconditions analogous to real-world rules.

The basic example of a dependent pair type describes pairs of values where the type of the second element depends on the value of the first. In general, we can write a dependent pair type as:

$$\sum_{a:\mathbb{A}} B(a)$$

Firstly note that the typical meaning of the $\sum$ operator is interpreted differently when read in a typing statement, it denotes a dependent pair type here, also sometimes called a sigma type for this reason. If $(a,b) : \sum_{a:\mathbb{A}} B(a)$, then we can say $a : A$, and $b : B(a)$. That is, b's type depends on the value of a. With this construct, we can also enforce relationships within a tuple, consider the following type:

$$\sum_{m:\mathbb{N}} \sum_{n:\mathbb{N}} ((m < n) = True)$$

This type describes tuples (m,n) of natural numbers where the first element is less than the second. Values inhabiting this type would consist of a triple containing: the two elements m and n, and a proof of $m < n = True$. Given that the definition of the $<$ operator is total, this expression simply reduces to $True = True$ and a proof of the above can be done via reflection, type-checking is decidable for total functions. Types of this form are essential to our typing framework: functions with arguments of this form are analogous to preconditions for those functions.

Alternate Notations We have chosen a notation similar that of Martin-Lof type theory since it is the seminal notation, and because the it is compact and simple to follow. Other notations are sometimes used in more application-oriented work, as in many functional programming papers, and may have minor variations. Table 1 shows the dependent pair types written in some of these alternate notations, as well as how these types could be written in Idris syntax.

Table 1. Differences in Dependent Type Theory Notations

Notation	Examples
Martin-Lof Type Theory	$\sum_{a:\mathbb{A}} B(a)$ $\sum_{m:\mathbb{N}} \sum_{n:\mathbb{N}} ((m < n) = True)$
Dependently-Typed Lambda Calculus	$\sum a : \mathbb{A}.B(a)$ $\sum m : \mathbb{N}.\sum n : \mathbb{N}.((m < n) = True)$
Dependently-Typed Lambda Calculus (Alt)	$\exists a : \mathbb{A}.B(a)$ $\exists m : \mathbb{N}.\exists n : \mathbb{N}.((m < n) = True)$
Idris	`(a : A) -> (b : a -> Type) -> Type` `mn:(Nat, Nat)**((fst mn < snd mn)=True)`

6.2 Expression Trees

To properly account for the effects of provenance, MeTS incorporates this information in type-checking through an expression tree construct. Essentially, types are a combination of base types and the operations that have been performed, in a recursively-defined

tree structure:

$$\mathbb{B} ::= Population \mid Time \mid GeoEntity \mid ...$$
$$\mathbb{O} ::= Sum \mid Sub \mid Mult \mid Div \mid Avg \mid ...$$
$$\mathbb{E} ::= Atom(\mathbb{B}) \mid Over(\mathbb{O}, (List\ \mathbb{E}))$$

That is, expressions are either an atom (base type) or some operation 'over' a set of expressions. This structure allows us to define type-checking functions, preconditions, that traverse these trees, an essential component of the primary mechanism behind MeTS.

6.3 Preconditions

Preconditions are defined by typing operation functions with dependent pair types consisting of the actual operands and a proof that they satisfy some set of preconditions. The general form is given by:

`operation :` $\sum_{\texttt{operands} : (\texttt{List}\ \mathbb{E})}$ `(opPreconditions(operands)=True)` $\rightarrow$ `result:`$\mathbb{E}$

The functions that evaluate preconditions, `opPreconditions` are structured in a way that facilitates re-use and sharing of preconditions. For example, the precondition definition function for population sums is given by:

```
popSumPreconditions : List Population → Bool
popSumPreconditions(ps) = disjointRegions(ps) & allSameTime(ps) &
allMeasured(ps)
```

The preconditions for a sum over populations contains more general precondition functions defined over geospatial regions, time, and provenance, that can be referenced in other operation preconditions. These specific preconditions model the real-world assumptions of a sum of populations, that it is a count of individuals over disjoint areas, at some consistent point in time. `allMeasured` is an example of a provenance-based precondition, its truth value is based on the the structure of the expression tree. Intuitively, the statement "the total population of Toronto and Hamilton 125.3 people", is nonsensical, a total population should never result in a non-whole number. Analogously, a sum of populations should never be performed over estimated or aggregated populations. The `allMeasured` function is the formal representation these real-world rules, and acts as a method of catching a violating operation before it happens, ensuring type-safe operations are meaningful.

6.4 Implementation

For the purposes of demonstrating MeTS, we implemented the type system in the dependently-typed programming language Idris [3] [1]. Idris was chosen for its simple to read syntax, and its decidable type-checking. Idris type-checking is decidable since Idris can enforce totality of functions, and ensure that any functions involved in a type-checking operation (like precondition functions) are total, and therefore, decidable. It

[1] Our proof-of-concept implementation with all the example census computations and more is available at `https://github.com/riley-momo/Meaningful-Type-Safety-For-Data-Science`

is important to note that our framework is independent of the implementation language chosen, part of the reason why we do not present our framework in Idris syntax, but a more general dependently-typed notation.

7 Application and Evaluation on StatCAN data

In order to deliver on the original motivation for meaningful types, we demonstrate MeTS in action on the previously mentioned StatCAN data. This section references the specific type-checking mechanisms that detect violations of real-world rules including disjointedness and provenance, and how this form of type-checking can be abstracted to general problems.

7.1 Disjointedness

Disjointedness is the lack of any overlap or parthood, modelling disjointedness requires a model of mereology, which we have implemented in MeTS for time and geospatial regions, as well as defining a general mereological interface. Take the example computation, performing addition over Campbelton's population, and that of Campbelton's provincial parts. This is not a type safe operation since addition over populations assumes geospatial disjointedness, to avoid doublecounting. Correctly identifying this error requires 3 elements: modelling the knowledge that the provincial parts are parts of the whole, a definition of disjointness for geospatial entities, and a precondition for the addition operator that the geospatial regions of the operands must be disjoint from one another. As for the first element, we leverage pattern-matching in a general mereological partOf function to accomplish this:

```
partOf : GeoEntity → GeoEntity → Bool
partOf(Campbelton (Quebec Part),Campbelton) = True
partOf(Campbelton (NB Part), Campbelton) = True
...
```

For this geospatial disjointedness, we use a qualitative mereology, as would be done in a formal logic; pattern matching here is analogous to relations in the A-box. For temporal mereology however, we do not need to rely on A-box style relations of parts; it is an arithmetic computation.

```
partOf : Time → Time → Bool
partOf(Interval(x1, x2), Interval(y1, y2)) = x1 <= y2 & y1 <= x2
...
partOf(t1, t2) = t1 == t2
```

This is a very obvious example of MeTS efficiency over reasoning, as this kind of quantitative reasoning in a formal logic would either require an ontology of time and processes, or rely on SMT solvers, as opposed to our comparatively simple use of functional programming techniques. For the definition of disjointedness in general, we define it in terms of the partOf relation:

```
disjoint(x,y) = not (partOf(x, y) OR partOf(y, x))
```

Finally, we require a precondition function which ensures all the geospatial regions of a set of populations are disjoint from one another, as a pre-condition for arithmetic sum of populations.

```
disjointRegions : (List Population) → Bool
disjointRegions(ps) = disjointList (getRegions ps)

popSumPreconditions : (List Population) → Bool
popSumPreconditions(ps) = disjointRegions(ps) & ...

populationSum : (∑ps:(List Population) (popSumPreconditions(ps) = True)) → Population
```

Note that for the purpose of this paper, we omit some basic function definitions and pattern matching (they are fully implemented in our proof-of-concept implementation), since function definitions like disjointList are basic exercises in functional programming, and not relevant to our key contributions. With these definitions, the sum of populations guarantees all the geospatial regions of the populations are disjoint, and thus ensures no errors of double-counting due to geospatial overlap. Furthermore, an addition operation requiring disjointedness over one of its 'stratifiers' is not specific to populations. Summing the mass of physical objects, for example, similarly assumes they do not occupy the same space; the methodology of type-checking is the same in both cases.

7.2 Provenance

Integrating provenance with type-checking is where the power of dependent typing is most evident. Revisiting the earlier example operation, an average of averages should not be type-safe. To correctly type this, there are two important components: the way operations derive new expression trees, and how precondition functions distinguish between different expression trees. Returning the correct expression trees is straightforward, we need only ensure the right-hand side of our pattern matching expressions include the correct operator:

```
populationSum : (∑ps:(List Population) (popSumPreconditions(ps) = True)) → Population
populationSum(ps) = Over(Sum, ps)

populationAvg : (∑ps:(List Population) (popAvgPreconditions(ps) = True)) → Population
populationAvg(ps) = Over(Avg, ps)
```

The next component is incorporating patterns of expression trees into precondition functions. For example, a precondition for population trees which have had an average or median operation performed on them:

```
aggregatePop : Population → Bool
aggregatePop(Over(Avg, ps)) = True
aggregatePop(Over(Median, ps)) = True
aggregatePop(_) = False
```

Precondition functions of this form allow us to make distinctions between kinds of data based on transformations that have been applied to them. This, in conjunction with list comprehension functions, like `all`, `none`, `atLeastOne`, etc, can be incorporated into operation preconditions to give us fine-grained control over which sequences of operations produce meaningful results for the given data types.

Where the expressive power of preconditions becomes even more evident is in the combination of mereological and provenance preconditions. Consider the earlier example computation, `avg((Toronto2016 - Toronto2011), (Guelph 2011 - Hamilton2011))`.

An explanation for why this computation is not meaningful requires both a mereological explanation and a provenance-based one. While both operands are *population differences* (provenance tree head is the same operation), they are different kinds of differences (mereological distinction). The complete preconditions for population average provide insight into how this is type checked:

```
avgPreconditions : List Population → Bool
avgPreconditions(ps) = ((allSameRegions(ps)∧differingTime(ps))∨
                        (allSameTime(ps)∧(disjointRegions ps)))∧
                        (all(measuredPop, ps)∨all(scaledPop, ps)∨
                        all(differencePop, ps)∨all(ratioPop, ps))
```

In order to compute a population average, the population values we are averaging over must belong to the type described by: all possible values of `ps` such that `popAvgPreconditions(ps) = True`. In our example, the provenance portion of the preconditions (after the $\wedge$) are satisfied; both operands are population differences. However, our mereological component is not satisfied; the operands vary over both time and regions. Since we cannot generate a proof of our example operands satisfying this precondition, it is not type-safe, and thusly not a meaningful computation.

8 Research Program

While the type system of the MeTS framework is the primary mechanism for meaningful data science, its broader impact is realized when placed within the greater research program of MeTS. This research program expands upon the MeTS framework by modelling its correspondence to formal knowledge models and by extending the implementation with tractability in mind.

8.1 Correspondence Theorem

In order to model concepts like time and mereology, fundamental assumptions about their semantics must be made. MeTS is no different, because behind type-checking functions and preconditions are ontological commitments. It is therefore important to make these commitments clear and transparent; no specific interpretation should be forced upon data scientists when implementing the framework. Furthermore, the ontological commitments should not be so entangled with the type system that it becomes impossible to adjust these commitments without significant re-engineering. This modular and ontologically-agnostic approach not only increases transparency, and supports more domains and applications, it also has broader implications for the knowledge modelling community.

The correspondence theorem of the MeTS framework intends to prove the soundness and completeness of a given MeTS program with respect to some formal ontology(ies). We have already developed FOL ontologies for census data which correspond to the implicit modelling commitments in the exemplary StatsCAN type system, including a method of proving the soundness of a MeTS type environment with respect to some FOL theory. MeTS is not intended to be used as a 'black-box' set of dependently-typed programs, but as a method of leveraging knowledge model(s) through a dependent type system for data science. This modular and ontologically-agnostic approach

not only increases transparency, and supports more domains and applications, it also allows for new possibilities to be explored within the knowledge modelling community.

8.2 Tractability

With data becoming increasingly available and used in many industries, data science tools should facilitate a fast and efficient data science pipeline. Considering the expressive power of MeTS and its reliance on programming paradigms usually not associated with data science, tractability is a critical factor. One significant benefit of MeTS is the expressive power it achieves while remaining decidable. However, decidability itself does not guarantee adoption, a production-ready implementation of MeTS should also prioritize integration within existing data science infrastructure. In our proof of concept implementation, the interface component and typing component of MeTS are both specified in idris, a general-purpose dependently typed programming langauge. However, we should not require data scientists to learn dependent types and functional programming in order to employ MeTS into their existing workflows. In order for the vision of MeTS to be completely realized and to promote adoption in the data science community, MeTS should include efficient out-of-the-box integration with common data science tools like pandas, Tableau, etc.

9 Conclusion

The current state of data science relies heavily on datatypes which lack the expressiveness to accurately model the real world phenomena they represent. Simple datatypes fail to typify data because their simplistic representation cannot capture the concepts which underlie data including time, mereology, and provenance. Existing work addresses this issue by supplementing datatypes with external documentation, provenance tracking, and knowledge models in informal and ad-hoc ways that require manual effort and are prohibitively complex. Conversely, MeTS embeds the rules, assumptions, and interpretations of real-world concepts in types, elevating type safety from a trivial check to a meaningful result. The mechanism behind the MeTS type system is dependent types and expression trees, which integrate temporal, mereological, and provenance information into preconditions for fundamental data science operations. Operations involving StatCAN census data demonstrates concrete examples of how MeTS enforces real-world rules of provenance, geospatial and temporal disjointedness through type-checking. The MeTS framework is also part of a larger research program including a correspondance theorem with broader impacts for knowledge modelling and type theory, as well as a focus on tractability including integration with common data science tools. Ultimately, the MeTS framework is a significant step towards transparent, verifiable, and meaningful data science.

References

1. Acar, U.A., Buneman, P., Cheney, J., Van den Bussche, J., Kwasnikowska, N., Vansummeren, S.: A graph model of data and workflow provenance. In: TaPP (2010)

2. Albuquerque, A., Guizzardi, G.: An ontological foundation for conceptual modeling datatypes based on semantic reference spaces. In: IEEE 7th International Conference on Research Challenges in Information Science (RCIS). pp. 1–12. IEEE (2013)
3. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of functional programming **23**(5), 552–593 (2013)
4. Canavotto, I., Giordani, A.: An extensional mereology for structured entities. Erkenntnis pp. 1–31 (2020)
5. Chen, P.P.S.: The entity-relationship model—toward a unified view of data. ACM transactions on database systems (TODS) **1**(1), 9–36 (1976)
6. Firat, A.: Information integration using contextual knowledge and ontology merging. Ph.D. thesis, Massachusetts Institute of Technology (2003)
7. GEBRU, T., MORGENSTERN, J., VECCHIONE, B., VAUGHAN, J.W., WALLACH, H., DAUMÉ III, H., CRAWFORD, K.: Datasheets for datasets. arXiv preprint arXiv:1803.09010 (2018)
8. Goh, C.H.: Representing and reasoning about semantic conflicts in heterogeneous information systems. Ph.D. thesis, Massachusetts Institute of Technology (1997)
9. Gregersen, H., Jensen, C.S.: Temporal entity-relationship models-a survey. IEEE Transactions on knowledge and data engineering **11**(3), 464–497 (1999)
10. Grüninger, M.: Ontology of the process specification language. In: Handbook on ontologies, pp. 575–592. Springer (2004)
11. Grüninger, M., Aameri, B., Chui, C., Hahmann, T., Ru, Y.: Foundational ontologies for units of measure. In: FOIS. pp. 211–224 (2018)
12. Grüninger, M., Chui, C., Ru, Y., Thai, J.: A mereology for connected structures. In: Formal Ontology in Information Systems. pp. 171–185. IOS Press (2020)
13. Grüninger, M., Li, Z.: The time ontology of allen's interval algebra. In: 24th International Symposium on Temporal Representation and Reasoning (TIME 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
14. Herschel, M., Diestelkämper, R., Lahmar, H.B.: A survey on provenance: What for? what form? what from? The VLDB Journal **26**(6), 881–906 (2017)
15. Hind, M., Mehta, S., Mojsilovic, A., Nair, R., Ramamurthy, K.N., Olteanu, A., Varshney, K.R.: Increasing trust in ai services through supplier's declarations of conformity. arXiv preprint arXiv:1808.07261 **18**, 2813–2869 (2018)
16. Löh, A., McBride, C., Swierstra, W.: A tutorial implementation of a dependently typed lambda calculus. Fundamenta informaticae **102**(2), 177–207 (2010)
17. Madnick, S., Zhu, H.: Improving data quality through effective use of data semantics. Data & Knowledge Engineering **59**(2), 460–475 (2006)
18. Martin-Löf, P., Sambin, G.: Intuitionistic type theory, vol. 9. Bibliopolis Naples (1984)
19. Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I.D., Gebru, T.: Model cards for model reporting. In: Proceedings of the conference on fairness, accountability, and transparency. pp. 220–229 (2019)
20. Peckham, J., Maryanski, F.: Semantic data models. ACM Computing Surveys (CSUR) **20**(3), 153–189 (1988)
21. Rijgersberg, H., Van Assem, M., Top, J.: Ontology of units of measure and related concepts. Semantic Web **4**(1), 3–13 (2013)
22. Winston, M.E., Chaffin, R., Herrmann, D.: A taxonomy of part-whole relations. Cognitive science **11**(4), 417–444 (1987)
23. Xiang, J., Knight, J., Sullivan, K.: Real-world types and their application. In: International Conference on Computer Safety, Reliability, and Security. pp. 471–484. Springer (2014)
24. Xiang, J., Knight, J., Sullivan, K.: Is my software consistent with the real world? In: 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE). pp. 1–4. IEEE (2017)