Formal Ontology in Information Systems
R. Ferrario and W. Kuhn (Eds.)
© 2016 The authors and IOS Press.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License.
doi:10.3233/978-1-61499-660-6-313

Ontologies of Dynamical Systems and Verifiable Ontology-Based Computation: Towards a Haskell-Based Implementation of Referent Tracking

Thomas BITTNER [a,1], Jonathan BONA [b] and Werner CEUSTERS [b]
[a] *Department of Philosophy, SUNY at Buffalo*
[b] *Department of Biomedical Informatics, SUNY Buffalo*

Abstract. Migrating data from electronic healthcare records (EHR) to data repositories for biomedical research provides an opportunity for the design of extract-transfer-load procedures that not only make the repository a faithful representation of what is stated in the EHR, but also of how what is stated in the EHR (may or may not) correspond to what in reality the statements are about. This includes, for example, annotating which EHR statements are inconsistent with other statements, or which statements cannot possibly be true because of what we know about reality. While one goal of ontologies is to provide background information for determining the reliability of assertions that have been introduced without using an underlying ontology, one goal of Referent Tracking is to make explicit all the implicit assumptions that need to be taken into account to interpret given data correctly. In this paper, using Basic Formal Ontology as an example, we explore the potential of Haskell to implement software that is provably correct with respect to the semantics specified in the ontology.

Keywords. Ontology based computing, Haskell, Functional programming, Ontology

1. Introduction

We embrace a vision according to which any piece of electronic data relevant to the health of individuals – wherever and for whatever reason or purpose generated – should instantly be integrated into a constantly growing data pool. Health information systems (HIS) should therefore be semantically interoperable and permanently linked in a network with components that are aware of all relevant data available. Existing initiatives (Semantic Web, Linked Open Data, the Internet of Things) provide valuable partial solutions to this end. What is needed in addition are mechanisms to determine and represent not only (1) how assertions (for instance diagnoses) relate to reality (diseases in patients) and how changes in the pool of assertions relate to changes in reality and vice versa, but also (2) the extent to which data are incomplete and inconsistent. As an example, we examined Electronic Healthcare Records (EHR) of 570,000 patients from Western New

[1]Corresponding Author, email: bittner3@buffalo.edu

York to assess the extent to which diagnostic assertions in these records correspond to disorders in the patients [1]. This analysis uncovered many ways in which the data fail to represent explicitly what they are supposed to represent. In [2] a method based on design patterns following the principles of Ontological Realism [3] and Referent Tracking (RT) [4] was tested to translate a clinical research dataset into a set of Referent Tracking Tuples. The patterns were created to arrive at a collection of assertions in which not only the portion of reality described by the dataset and the dataset itself were represented in a way that mimics the structure of reality, but so also the relations between components of this dataset on the one hand and the corresponding portions of reality on the other hand. Although it was demonstrated that RT could handle all idiosyncrasies encountered in the dataset, it required a thorough understanding of and expertise in applying the underlying theories of the ontology so as not to make any mistakes. The work presented here builds further on these efforts by introducing a method that attempts to minimize the risk for committing mistakes by writing software for processing data in ways that are provably correct with respect to the semantics specified in the ontology, using the representational units of the Basic Formal Ontology as an example.

1.1. Ontology and Dynamic Systems

Most top-level ontologies (including BFO [5] and DOLCE [6]) distinguish two classes of entities according to how they evolve over time. Continuants exist in full at every moment they exist at all. By contrast, occurrents never exist in full at any point in time. At every moment in time only a single temporal stage of a given occurrent can be present. Continuants and occurrents form disjoint categories whose instances are related by formal (participation) relations. Continuants fall into two categories: ontologically independent and dependent continuants. Dependent continuants can exist only at times at which they inhere in some independent continuants and include qualities, roles, and dispositions.

At every moment in time a continuant is in a particular state – its state at that time. The state at a time is determined by the continuant's mereological make up at that time in conjunction with the collection of everything that inheres in it at that time. For an independent continuant to exist at a time means to be in its current state at that time. Independent continuants change over time by being at different times in, f.i., different states, by having different parts and/or by having different entities inhering in them.

To simplify the presentation we will focus on non-mereological aspects of states of independent continuants. From this simplified perspective, particular states are constituted by collections of particular qualities/roles/dispositions that jointly inhere in a particular independent continuant at given points in time. For example, the state of a particle in Newtonian mechanics is determined by its location in some frame of reference and its velocity (the rate of change of location). The *state spaces* are associated with universals of independent continuants. A state space is the class of possible states an instance of that universal can be in at a given time. States and state spaces are manifestations of the fact that only very specific combinations of quality universals, role universals and disposition universals can jointly be exemplified in a given independent continuant at a given time. Since state particulars are collections of particular qualities/roles/dispositions, it follows that state particulars are dependent continuants. State types are instantiated by state particulars. Examples of state types from physics include the states of (relative) rest, inertial movement, accelerated movement, etc.

Processes are occurrents which take the continuants that participate in them from one state to another state by affecting the mereological structure of the participating independent continuants and/or affecting the qualities/roles/dispositions inhering in them. Special classes include processes that create continuants in certain states, destroy continuants or leave the states of the participating continuants unchanged (e.g., [7,8]).

1.2. Biomedical Ontology and Dynamical Systems

In addition to logical and metaphysical constraints at the top level there are domain-specific constraints on possible states of independent continuants and the processes that can transform between states. For example, in classical mechanics, the only admissible processes are those that transform states in ways that conserve the total energy of closed systems.

As in physics, domain specific constraints restrict what is (onto-)logically possible to what is physically/chemically/biologically/... possible. That is, domain-specific constraints on possible states and possible sequences of states find their manifestations in the laws of physics/chemistry/biology (...laws of nature). Domain-specific constraints on sequences of states naturally extend to constraints on processes and their compositional structure. In addition to constraints arising from laws of nature there may be constraints imposed by human conventions: there may be specific conventions on the sequences of certain treatments in medicine, etc.

1.3. Referent Tracking

While most ontologies and ontology based applications focus on logical interrelations between universals, ontology also plays an important role in the processing of instance data (e.g., [4,3]). To illustrate this we use the example of referent tracking (RT) in bio-medicine [4]. Roughly, while an ontology of a dynamic systems specifies the space of ontological *possibilities*, a RT system records *actual* states, sequences of *actual* states, and associated processes within the space of ontological possibilities. In the specific context of bio-medicine the goal of RT is to create an ever-growing pool of data relating to bio-medical particulars, their states at given times (states of pregnancy, states of elevated blood pressure) and the changes they undergo over time. Within a RT system, all such entities and their states are referred to directly and explicitly. In addition, a referent tracking system can be set up in such a way that it ensures that all particular entities and their states that are referenced are at the same time classified with respect to the BFO-based system of ontologies [4].

1.4. Goals of the Work Presented

It follows from the above that it ought to be part of the domain ontology underlying a dynamical system to explicate necessary and sufficient conditions that single out the physically/chemically/biologically/medically/... possible states, possible sequences of states, and associated processes. In the context of referent tracking it will in addition be necessary to implement decision procedures that compute whether or not such conditions are satisfied for a given state, sequence, and process. Unless these requirements are met it is impossible to import data into a referent tracking system in a provably correct way, nor is it possible to maintain such data in a way that only permits changes that are verifiably consistent with previous states and the physical, chemical, biological laws as well as medical regulations and other rules that constrain admissible changes of states.

The specification of such necessary and sufficient conditions in many cases requires highly expressive languages which in general lack general-purpose automatic decision procedures that are provably correct. Our aim is to show how the functional language Haskell can be used to integrate an ontology that is strong enough to describe medical reality as a dynamical system and, in addition, to implement computation (decision procedures) as *executable* formal specifications [9] in ways that are provably correct with respect to the semantics specified in the underlying ontology.

2. Methodology

2.1. Ontology Based Computation

A computation adheres to an ontology if (I) the computation accepts as input only data items that are structured with respect to a given ontology; (II) the computation outputs only data items that are structured with respect to a given ontology; and (III) the transition between input and output states that are computable by a program coincide with the possible sequences of changes portions of reality can undergo according to the ontology.

Logically, a collection of data faithful to reality is a set of ground propositions (propositions formed by predicate and constant symbols) all of which are true of a given portion of reality. Such a set of propositions is structured with respect to a given ontology if and only if on the intended interpretation specified in the underlying ontology: (1) the sorts of the underlying formal language correspond to the basic categorical distinctions of the ontology (e.g., particulars vs. universals); (2) unary predicate symbols pick out sub-categories according to the classification of the underlying ontology, (3) n-ary predicates refer to formal relations recognized by the ontology; (4) as a set of propositions, the data is logically consistent with the axioms of the ontology.

We use the term of *pure ontology-based computing* to refer to computation that

(a) is provably correct with respect to some formal specification[2] correctness can be verified using type inference in conjunction with relatively simple inductive proofs over recursive data structures;
(b) adheres to a given ontology in the sense of points (I – III) above if and only if it passes the type check, i.e., a program adheres to an ontology if and only if it is syntactically and type-theoretic well-formed in a way that can be verified formally in a computationally efficient way before the program runs for the first time.

These points will be justified and discussed in more detail in Sec. 2.2.

For practical purposes it will occasionally be necessary to include non-pure code that does not meet the strong requirements in (a) and (b). For example, it may be necessary to access resources that are not part of the given program, e.g., to use the input or output functions of the underlying operating system, to read from a database, to input globally

[2]Formal specifications are formal techniques to describe a computation intended to aid the design and implementation of algorithms which realize that computation by verifying key properties of interest through rigorous reasoning tools whereby [10]

unique identifiers, or to request user input, etc. To accommodate such needs we use the term *verifiable ontology-based computation* to refer to computation based on code with the following properties:

(i) the code that processes the data within the RT system is pure and satisfies (a) and (b);
(ii) the non-pure code (input, output, user interface) is clearly syntactically and semantically separated from the pure code and can interact with the pure code only via well-defined interfaces. The well-formedness and the semantic coherence of the code returned by the non-pure computation can be verified via type inference

2.2. Haskell

In what follows the functional language Haskell [11] is used to realize verifiable ontology-based computing. Haskell is chosen because it offers the following features: (1) It features functions as first-order primitives; (2) It has a strong static typing system; (3) It supports explicit data flow by having immutable data structures and featuring lazy evaluation; (4) It supports syntactic and semantic features that allow to separate 'pure' from 'impure' code in a logically well-defined manner.

Functions in Haskell Are First-class Primitives: they may be passed as arguments to and returned as results of other functions, they may form components of composite data structures and may be lists of functions. Functions may be stored in records, etc.

The ontological importance of the category of states in conjunction with the role of processes in transforming independent continuants from one state to another state was discussed above. Logically, states are functional relations that relate independent continuants and times to the collections of qualities/roles dispositions that inhere in them at that time:

$$\begin{array}{l} State : IC \times T \rightarrow (DC_1 \times \ldots \times DC_n) \text{ such that} \\ (c,t) \mapsto (x_1 \ldots x_n) \text{ iff } (inheres\, x_1\; c\; t)\ \&\ \ldots\ \&\ (inheres\, x_n\; c\; t) \end{array}$$

That is, the particular state of the independent continuant $c \in IC$ at time $t \in T$ which is constituted by the particular dependent continuants $x_1 \ldots x_n \in (DC_1 \times \ldots \times DC_n)$ is represented by the map $(c,t) \mapsto (x_1 \ldots x_n)$. The function $State : IC \times T \rightarrow (DC_1 \times \ldots \times DC_n)$ as a whole is a natural representation of the state universal instantiated by the particular states $(c,t) \mapsto (x_1 \ldots x_n)$. Similarly, processes are naturally represented by functions of signature $P : State_1 \rightarrow State_2$ mapping states that are instances of the state universal $State_1 : IC \times T \rightarrow (DC_1 \times \ldots \times DC_n)$ (a function) to states that are instances of the state universal $State_2 : IC \times T \rightarrow (DC_1 \times \ldots \times DC_k)$. Sequences of states are functions of signature $SQ : T \rightarrow State$.

Since many of the ontologically significant features of a dynamic system are naturally represented as functions, a language which is designed around the computation of functions has many advantages for encoding ontology-based computation.

A Strong Static Typing System means that the type inference ensures that there cannot be any run-time errors caused by type errors once a program has passed the type checking. It follows that to the degree to which one is able to encode ontological constraints and computation in a way that is amendable to type inference, passing the type check at compile time ensures the ontological and computational correctness of the program at run time. This is particularly important in domains in which errors in information processing have particular serious consequences. This will be discussed in more detail in Section 3.

Explicit Data Flow and Lazy Evaluation ensure that the value of an expression depends only on its free variables. In contrast to imperative languages free variables are bound to expressions and not assigned values. Value assignment can be changed. By contrast variables can be bound to expressions only once. From this it follows that the result of a function only depends on its arguments. In conjunction with Lazy evaluation (roughly, parameters of functions are only evaluated as needed) this makes such programs especially easy to reason about by standard equational reasoning (the kind of reasoning we all are taught in high school mathematics) [12].

Induction and Recursion An important aspect of functional languages like Haskell is that most computations involve inductive types – types all of whose values can be constructed by mathematical induction such as the natural numbers, lists, trees, etc. For these kinds of types computation can be realized by recursive functions. Since inductive types are constructed by mathematical induction, the correctness of computation by recursion is proved by proofs that essentially use the inductive definitions of the underlying types. That is, in functional languages the principles of induction and recursion coincide [13].

Since recursion is the main computational technique, a terminating pure Haskell program counts as an inductive proof of a theorem. That is, testing a program often counts as a proof of computational aspects of a program that cannot directly be checked by type inference. Clearly, this feature makes testing much more powerful and a possible alternative to more general but manual proofs. The combination of automatic and semi-automatic, and manual techniques of proof maximize the expressive power of the language by maintaining the requirement of formal verification.

Monads In functional programming, a monad is a structure that abstractly represents and encapsulates computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of certain types together [14]. This allows for the formal representation of pipelines that process data in steps, in which each processing step has associated processing rules provided by the monad. Intuitively, monads can be compared to assembly lines, where a conveyor belt transports data between functional units that transform it one step at a time [15,16]. Monads are thus the formal means to implement generic sequential types such as

First do A then do B (possibly using the output of A),
then do C (possibly using the outputs of A and B)

where A, B, and C are (specifications of) computations in ways that the type checker can determine whether the given sequence of computations is of a certain type.

This capability of abstractly describing computations provides means to separate 'pure' from 'impure' code in a logically well-defined manner by separating the specification of sequences of actions from their execution. Consider the `IO` monad in Haskell [11]: A value of type `(IO t)` is an action/computation that, when performed, may do some input/output before delivering a result of type `t`. Actions of type `(IO t)` are collected and, if executed, executed in the specified sequence after all the pure computations have been performed. Explicit data flow ensures that the order of all pure computations is irrelevant [12] and thus the order of the execution of a program with impure components only depends on the order of the sequence of its impure components. The impure code then is executed in the order specified in the IO monad after all the pure computations have been completed. Since all the pure computation is terminated before the impure computations begin the impure computations cannot affect the pure computations and thus, all the proofs about the correctness of the pure components remain valid.

2.3. Haskell and Referent Tracking

A language with a strong static type system such as Haskell provides means to implement computation in a way that the code compiles only if it adheres to the underlying ontology. The ontology underlying the RT systems studied here is the Basic Formal Ontology (BFO) [5] extended by the notion of a state as an additional type of 'dependent continuant' – which is naturally represented as a function – in conjunction with the explicit introduction of a unique identifier for every particular and every universal.

A basic requirement of the referent tracking paradigm is that identifiers are not only unique but also immutable [4]. The explicit data flow in Haskell guarantees immutability. Since variables are bound and cannot be changed, the initial state as well as all intermediate states are preserved in the system as long as only pure computation is performed.

The explicit nature of the data flow of a RT system implemented in Haskell ensures the correctness of the tracking in the following sense: the way in which the current state of a system arises from some initial state via a finite sequence of changes, can be verified formally. This is because the logical and ontological correctness of every possible state change is verified at the time of compilation by the type system in conjunction with (possibly manual) inductive proofs. Thus, the current and every previous state of the system must be logically and ontologically correct if the initial state of the system is logically and ontologically correct. At this point it is an open question of how the provably correct state of the complete history of a RT system can be maintained through the impure computation that is required to store and retrieve data from external media. There are however cryptographic means (checksums, etc.) that may be suitable for guaranteeing that states cannot be manipulated outside the pure part of the system.

3. Results

The following shows how Haskell is used to ensure that the code of the RT system compiles only if adheres to the underlying ontology. To understand strong type systems and the role of types in ontology based computing in Haskell one needs to understand the notions of concrete types, type constructors, type class declarations, type class instance declarations and qualified types. Discussing language specific features in detail goes beyond the scope of this paper and has been covered extensively in the literature e.g., [17,13,18,15,14,19]. In applied ontology those notions have been discussed extensively in the pioneering work of Frank, Kuhn and others in Geographic Information Science, e.g., [20,21,22].

3.1. Ontological Categories and Unique Identifiers

At the heart of a RT systems are identifiers (UIs) which uniquely identify and re-identify entities and thereby allowing them to be tracked over time. For convenience suppose that all UIs are integers. To distinguish different kinds of UIs new type constants are introduced. Following the RT paradigm the type `UUI` collects all the UIs for universals. Similarly the type `IUI` collects all UIs for particulars. According to the RT paradigm not only particulars and universals get assigned unique UIs but also facts about the world that involve those entities. Consider the fact that *particular x instantiates universal y at time t*. If this fact is recorded by the system then the recorded fact receives a specific UI which is of type `INST_UI`. Similarly for facts about other relations postulated in the underlying top-level ontology (e.g., BFO in conjunction with the OBO relation ontology).

To specify these ideas formally a type (constant) for every kind of UI is introduced in conjunction with the data constructors which create particular typed UIs. To keep the notation simple the same names are used for the type and the data / value constructors. The context will always be sufficient to disambiguate.

```
data UUI = UUI Int        data INST_UI = INST_UI Int
data IUI = IUI Int        ...
```

One of the fundamental properties that is shared by all UIs is that they are kinds of things that can/need to be identified and distinguished. This is made explicit by declaring type class instances of the `Eq` type class for the various UI types and telling the systems what it means for two `UUIs`, `IUIs`, etc to be equal, e.g.,

```
instance Eq UUI where UUI x == UUI y = x == y
instance Eq IUI IUI x == IUI y = x == y     ...
```

An important aspect of declaring instances of the `Eq` type class in this way is that expressions like `UUI x == IUI y` will not be evaluated as `False` but will not even compile because expressions that test the identity of non-comparable things are semantically not well-formed, i.e., *meaningless* rather than false. This is the first example of how qualified types are used to enable the type engine to reject expressions that are semantically ill-formed according to the underlying ontology. BFO postulates a number of categories such as Continuant, Occurrent, etc. For every BFO category an algebraic type consisting of a pair of type and data constructors is introduced:

```
data CAT_Continuant a = CAT_Continuant a
data CAT_Occurrent a = CAT_Occurrent a
data CAT_I_Continuant a = CAT_I_Continuant a
data CAT_D_Continuant a = CAT_D_Continuant a
```

In RT systems the ontological distinction between particulars and universals needs to be made explicit at the type level. That is, for a BFO category such as *continuant* there needs to be a type for particulars that are continuants and a type for universals all of whose instances are continuants. This is because particulars are tracked via Instance Unique Identifiers (IUIs) and universals are tracked via Universal Unique Identifiers (UUIs). The type variables in the declaration of the type constructors for the BFO categories will allow the distinction between particulars of a given category identified by IUIs and universals all of whose instances belong to a given category and which are identified by UUIs. The system is then capable of distinguishing at the type level UIs for continuant particulars (`CAT_Continuant IUI`), UIs for universals all of whose instances are continuant particulars (`CAT_Continuant UUI`), UIs for occurrent particulars (`CAT_Occurrant IUI`), and so on for all BFO categories.

There are operations that are meaningful for all types with unary type constructors which type parameter ranges over type class instances of `UI`. They are collected in the type class `CAT`. An example is a function of type `(UI a) => (t a) -> a` that maps types with unary type constructors to the type of the constructors type parameter.

```
class CAT t where getUI :: (UI a) => (t a) -> a
```

Type classes in this sense are like algebraic theories and can be used to specify the signatures of algebraic structures such as semi-groups, Abelian groups, etc. (e.g., [22]).

Type class instants are declared for all types for which this operation is meaningful

```
instance CAT CAT_Continuant where getUI (CAT_Continuant a) = a
instance CAT CAT_Occurrant where getUI (CAT_ Occurrant a) = a
```

For all implementations of `getUI` one may require that the axiom `(t a) = (t (getUI (t a)))` holds. This must be proven to hold for every specific type class instance declaration. For finite types this can be done by a program that enumerates all possibilities. For infinite types this must be done manually via proofs by induction. When using type classes to specify a multiplicative group one would require that for the unit element `1` of the group the axiom `x*1=1*x=x` holds for all *x* and one would prove this for all type class instances.

An important aspect of the BFO ontology is that the categories are partially ordered by the isA relation. The type class `SubCat` collects operations that are meaningful for formally characterizing the hierarchically ordered BFO categories as a partial order.

```
class (CAT s, CAT t) => SubCat s t where
     subCat :: (UI a) => s a -> t a -> Bool
     subCat _ _ = True
```

The type class declaration introduces a new constrained type `s a -> t a -> Bool`. This type is constrained in three ways: firstly, whatever types are bound to the type variables `s` and `t` must also be instances of the type class `CAT`. Secondly, the types `t` and `s` are type constructors rather than concrete types: they expect a type parameter to become concrete types. The type parameter `a` is constrained to type class instances of the type class `UI`. In addition it is required to be the type parameter of both, the type constructors `s` (e.g., `(s UUI)`) and `t` (e.g., `(t UUI)`).

Instance declarations for the `SubCat` type class then tell the system for which pairs of type constructors of the `CAT` type family the `subCat` relation is defined:

```
instance SubCat CAT_Continuant CAT_Entity
instance SubCat CAT_Occurrant CAT_Entity
instance SubCat CAT_Process CAT_Occurrant
```

To explicitly enumerate the `isA` hierarchy in this way may seem ineffective but remember that this code can be generated automatically from a fully classified OWL-DL ontology.

Given those declarations, expressions such as '(subCat (CAT_Continuant (UUI 2))', '(CAT_Entity (UUI 1))' will evaluate to `True`. By contrast, expressions such as '(subCat (CAT_Entity (UUI 1))', '(CAT_Continuant (UUI 2)))', '(subCat (CAT_Occurrent (UUI 3)) (CAT_Continuant (UUI 2)))', or '(subCat (CAT_Continuant (IUI 2)) (CAT_Entity (UUI 1))' will not evaluate to `False` but cause a type error during the 'compilation' phase. That is, the type system will not accept expressions that do not adhere to the `isA` hierarchy of the underlying ontology. It rejects expressions that are semantically ill-formed with respect to the underlying ontology.

Types classes which restrict type variables to particulars and universals can be defined as follows. A particular is a type class which has a type constructor as its type parameter `t`. This parameter is restricted to types that are instances of the type class `CAT`. In addition, in all functions collected in this type class the type parameter is always bound to the type `IUI`, i.e., `(t IUI)`. This expresses on the type level that every particular belongs to a BFO category and in addition has an IUI. The type class for universals is very similar: every universal belongs to one BFO category and in addition has a UUI.

```
class (CAT t) => Particular t where
    isParticular :: t IUI -> Bool
    isParticular _ = True
    getIUI :: (t IUI) -> IUI
    getIUI  = getUI
```

Declarations for type class instances are then required for every particular type for which there are particulars/universals according to the underlying ontology.

```
instance Particular CAT_Continuant
instance Particular CAT_Occurrent
```

Again, it is impossible to apply the functions `isParticular` and `getIUI` to anything that is not explicitly declared as an instance of the type class of `Particular`. Similarly for the `Universal` type class.

3.2. Foundational Relations

Besides categories such as particulars, universals, continuants, occurrents, etc. top-level ontologies also introduce foundational relations that relate particulars and/or universals of the same or a different category. There is the *parthood* relation between particulars, the *instantiation* relation between particulars and universals, the *participation* relation between continuants and occurrents, and the *inherence* relation between dependent and independent continuants. To study how to integrate ontological constraints that characterize formal relations into a Haskell program consider the instantiation relation. The declaration of the type of an instantiation relation consists of the type and data constructors:

```
data InstOf t = InstOf INST_UI (t IUI) (t UUI) Time.
```

The type constructor `InstOf` has as type parameter the unary constructor function `t`. Its argument is restricted to the type `(t IUI)` for instantiating entities and to `(t UUI)` for instantiated entities. This declaration ensures that particular members of the type `(InstOf CAT_Continuant)` have values such as

```
InstOf (INST_UI 12) (CAT_70F (IUI 16)) (CAT_70F  (UUI 11))        (1)
       (Time 2015 05 14 9 00 00)).
InstOf (INST_UI 12) (CAT_D_Continuant (IUI 16))                   (2)
       (CAT_D_Continuant (UUI 11)) (Time 2015 05 14 9 00 00)).
```

The type Time is assumed to be declared as something like: `data Time = Time Year Month Day Hour Min Sec`, etc. Clearly, both statements are logically correct but from an ontological perspective only the first one is well-formed. Statement (1) expresses the state of affairs that the quality particular labeled `(IUI 16)` is an instance of the quality universal `CAT_70F` (a quality determinate [23] labeled `(UUI 11)`) at the given time. `(CAT_70F (UUI 11))` in turn is a sub-universal of the quality universal `(CAT_Temperature, (UUI 12))` which follows from a declarations of the form

```
instance SubCat CAT_70F CAT_Temperature
instance SubCat CAT_Temperature CAT_Quality
instance SubCat CAT_Quality CAT_D_Continuant
```

where the quality determinate `CAT_70F` is a subcategory of the quality determinable [23] `CAT_Temperature`. By contrast, statement (2) is ontologically ill-formed. The instantiating category is too general and ought to be inferred and not be stated explicitly. That is, by declaring a determinate quality particular `(CAT_70F (IUI 16))` to be an instance of a universal/category that is much further up in the classification hierarchy than the associated quality determinable `CAT_Temperature`. Statement (2) thereby violates the principle that explicit instantiation of qualities should mirror the interrelations between determinate and determinable quality universals [23].

At the type level it is possible to restrict the explicit instantiation of universals by declaring a type class `Instantiable` of which type class instances are declared only for universals that are leafs or near the bottom of the classification hierarchy of the underlying ontology:

```
class (Universal t) => Instantiable t where
     isInstantiable::  -> (t UUI) -> Bool
     isInstantiable _ = True

instance Instantiable CAT_70F
```

The type constructor for the `InstOf` data type then could be restricted to categories for which type class instances for the type class `Instantiable` exist:

```
data (Instantiable t) => InstOf t = InstOf INST_UI (t IUI) (t UUI) Time
```

If no type class instance declaration of the form `instance Instantiable CAT_D_Continuant` exists the type system will reject statement (2).

3.3. Disclaimer

The code presented above is meant to illustrate how certain language features of Haskell support ontology based computing by enabling the type system 'understand' and to enforce the commitments of an underlying formal ontology. Here we focused on Haskell's type class system. Space limitations prohibit the discussion of how more specific language features such as Monads can be used (a) to express ontological constraints on possible states and possible sequences of states in ways that can be enforced on the level of types and (b) to specify impure computations.

4. Discussion

Ontologies are usually presented as axiomatic theories in first order predicate logic or a less expressive description logic. In such a logic-based environment tasks of determining whether certain necessary and sufficient conditions are satisfied are usually realized as deductive proofs. Much of the success of computational ontologies is rooted in the fact that proofs in such systems are performed automatically. But there is a trade off between expressive power of a formal language and the computational complexity of associated automatic proof procedures [24]: the stronger the expressive power of a formal language the more computationally expensive are the reasoning algorithms.

4.1. Dynamic Systems, Situation Calculus and Planning in Artificial Intelligence

Logic and deduction-based methods to automated reasoning about dynamic systems has a long history in Artificial Intelligence, e.g., [25,26,27]. Unfortunately, reasoning problems in AI tend to be highly complex due to the aim of modeling intelligent agents and the complexity of the planning problems those agents encounter. In addition, situation calculus and many propositional approaches to reasoning about dynamic systems are subject to the frame problem (e.g., [28]). By contrast, the class of problems addressed in ontology-based programming in the particular context of Referent *Tracking* is much simpler because (a) RT is not a planning problem and (b) RT is highly constrained in that it is subject to not only constraints of the physical world but also of medical rules and guidelines, etc. Finally, describing dynamic systems in a functional language such as Haskell seems to avoid some of the problems related to the frame problem due to (i) the fact that functions are first order primitives (e.g., [22]), (ii) the availability of static type-level reasoning, and (iii) the fact that general purpose deduction is replaced by formally specified problem-specific computation.

4.2. Problems with OWL-DL

The ‘sweet-spot’ in the trade-off between the expressive power and computational efficiency of a logic-based language is widely believed to be captured in the ontology language OWL-DL [29] in conjunction with automated theorem provers such as Racer, Fact, Pellet, etc. [30,31,32] and associated RDF triple stores for the storage and manipulation of instance data [29]. These tools support the automatic derivation of the hierarchy of ontological categories specified in OWL-DL [29] (historically T-box reasoning [33]) in conjunction with the manipulation of data about particulars that instantiate those categories. Such instance data is stored in RDF triple stores [29] which can be manipulated by the underlying reasoning engine (historically A-box reasoning [33]). This includes consistency checks, the computation of transitive and compositional closures of relations, etc.

In the context of ontological computing is is natural to refer to the manipulation of a RDF triple store using a deduction-based automatic reasoner and an OWL-DL-based ontology as the prototypical example of *ontology-based computing as automated deduction*. Within this framework every ontologically possible state and state change must be derivable from some initial state by means of automated deduction. The advantage of this computing paradigm is that the resulting computation is provably correct and adheres to the underlying ontology.

The trade-off between the expressive power of a formal language and the computational efficiency of the associated formal reasoning now entails constraints on the complexity of the ontology and by extension the representations of possible state and/or constraints on the computations that can be performed automatically by means of deduction. The more expressive the formal language that is required to specify a formal ontology, the less likely it is that general computation can be formulated and executed as automated deduction in a decideable and efficently computable way. In particular *ontology-based computing as automated deduction* as it is realized in current OWL-DL-based representation and reasoning systems is rather limited in the degree to which the logical properties of formal relations and the logical constraints on interrelations between them can be specified formally and exploited in the automatic reasoning (e.g., [34]). In fact, many current ontologies that support ontology-based computing as automated deduction, as defined above, can specify many of the logical properties of formal relations only informally in the form of annotations (e.g., in [5]). Informally specified logical properties, of course, cannot be incorporated in the automatic reasoning process. In particular this means that in an OWL-DL-based framework it is difficult to rigorously and precisely specify constrains on possible states of portions of reality and the laws that constrain possible sequences of states in conjunction with the processes through which those changes are manifested. Although there is an active research community permanently pushing the envelope of what can be derived in OWL-DL-like representation and reasoning systems, there remains the need for computation tasks that involve reasoning about ontologically possible states, ontologically admissible sequences of states, in conjunction with the processes that realize those changes.

4.3. Issues with Imperative Programming Languages

Frequently, imperative programming languages ranging from C and C++ to Python and Perl are used as implementation tools for such more complex computational tasks. Unfortunately, it is very difficult to formally verify such programs and to prove that the reasoning/computation that is implemented in such programs is sound, complete, and adheres to a specific ontology [35,10]. Despite extensive testing, programs written in imperative programming languages remain prone (1) to logical errors in the sense that they perform incorrect reasoning/computations and (2) to semantic errors in the sense that they perform reasoning/computations that does not adhere to the ontology specifying the underlying semantics.

5. Conclusions

The goal of *verifiable ontology-based computing* is to use programming languages with explicit data flow and a strong static type system to ensure that a program passes a type checker only if it is logically and ontologically correct. First results indicate that in the context of referent tracking systems it is possible to achieve this goal by using the type classes of the functional language Haskell.

However, it is important to see that verifiable ontology-based computing is not a replacement of OWL-DL based ontologies and computation by deduction. Both paradigms complement each other. Our results indicate that the code whose purpose is to make the type system of Haskell 'understand' the classificatory aspects of an ontology is derivable automatically from a fully classified OWL-DL ontology. Verifiable ontology based computing is about integrating ontologies into programs and ensuring that programs adhere to an ontology. It is not about developing ontologies. However the function-based nature of Haskell in conjunction with other higher order features indicate that ontology-based computing in Haskell may open up possibilities to integrate into programs ontologies that can only be expressed in full first or higher order languages.

Acknowledgement: This work was supported in part by Clinical and Translational Science Award NIH 1 UL1 TR001412-01 from the National Institutes of Health and by SUNY Research Foundation award 689020 'Planning Grant to produce a road map to the creation of the SUNY-wide Centralized "big-data" repository (CIDR) of SUNY electronic health record data'.

References

[1] J. Bona and W. Ceusters. Replacing EHR structured data with explicit representations. In *International Conference on Biomedical Ontologies, ICBO 2015*, pages 85–86, 2015.
[2] W. Ceusters, C. Y. Hsu, and B. Smith. Clinical Data Wrangling using Ontological Realism and Referent Tracking. In *International Conference on Biomedical Ontologies, ICBO 2014, Houston, Texas, Oct 6-9, CEUR Workshop Proceedings 2014;1237:27–32.*, 2014.
[3] B. Smith and W. Ceusters. Ontological realism: A methodology for coordinated evolution of scientific ontologies. *Applied ontology*, 5(3–4):139–188, 2010.
[4] W. Ceusters and B. Smith. Strategies for referent tracking in electronic health records. *Journal of Biomedical Informatics*, 39(3):362–378, 2006. Biomedical Ontologies.
[5] B. Smith. Basic Formal Ontology: `https://github.com/BFO-ontology/BFO`, 2005.
[6] A. Gangemi, N. Guarino, C. Masolo, and A. Oltramari. Sweetening WORDNET with DOLCE. *AI Mag.*, 24(3):13–24, September 2003.
[7] B. Smith and P. Grenon. The Cornucopia of Formal-Ontological Relations. *Dialectica*, 58(3):279–296, 2004.
[8] P. Grenon and B. Smith. SNAP and SPAN: Towards Dynamic Spatial Ontology. *Spatial Cognition and Computation*, 4(1):69–103, 2004.
[9] G. Silver, O.-H. Hassan, and J. Miller. From domain ontologies to modeling ontologies to executable simulation models. In *Simulation Conference, 2007 Winter*, pages 1108–1117, 2007.
[10] B. Liskov and J. Guttag. Writing Formal Specifications. In B. Liskov and J. Guttag, editors, *Abstraction and Specification in Program Development*. 1986.
[11] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2008.
[12] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32:98–107, 1984.
[13] S. Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
[14] P. Wadler. How to Declare an Imperative. *ACM Comput. Surv.*, 29(3):240–263, September 1997.
[15] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.
[16] Wikipedia. Monad (functional programming) — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 14-September-2015].
[17] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2 edition, 1999.

[18] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA '93: Conference on Functional Programming and Computer Architecture, Copenhagen, Denmark*, pages 52–61, New York, N.Y., 1993. ACM Press.

[19] S. L. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.

[20] A. Frank. Specifying Open GIS with Functional Languages. In M. Egenhofer and J. Herring, editors, *SSD'95 Proceedings*. Springer-Verlag, 1995.

[21] W. Kuhn. Ontologies in support of activities in geographical space. *International Journal of Geographical Information Science*, 15:613–631, 2001.

[22] A. Frank and D. Medak. Executable Axiomatic Specification Using Functional Language - Case study: Base Ontology for a Spatio-Temporal Database. Technical report, Department of Geoinformation, 1997.

[23] D. H. Sanford. Determinates vs. Determinables. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2011 edition, 2011.

[24] M. C. Loui. Computational Complexity Theory. *ACM Computing Surveys*, 28(1), 1996.

[25] C. Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 219–239, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[26] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[27] H. J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[28] R. Reiter. Artificial intelligence and mathematical theory of computation. chapter The Frame Problem in Situation the Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression, pages 359–380. Academic Press Professional, Inc., San Diego, CA, USA, 1991.

[29] W3C OWL Working Group. OWL 2 Web Ontology Language. Technical report, http://www.w3.org/TR/owl2-overview/, 2012.

[30] V. Haarslev and R. Möller. Racer: A Core Inference Engine for the Semantic Web. pages 27–36, 2003.

[31] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2), 2007.

[32] I. Horrocks. The FaCT system. In H. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, volume 1397 of *Lecture Notes in Artificial Intelligenc*, pages 307–312. Springer-Verlag, 1998.

[33] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, Cambridge, UK, 2002.

[34] T. Bittner. Logical properties of foundational mereogeometrical relations in bio-ontologies. *Applied Ontology*, 4(2):109–138, 2009.

[35] K. Lüttich and T. Mossakowski. Specification of Ontologies in CASL. In A. C. Varzi and L. Vieu, editors, *Formal Ontology in Information Systems – Proceedings of the Third International Conference (FOIS-2004)*, pages 140–150, 2004.