Technical Report Number 16

$\mathcal{HOOT}$

A Language for Expressing and Querying $\mathcal{H}$ierarchical $\mathcal{O}$ntologies, $\mathcal{O}$bjects, and $\mathcal{T}$ypes

A Specification

Hassan Aït-Kaci

December 2014

Publication Note

Author's address:

LIRIS—UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08
Email: `hassan.ait-kaci@univ-lyon1.fr`

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 16**

$\mathcal{HOOT}$

A Language for Expressing and Querying $\mathcal{H}$ierarchical $\mathcal{O}$ntologies, $\mathcal{O}$bjects, and $\mathcal{T}$ypes

A Specification

Hassan Aït-Kaci

`hassan.ait-kaci@univ-lyon1.fr`

December 2014

Abstract

$\mathcal{HOOT}$ is a knowledge-base language designed to express and query $\mathcal{H}$ierarchies, $\mathcal{O}$bjects, $\mathcal{O}$ntologies, and $\mathcal{T}$ypes. *Hierarchies* are used to organize taxonomic concepts. *Objects* are used to represent instances of these concepts. *Ontologies* are used to constrain the structure and properties of objects organized in such conceptual hierarchies. *Types* defined in such ontologies embody the formal axioms defining concepts, which axioms must be verified by objects that are instances of the defined concepts. Operationally, these axioms can be used as constraints. Thus, $\mathcal{HOOT}$'s computational semantics consists in *normalizing* such constraints as needed to guarantee their consistency *w.r.t.* an ontology (a Terminological Box, or "TBox") and of the objects referring to it that make up the data as an $\mathcal{RDF}$ triplestore (an Assertional Box, or "ABox"). It also uses TBox knowledge-based reasoning to optimize queries to be submitted to an ABox triplestore before compiling them into $\mathcal{SPARQL}$. This document motivates and specifies the design of a first version of $\mathcal{HOOT}$—its syntax and operational semantics.

Keywords: Semantic Web; Knowledge Representation; Ontological Reasoning; Consistency Checking; Constraint Normalization; Knowledge-Based Processing; $\mathcal{SPARQL}$ Querying.

Table of Contents

1 Introduction

This document is the initial technical specification of a computer language called $\mathcal{HOOT}$ designed for processing taxonomic attributed ontologies. The presentation is organized as follows. Section 2 puts this work in context and motivates it. Section 3 describes the basic idea underlying our contribution in informal terms. Section 4 comprises the essentials of this work's technical material. It describes the (formal and operational) syntax and semantics of $\mathcal{HOOT}$ as a semi-structured object management and query language. Section 4.1 defines a specific grammatical and lexical formal syntax for $\mathcal{HOOT}$. Section 4.2 describes the essential nature of semi-structured, possibly distributed, data and knowledge this syntax is intended to denote in the form of "*order-sorted featured*" ($\mathcal{OSF}$) graphs. Section 4.3 focuses on the part of the language that concerns types and axioms—its terminological part. Then, Section 4.4 discusses a few facts concerning the representation and use of factual data in $\mathcal{OSF}$-form. Finally, Section 4.5 overviews a possible $\mathcal{RDF}$-representation for $\mathcal{OSF}$-terms and a schematic operational interpretation with a $\mathcal{SPARQL}$ query management system when access to intensional data is required. It is followed by a conclusion in Section 5.

2 Motivation

The essential difference between a Data Base (DB) and a Knowledge Base (KB) is that a DB can only provide answers to queries from explicit data, while a KB can use its knowledge and deductive power to provide answers to queries from implicit facts *inferred* to be true, as well as from data *inferred* to be relevant. We are interested in specifying a KB language keeping notation and semantics simple and intuitive, while enabling efficient processing. This is what the Relational Model has provided for DB processing with unmatched success thus far—by being intuitive, formally simple, and operationally efficient to process. Such a state of affairs has yet to be achieved for KB processing.

While $\mathcal{RDF}$ has admittedly provided a (relatively) simple and intuitive representation for knowledge encoded as graphs, efficient reasoning using knowledge about $\mathcal{RDF}$-data has turned out to be a trickier goal to attain. $\mathcal{RDF}$-triple stores comprise Linked Data sets of $\mathcal{RDF}$ triples making up extensional data query-able using knowledge in KBs. This is because data populating triplestores must abide by the intensional knowledge specifying its properties in the KBs they refer to. Knowledge expressed in a KB allows drawing inferences to materialize implicit facts about this data. The set of axioms making up knowledge from which to make such implicit inferences is called a *Terminological Box*—or *TBox* for short. The set of extensional data is called an *Assertional Box*—or *ABox* for short. A TBox may be construed as a kind of Database Schema; or, as type definitions and axioms. Even if limited to simple logical axioms, implicit knowledge is derived as logical consequences thanks to deductive reasoning using inference rules. As a result, a KB should also be capable of answering generic queries about properties of data without necessarily involving actual data retrieval. It should also be capable of using its knowledge to focus data retrieval only on relevant data.

Similarly to how SQL can retrieve explicit data from relational data base, *querying* explicit $\mathcal{RDF}$ data can be done with $\mathcal{SPARQL}$.[1] In fact, $\mathcal{SPARQL}$ is an $\mathcal{RDF}$-triple query language that is a straightforward adaptation of SQL, with similar operations (*viz.*, `select`, `project`, `join`, `ordered-by`, *etc.*, ...) working on the $\mathcal{RDF}$ graph data model rather than relational tables. *Reasoning* with $\mathcal{RDF}$-based knowledge, however, has turned out to be more challenging.

Therefore, the purpose of this document is to restart from scratch and build from the ground up. It describes the syntax and operational semantics of a basic language called "$\mathcal{HOOT}$," a formal and operational system for representing and querying knowledge and data in the form of $\mathcal{H}$ierarchical $\mathcal{O}$ntologies, $\mathcal{O}$bjects, and $\mathcal{T}$ypes. This language is basic in that it is only a bare system that has many limitations as to the complexity of the reasoning it can carry out. Yet, it can express non-trivial ontological knowledge and is designed to support efficient graph-constraint inference. Importantly, these limitations are in no way definitive. Most are simply due to it being a core language meant to be extended with more expressive and/or efficient constructs and operations in future elaborations—this is only $\mathcal{HOOT}$ V0.0.

3 Basic Idea

This version, as well as any future elaboration, of $\mathcal{HOOT}$ is meant for reasoning with, and querying, an $\mathcal{RDF}$ knowledge base, whereby answers may be derived both from axiomatic knowledge about data and the data it describes. It uses a graph-based knowledge reasoning formalism stemming from *first-order term unification* as used in Logic Programming (*e.g.*, Prolog), but extended to a much more expressive, yet as efficient, order-sorted graph-constraint logic. Like Prolog term unification, its essential operational semantics is a syntax-driven *constraint normalization* process [3].

Quite importantly, and deliberately, it uses a simple, intuitive, and familiar *universal* representational syntactic structure. Here, "universal" is meant in the sense of LISP, where *everything* is an S-expression, or Prolog, where *everything* is a first-order term. $\mathcal{HOOT}$'s universal syntactic structure is a labeled graph which can be used for representing knowledge (*i.e.*, TBox axioms), as well as data (*i.e.*, ABox objects) and queries (involving a TBox, with or without an ABox). In $\mathcal{HOOT}$'s case, we represent *everything* as a directed labeled graph. For example, let us assume that we wish to describe a 30-year-old person with an id consisting of a name, itself made of two parts: a first name and a last name, both represented as strings. Let us also say that we wish to indicate that such a person has a spouse that is a person sharing his or her id's last name, and such that this latter person's spouse is the first person in question. We propose to represent this information as the graph in Figure 1.

This graph consists of a set of nodes and arcs between some nodes. This graph, however, is a *labeled* graph. There are two kinds of label symbols: one kind is for the nodes (*e.g.*, **`person`**, **`name`**, *etc.*), and the other is for the arcs. We call the symbols labeling nodes "*sorts*," and the symbols labeling arcs "*features*" (*e.g.*, **`spouse`**, **`age`**, *etc.*). Intuitively, sort symbols denote sets, and feature symbols denote functions be-

[1] `http://www.w3.org/TR/rdf-sparql-query/`

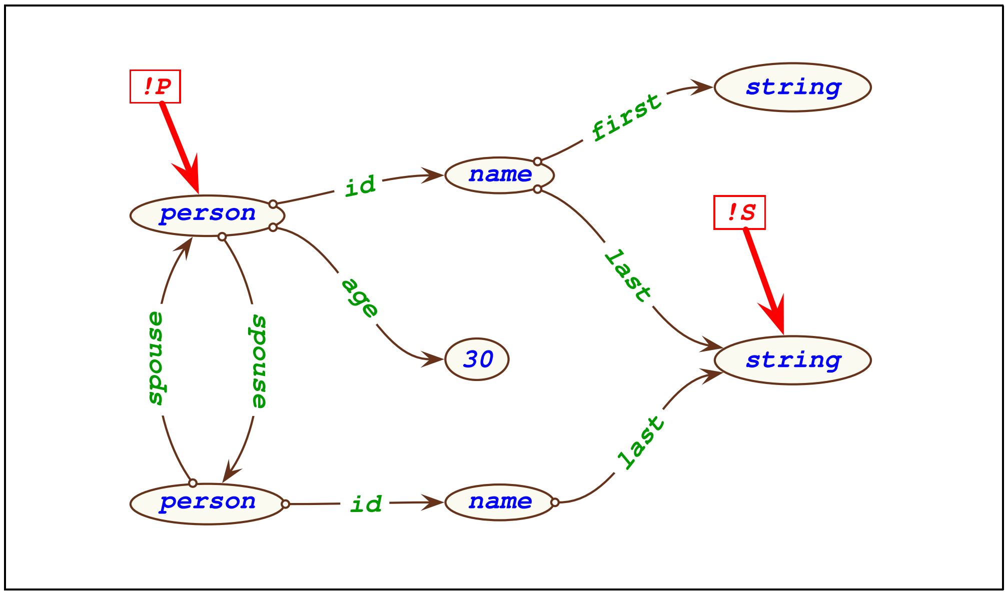

Figure 1: Example of $\mathcal{OSF}$ graph

tween these sets. We will identify value-denoting symbols such as ***30*** as sorts as well since they can be thought of as singleton sets (*e.g.*, in this case the set containing the single integer 30). We will also assume that sort symbols are partially ordered with a "*subsort*" relation "***is-a***" denoting set inclusion on the sets denoted by the sorts. This justifies calling such a graph an "*order-sorted feature*" graph; or, $\mathcal{OSF}$ graph for short.

The reader may wonder what the other labels, such as ***!P*** and ***!S***, that appear in the graph of Figure 1 stand for. These are simply used as pointers to designate specific nodes—so we call them *reference tags*. The symbols used for such tags are only relevant in a specific context to indicate, as is the case in this example, that the nodes they refer to are shared.

The above informal description of an $\mathcal{OSF}$ graph is certainly not difficult to understand to anyone with basic knowledge of the kind of data structure used in object-oriented programming to represent a structured object. This is indeed a good way to think about it. It is this intuitive understanding that we wish the reader to keep in mind. Our intention, however, is to go beyond merely representing and using structured types and objects; we also want to be able to *reason* about them, while never losing this simple intuition. Thus, we propose a simple syntax to represent these graphs that will enable us to do so. For example, we represent the graph shown in Figure 1 using the syntax shown in Figure 2.

The reader with some knowledge of Logic Programming will have noted that this syntax generalizes that of Prolog terms. Indeed, a Prolog term can be seen as a restricted kind of $\mathcal{OSF}$ term, where sort symbols are data constructors (or, to use a more formal logical jargon, uninterpreted Skolem functions); feature symbols are (implicit) subterm positions; and, reference tags are so-called "logical variables." It is to stress this fact that we call an expression such as that in Figure 2 an "$\mathcal{OSF}$ *term*." Note that, unlike a Prolog term where subterms are written following an implicit position order,

```
!P : person(id → name(first → string,
                      last → !S : string),
            age → 30,
            spouse → person(id → name(last → !S),
                            spouse → !P)).
```

Figure 2: $\mathcal{OSF}$ term syntax for the $\mathcal{OSF}$ graph of Figure 1

an $\mathcal{OSF}$ term may be written up to permutation of its subterms since explicit feature labels allow specifying subterms in any order while still representing the same $\mathcal{OSF}$ graph. For example, the $\mathcal{OSF}$ term in Figure 3 could as well be used to represent the same $\mathcal{OSF}$ graph.

```
!P : person(age → 30,
            spouse → person(spouse → !P,
                            id → name(last → !S : string)),
            id → name(last → !S,
                      first → string)).
```

Figure 3: Equivalent $\mathcal{OSF}$ term syntax for the $\mathcal{OSF}$ graph of Figure 1

An $\mathcal{OSF}$-graph can be characterized as the set of all the arcs comprising it. Thus, each arc in such a graph can be construed as a triple of the form $\langle$**Domain**, **Feature**, **Range**$\rangle$ denoting the signature of a function (called its "**Feature**") from a set-denoting sort (called its "**Domain**") to a set-denoting sort (called its "**Range**"). Sorts are partially-ordered—this sort ordering denoting set inclusion. Node sharing is indicated with node *reference tags*.

The next section specifies a basic proof-of-concept prototype for $\mathcal{HOOT}$ using the $\mathcal{OSF}$ formalism to represent, reason with, and query, both intensional and extensional information.

4 The Knowledge-Based Query Language $\mathcal{HOOT}$

We now proceed to give a detailed description of $\mathcal{HOOT}$ as a basic knowledge representation and query language based on the $\mathcal{OSF}$ formalism. A $\mathcal{HOOT}$ knowledge system is a set of *modules*. A module is a syntactic unit that may be one of three kinds:

1. a terminological module (TBox unit) used to define schematic axioms for $\mathcal{OSF}$ structures referring to this module; it usually declares a partially ordered sort taxonomy and a $\mathcal{OSF}$ signature of properties verified by some sorts in this taxonomy, which are recursively inherited by all subsorts;
2. an assertional module (ABox unit) comprising only *ground* labeled-graph ob-

jects denoting individuals—*i.e.*, $\mathcal{OSF}$ object structures populating a graph database conforming to specific terminological modules as schema;

3. a query module (QBox unit) wherein one may specify queries in the form of partially instantiated $\mathcal{OSF}$ structures referring to concepts and features used in terminological modules, seeking to resolve them against any terminological modules they referred to, and/or the contents of specified assertional modules that verify the knowledge declared in the terminological modules.

All three rely on, and use, the syntax of $\mathcal{OSF}$-terms, albeit each with its own particular syntactic restrictions and its own operational semantics. Informally, these can be described as follows:

1. a $\mathcal{HOOT}$ TBox consists of a set of $\mathcal{OSF}$ sorts and features declarations defining the structure of an ontology (namely, its "is-a" sort taxonomy and domain/range constraints on features inherited down this taxonomy); and, an operational semantics that amounts to maintaining the constituents of the TBox in consistent normal form;
2. a $\mathcal{HOOT}$ ABox consists of a set of $\mathcal{OSF}$ ground objects, all of which are instances abiding by the type axioms of a specific TBox; and, an operational semantics that amounts to maintaining the constituents of the ABox in consistent normal form with respect to its TBox (*i.e.*, keeping it "clean" of ill-typed objects and objects referring to such ill-typed objects);
3. a $\mathcal{HOOT}$ QBox consists of a set of $\mathcal{OSF}$ forms that must be kept in consistent normal form *w.r.t.* a TBox, query the TBox, and also the ABox using constraints identifying TBox knowledge and/or the set of objects in an ABox satisfying them; and, an operational semantics consisting of the efficient retrieval of this set (possibly through $\mathcal{RDF}$ query engines).

Thus, it is important to realize that the $\mathcal{OSF}$ term syntax is used in all three parts to represent labeled feature graphs, although with slightly different interpretations due to what they represent and how they are used.

4.1 Syntax

As will be made clear in Section 4.2, an $\mathcal{OSF}$ term such as the one displayed in Figure 2 is in fact a specific kind of $\mathcal{OSF}$ term called "ψ-term." But first, let us specify the formal syntax of general $\mathcal{OSF}$-terms. This is given by the grammar in Figure 4 and the lexical grammar in Figure 5.

In Figure 4, the symbol '`@`' stands for "*anything*" (*i.e.*, the set of all denotable values). It denotes the supreme mother-sort of all sorts—*i.e.*, the abstract sort $\top$ (pronounced "top"). Note also that this grammar of the $\mathcal{HOOT}$ syntax allows for "***`setOf`***" sorts. These sorts allow expressing *roles* in the $\mathcal{OSF}$ formalism, since roles are binary relations, which therefore can be seen as set-valued functional features along the lines described in [4]. Finally, in this grammar, the symbols *`Tag`*, *`BuiltInValue`*, *`BuiltInSort`*, *`Identifier`*, and *`Feature`* are terminal symbols whose lexical structure is detailed in Figure 5. We omit giving rules detailing the lexical structure of the terminal symbols *`IntegerValue`*, *`FloatValue`*, *`CharacterValue`*, and

```
          OsfTerm  ::=  UntaggedOsfTerm
                     |  Tag [ ':' UntaggedOsfTerm ]

  UntaggedOsfTerm  ::=  BuiltInValue
                     |  BuiltInSort
                     |  SetValue
                     |  SetOfSort
                     |  ObjectSort [ '(' SubTermList ')' ]

         SetValue  ::=  '{' [ SetElementList ] '}'

   SetElementList  ::=  SetElement [ ',' SetElementList ]

       SetElement  ::=  BuiltInValue | OTag | SetValue

        SetOfSort  ::=  'setOf' '(' SetElementSort ')'

   SetElementSort  ::=  Sort | SetOfSort

             Sort  ::=  BuiltInSort | ObjectSort

       ObjectSort  ::=  '@' | Identifier

      SubTermList  ::=  SubTerm [ ',' SubTermList ]

          SubTerm  ::=  [ Feature '→' ] OsfTerm
```

Figure 4: BNF grammar for $\mathcal{OSF}$-term syntax

`StringValue`, since they are of the standard of modern programming languages (say, Java)—in particular, a string is a double-quoted sequence of characters, and a character is a single-quoted literal character or its code.[2]

Note also that in Figure 5, there are three kinds of tags. A tag is essentially a syntactic marker for shared terms or query results—it indicates (or "points to") an $\mathcal{OSF}$-term within the *scope* of a $\mathcal{HOOT}$ language construct in a TBox, ABox, or QBox. In other words, any consistent renaming of tags within a defined scope does not change the value of the denotations of the structure in which they appear. Of course, such a scope must be well defined, whether local or global, depending on the nature of the structure, exactly as it is the case in programming languages. More precisely:

- an `ETag` (for "*Equation Tag*")—starting with a '**!**'—is just a tag indicating an equation between parts within the scope of the $\mathcal{OSF}$-term in which is occurs;
- an `OTag` (for "*Object Tag*")—starting with a '**#**'—is a particular kind of `ETag` which may only point to a specific instance, necessarily a "ground" object, in an ABox, or to such a ground object in the global scope of a TBox;[3]
- a `QTag` (for "*Query Tag*")—starting with a '**?**'—is also a particular kind of

[2] `http://character-code.com/`

[3] An `OTag` is what is usually called an *Object Identifier* or "*OID*."

```
            Tag  ::=  ETag | OTag | QTag

           ETag  ::=  '!'  TagId

           OTag  ::=  '#'  TagId

           QTag  ::=  '?'  TagId

          TagId  ::=  (Letter | '_' | '-' | Digit)+

   BuiltInValue  ::=  BooleanValue
                   |  IntegerValue | FloatValue
                   |  CharacterValue | StringValue

   BooleanValue  ::=  'true' | 'false'

    BuiltInSort  ::=  'boolean'
                   |  'integer' | 'float'
                   |  'character' | 'string'

        Feature  ::=  Identifier | NumericFeature

     Identifier  ::=  Letter (Letter | '_' | '-' | Digit)*

 NumericFeature  ::=  NonZeroDigit Digit*

         Letter  ::=  'a' | 'A' | ... | 'z' | 'Z'

          Digit  ::=  '0' | NonZeroDigit

   NonZeroDigit  ::=  '1' | ... | '9'
```

Figure 5: Lexical BNF grammar for terminal symbols used in the grammar of Figure 4

> `ETag` which may only occur in the scope of a QBox to indicate which parts of the structure are to be returned as answers to the query. Any other structure sharing within the scope of a QBox is indicated using a general `ETag` or an `OTag`, as the case may apply.

Like a logical variable, an `ETag` and a `QTag` is essentially a "scoped pointer." As such, it points to a sort node, a value node, or to another tag—and thus must be *dereferenced*. In the latter case, it is said to be *bound*. An "unbound" tag is conveniently represented as a self-referring tag, as commonly done in Prolog implementations [2] (see Page 11 for details). Thus, it can be argued that a tag is always bound since it always refers to a node or another tag—itself, by default.

4.2 Order-sorted featured terms

Since $\mathcal{OSF}$ terms make up the basic data structure to be used by $\mathcal{HOOT}$, it is only fair that the reader get a clear view of what these are, and where they come from. In fact, the examples given in the previous section are, formally speaking, $\mathcal{OSF}$ terms that are in *normal* form. Such a normal form ensures that it is devoid of inconsistency. An $\mathcal{OSF}$ term in normal form is called a *ψ-term*. The expression "ψ-term" was introduced originally by the author in his PhD thesis [1] as a shorthand for "*Property Structure Inheritance Term*," as a formalization of some parts of Ron Brachman's "*Structured Inheritance Networks*" (or "*SI-Nets*") defined informally in his PhD thesis [11].

Normal form for $\mathcal{OSF}$ terms is defined as follows.

DEFINITION 1 ($\mathcal{OSF}$-TERM NORMAL FORM) *An $\mathcal{OSF}$-term is said to be in normal form if, and only if:*

- *it is just the inconsistent sort $\perp$;*[4]
- ***or**,*
  - *the sort $\perp$ appears nowhere in it; **and**,*
  - *all subterms are indicated by distinct features (i.e., there may not be duplicate feature attached to a given sort); **and**,*
  - *each tag occurs at most once as the root of a term; all other occurrences, if any, must be the tag alone with no term.*

For example, while the $\mathcal{OSF}$-term in Figure 6 is an $\mathcal{OSF}$-term is well formed syntactically according to the syntax specified in Figures 4 and 5, it is not in normal form according to Definition 1. On the other hand, the $\mathcal{OSF}$-term in Figure 7 is in normal form; and thus is a ψ-term. It corresponds to the normal form of the $\mathcal{OSF}$-term of Figure 6.

```
!P : person(id → @(first → "John"),
            id → name(last → !S,
                      first → string),
            age → 42,
            spouse → married-person(address → !A : location,
                                    age → integer),
            spouse → @(id → name(first → "Jane",
                                 last → !S : "Doe"),
                       id → name(first → string),
                       spouse → !P : married-person(address → !A,
                                                    age → integer))).
```

Figure 6: Example of $\mathcal{OSF}$-term that is not in normal form

Normalizing an $\mathcal{OSF}$ term can be presented formally as a set of constraint simplification rules acting on a conjunction of simple atomic constraint granules of the form:

[4]This symbol is pronounced "bottom" and denotes the empty set.

```
!P : married-person(id → name(first → "John",
                              last → !S : "Doe"),
                    address → !A : location,
                    age → 42,
                    spouse → married-person(id → name(first → "Jane",
                                                      last → !S),
                                            address → !A,
                                            spouse → !P,
                                            age → integer)).
```

Figure 7: Example of $\mathcal{OSF}$-term in normal form—a ψ-term

- $X : s$,
- $X.f \doteq X'$, and
- $X \doteq X'$.

It is not difficult to see how this formal notation can be obtained from the syntax of an $\mathcal{OSF}$ term by "dissolving" it into the set of sorted node and labeled arcs comprising it. The constraint $X : s$ stands for a sorted node (the sort being s), tagged by a tag X; the constraint $X.f \doteq X'$ represents an arc labeled by a feature f between nodes tagged by X and X'; and $X \doteq X'$ stands for a constraint equation indicating that the graphs with root nodes X and X' are actually the same graphs—which is the case when there are duplicate feature arguments for a same tag.

(1) **SORT INTERSECTION**:

$$\frac{\phi \;\&\; X : s \;\&\; X : s'}{\phi \;\&\; X : s \wedge s'}$$

(2) **FEATURE FUNCTIONALITY**:

$$\frac{\phi \;\&\; X.f \doteq X' \;\&\; X.f \doteq X''}{\phi \;\&\; X.f \doteq X' \;\&\; X' \doteq X''}$$

(3) **INCONSISTENT SORT**:

$$\frac{\phi \;\&\; X : \bot}{X : \bot}$$

(4) **VARIABLE ELIMINATION**:

$$\frac{\phi \;\&\; X \doteq X'}{\phi[X'/X] \;\&\; X \doteq X'} \;\text{[if } X' \text{ occurs in } \phi]$$

Figure 8: Basic $\mathcal{OSF}$-constraint normalization rules

Formal normalization rules acting on conjunctive sets of these granules were given in [8], and recalled in Figure 8. However, while this manner of presenting $\mathcal{OSF}$-term normalization is perfectly satisfying to formalists, it may leave many programmers rather puzzled as to how to implement this process in a conventional object-oriented programming language such as Java. Therefore, we next specify such an algorithm in the form of pseudocode. Besides being closer to an actual implementation, this algorithm has also the merit of working on the structure of an $\mathcal{OSF}$-term as it is written and parsed—*i.e.*, without the need to "dissolve" it first into constraint granules as described above.

Basically, normalizing $\mathcal{OSF}$-term notation consists of transforming the raw syntax of an $\mathcal{OSF}$ term that was parsed using the BNF-grammar rules given in Figures 4 and 5 into a data structure representing a ψ-term. This process is specified as the pseudocode described as Algorithm 1. It consists of two steps: (1) building a (possibly incomplete) ψ-term with one structure per tag and per feature argument, collecting possible duplicate structures as equations to be solved; then, (2) resolving any such collected equations.

```
function NormalizeOsfTerm (RawOsfTerm raw) returns Tag
    Tag tag ← BuildPsiTerm (raw);
    if tag ≠ inconsistentTag and SolveEquations () then
        return tag;
    end
    return inconsistentTag;
end
```

Algorithm 1: Normalizing $\mathcal{OSF}$-term syntax into a ψ-term structure

Before we explain this pseudocode in detail, we need to describe the data structures this pseudocode uses. These are defined to represent the raw syntactic structure of an $\mathcal{OSF}$ term and the data structure of a ψ-term. We do so next, as well as describing the operations used on these structures as pseudocode.

An $\mathcal{OSF}$ term to be normalized comes from parsing notation such as that in Figure 6 into a raw syntax-tree form that uses only string tokens (as opposed to meaning-carrying data structures). These tokens are essentially strings standing for tags, sorts, and features. We represent such a syntax-tree as an object of class `RawOsfTerm` consisting of three fields:

- `tagName`—a string, set to `null` if there is no tag;
- `sortName`—a string, set to `"@"` by default (denoting $\top$, the most general sort subsuming all other sorts);
- `subterms`—a list of pairs of objects of the form $\langle$`feature,term`$\rangle$, representing a feature and the `RawOsfTerm` syntax-tree object that is the subterm it designates; this list is empty by default.

Again, this syntax is not guaranteed to be that of an $\mathcal{OSF}$ term in normal form; so there may be duplicate feature arguments for a same tag, and the same tag may occur as the root of several `RawOsfTerm` objects.

As for a normalized $\mathcal{OSF}$ term to be built from a raw $\mathcal{OSF}$ syntax tree, it is represented as an object of class `PsiTerm`, which consists of two fields:

- `sort`—of type `Sort`;
- `subterms`—a hash table mapping features (strings) to objects of type `Tag`.

The class `Sort` is the type representing partially-ordered symbols making up a concept taxonomy. We will also assume that known sorts are stored in a global (static) hash table, called `taxonomy`, associating strings (sort names) to `Sort` objects. A global (static) method `getSort(String)` will return a sort given its name. For now, we do not worry about what to do if this sort is not defined. We also assume a binary operation on sorts, denoted $\wedge$, that returns their *maximum lower bound*. We will make such details clearer in Section 4.3.2, when we give details on taxonomy classification by encoding sorts using bit-vectors as described in [5].

The data structure `Tag` is a class consisting of three fields:

- `name`— a string: the tag's name;
- `value`—of type `Tag`;
- `term`—of type `PsiTerm`, which is by default set to a `PsiTerm` object with `sort` field set to the top sort, and empty `subterms`.

The field `value` is set by default to point to the tag itself. But it may also be *bound* to another object of type `Tag`. A tag bound to itself (the default), is said to be *unbound*. It is said to be *bound* otherwise. The argumentless method called `deref()` for the class `Tag` returns the first unbound `Tag` object obtained by following the chain of such objects accessible through their `value` field.

We will also assume a global (static) hash table, called `knownTags`, associating strings (tag names) to `Tag` objects. A global (static) method `getTag(String)` will behave as follows on its argument (called `tagName`):

- if `tagName` is `null`, `getTag(tagName)` will create a new tag name and a new tag object with this name, install the pair in the table `knownTags`, and return the newly created `Tag` object; else,
- if there is no `Tag` object in the table `knownTags` associated to the string `tagName`, a new `Tag` object is created with this name, associated to it in the table `knownTags`, before being returned; else,
- the `Tag` object already associated to `tagName` in the table `knownTags` is returned.

In this manner, a `Tag` object is uniquely identified by its name. The `knownTags` table is initialized always to contain a special tag, called `inconsistentTag`, defined to have as name the string `"{}"`, and as `value`, a special `PsiTerm` object called `bottom`, with `sort` set to $\perp$ and no `subterms`. This will be the object returned to indicate an inconsistent $\mathcal{OSF}$ term.

Building a ψ-term for an $\mathcal{OSF}$-term syntax tree using the data structures described above is specifed as Algorithm 2. It defines a function called BUILDPSITERM that takes an object of type `RawOsfTerm` and returns an object of type `Tag`. This function recursively walks down the syntax-tree structure, building one ψ-term per subtree and installing it as subterm of the ψ-term being built. If there are duplicate ψ-terms for a feature argument, they are collected into a global (static) `Stack` called `equations`.

Solving the equations accumulated in the `equations` stack by BUILDPSITERM is done by the Boolean function SOLVEEQUATIONS, specified as Algorithm 3. This function returns **true** if, and only if, the $\mathcal{OSF}$ term being normalized is consistent. Algorithm 3 is in fact the ψ-term unification algorithm introduced in [1] and used in [7] for order-sorted Logic Programming. It is a simple but powerwul inference algorithm that performs deduction modulo an "***is-a***" concept taxonomy.

```
function BuildPsiTerm (RawOsfTerm raw) returns Tag
    Tag tag $\leftarrow$ getTag(raw.tagName);
    Sort sort $\leftarrow$ tag.term.sort $\wedge$ getSort(raw.sortName);
    if sort $=$ $\bot$ then
        return inconsistentTag;
    end
    tag.term.sort $\leftarrow$ sort;
    foreach $\langle$feature,rawsub$\rangle$ $\in$ raw.subterms do
        Tag subtag $\leftarrow$ BuildPsiTerm (rawsub);
        if subtag $=$ inconsistentTag then
            return inconsistentTag;
        end
        if $\langle$feature,_$\rangle$ $\notin$ tag.term.subterms then
            tag.term.subterms.add($\langle$feature,subtag$\rangle$);
        else
            equations.push(tag.term.subterms.get(feature));
            equations.push(subtag);
        end
    end
    return tag;
end
```

Algorithm 2: Building ψ-term structure from $\mathcal{OSF}$-term syntax

4.3 Knowledge—$\mathcal{HOOT}$ TBox

This section describes the $\mathcal{HOOT}$ language terminological part, which contains the axioms that constrain the structural and semantic nature of objects they range over.

4.3.1 Sort and feature declarations

The most basic semantic information is that of declaring a conceptual taxonomy. This set of declarations is organized into a partial order among concepts, whereby all properties of a concept are implicitly inherited by all its subconcepts.[5]

As for ensuring that structural information is semantically consistent with constraints declared in a TBox, it is possible to declare the form expected by a ψ-term when its root-sort symbol is such-and-such symbol. This kind of structural constraint must also be inherited down the concept taxonomy. Ideally, it would be desirable that a TBox express $\mathcal{OSF}$ theories specifying constraints of a more complex nature than simple feature domain/range. However, enforcing arbitrary constraints can turn out to be quite tricky, or even undecidable [9]. This is why we wish to start only with domain/range constraints: the problem is not only decidable, but clever processing

[5]We refer to “concepts” equally as “sorts” or “types” (same for sub/superconcept as sub/supersort) for the symbols defined in a taxonomy, each concept (or sort, or type) denoting a set and subconcept (subsort, subtype) relation denoting set inclusion.

```
function SOLVEEQUATIONS () returns Boolean
    Tag lhs;
    Tag rhs;
    while not equations.isEmpty() do
        lhs ← equations.pop().deref();
        rhs ← equations.pop().deref();
        if lhs ≠ rhs then
            Sort sort ← lhs.term.sort ∧ rhs.term.sort;
            if sort = ⊥ then
                return false;
            end
            rhs.sort ← sort;
            lhs.value ← rhs;
            foreach ⟨feature,lhsarg⟩ ∈ lhs.term.subterms do
                if ⟨feature,rhsarg⟩ ∈ rhs.term.subterms then
                    equations.push(rhsarg);
                    equations.push(lhsarg);
                else
                    rhs.term.subterms.add(⟨feature,lhsarg.deref()⟩);
                end
            end
        end
    end
    return true;
end
```

Algorithm 3: Solving ψ-term equations

also enables surprisingly expressive TBox reasoning and remarkably efficient ABox querying as demonstrated in [10]. We next specify $\mathcal{HOOT}$'s TBox declaration syntax, then we detail how to preprocess such knowledge for efficient TBox inference and ABox querying.

A basic grammar's set of BNF rules defining the $\mathcal{HOOT}$ syntax for TBox sort and feature declarations is given in Figure 9.

```
     OSFDECLARATIONS ::= ISADECLARATION+
                       | FEATUREDECLARATION+

      ISADECLARATION ::= SORTLIST `is-a' SORTLIST `.'

            SORTLIST ::= SORT (`,' SORT)*

  FEATUREDECLARATION ::= Feature `:' DOMAINRANGELIST `.'
                       | SORT `(' FEATURERANGELIST `)' `.'

     DOMAINRANGELIST ::= DOMAINRANGE (`,' DOMAINRANGE)*

         DOMAINRANGE ::= SORT `→' SORT

    FEATURERANGELIST ::= FEATURERANGE (`,' FEATURERANGE)*

        FEATURERANGE ::= SORT
                       | FEATURE `→' SORT
```

Figure 9: BNF grammar for $\mathcal{OSF}$ declarations

The most basic sort declaration is of the form: "s ***is-a*** t." which declares the sort s as an immediate subsort of sort t. The shorthand: "$\mathtt{s}_1, \ldots, \mathtt{s}_n$ ***is-a*** $\mathtt{t}_1, \ldots, \mathtt{t}_m$." is the same as the $n \times m$ declarations: "$\mathtt{s}_i$ ***is-a*** $\mathtt{t}_j$." for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

The most basic feature declaration is of the form: "$\mathtt{f} : \mathtt{d} \to \mathtt{r}$." which declares that the feature f's domain is the sort d, and its range is the sort r. The form: "$\mathtt{f} : \mathtt{d}_1 \to \mathtt{r}_1, \ldots, \mathtt{d}_n \to \mathtt{r}_n$." is a shorthand for the n feature declarations: "$\mathtt{f} : \mathtt{d}_1 \to \mathtt{r}_1$.", …, "$\mathtt{f} : \mathtt{d}_n \to \mathtt{r}_n$.". We can also use the notation: "$\mathtt{d}(\mathtt{f}_i \to \mathtt{r}_i, \ldots, \mathtt{f}_n \to \mathtt{r}_n)$." as a shorthand for the n feature declarations: "$\mathtt{f}_1 : \mathtt{d} \to \mathtt{r}_1$.", …, "$\mathtt{f}_n : \mathtt{d} \to \mathtt{r}_n$.". Note that, as is the case in $\mathcal{OSF}$-term syntax, implicit-position features can be used. Thus, the declaration: "$\mathtt{d}(\mathtt{r}_i, \ldots, \mathtt{r}_n)$." is a shorthand for the n positional-feature declarations: "$1 : \mathtt{d} \to \mathtt{r}_1$.", …, "$n : \mathtt{d} \to \mathtt{r}_n$.". Finally, mixing explicit-feature and implicit-position argument range specifications are allowed as is the case in the syntax of subterms of $\mathcal{OSF}$ terms. Implicit-positions are simply identified in the order they come to the number-feature corresponding to their rank in the argument list. Number-features can also be specified explicitly as well.

4.3.2 TBox reasoning

We now explain static preprocessing of the TBox declarations to optimize inference. Operationally, a $\mathcal{HOOT}$ TBox is statically processed by:

1. *encoding the sort taxonomy* using bit-vectors to make sort intersection, union, and complementation compiled into efficient logical operations on bit-vectors (namely, **and**, **or**, and **not**) [5, 6];
2. *propagating and checking the consistency* of a feature's domain/range declarations through the taxonomy by inheritance.

Bit-vector encoding of sorts Knowledge in a TBox consists of the "***is-a***" ordering among sorts making up a taxonomy. This may be viewed as constituting a logical theory (*i.e.*, a set of axioms) limited to monadic implications since "a ***is-a*** b" expresses the same logical semantics as the monadic-implication axiom $\forall x : \ a(x) \ \Rightarrow \ b(x)$, where a sort is formalized as a monadic predicate. The point of using partially-ordered sorts as done in $\mathcal{OSF}$-constraints rather than general-purpose logical reasoning with monadic-implication axioms is that it is operationally much more efficient. We recall next how this may be done in virtually constant time.

In [5], a method is described to encode sorts as bit-vectors. It is recalled here as the pseudocode procedure ENCODETAXONOMY expressed as Algorithm 4. The class

```
procedure ENCODETAXONOMY ()
    Set layer ← $\bot$.parents;
    while layer $\neq \emptyset$ do
        foreach x $\in$ layer do
            x.code ← $2^{\texttt{x.index}} \vee \bigvee_{\texttt{y}\in\texttt{x.children}}$ y.code;
            x.coded ← true;
        end
        layer ← $\bigcup_{\texttt{x}\in\texttt{layer}}$ x.parents;
        foreach x $\in$ layer do
            if $\exists$ y $\in$ x.children such that not y.coded then
                layer.remove(x);
            end
        end
    end
end
```

Algorithm 4: Encoding the sorts of a taxonomy as bit-vectors

`Sort` is given two new fields of type `Set`, called "`parents`" and "`children`" containing respectively, for any sort, its sets of immediate children and parents in the taxonomy. Thus, for every sort object, these sets are filled with sorts by processing the "***is-a***" declarations. In addition, the class `Sort` has a field called "`code`" that is its bit vector representation (initialized to be all zeroes), a field called "`index`" that is an integer, its unique characteristic rank in the array `taxonomy` containing all the sorts, and a `Boolean` field called "`coded`" indicating whether this sort has been encoded or not (so it is initially set to **false**). The procedure is a very efficient computation of the reflexive-transitive closure of the "***is-a***" ordering as explained in [6].[6]

Processing and enforcing feature declarations In addition to taxonomic ordering, knowledge in a TBox may be enhanced with additional axioms, as long as enforcing these axioms may be performed computationally. We next add another kind of axioms to a TBox to express domain and range constraints for features, which denote functions. This enables making consistency inference regarding which feature appears between which sorted node in an $\mathcal{OSF}$ term, and use such inference to normalize further a ψ-term using this knowledge.

For example, let us assume that the sort `married-person` was declared as a subsort or the sort `person`. Let us also assume the feature declaration `spouse` : `married-person` $\rightarrow$ `married-person`. Then, with these declarations, the ψ-term in Figure 2 is further normalized into the ψ-term in Figure 10.

```
!P : married-person(id → name(first → string,
                              last → !S : string),
                    age → 30,
                    spouse → married-person(id → name(last → !S),
                                            spouse → !P)).
```

Figure 10: Feature-normalized ψ-term of Figure 2

Formally, declaring the domains and ranges of features of a set $\mathcal{F}$ as sorts from a partially-ordered set of sorts $\mathcal{S}$ (formalizing a taxonomy) can be expressed as a set of constraint axioms of the form $\{f_i : d_i \rightarrow r_i\}$, where $f_i \in \mathcal{F}$, $d_i \in \mathcal{S}$, and $r_i \in \mathcal{S}$. We call such a set of feature constraints an $\mathcal{OSF}$ *theory*. A domain/range feature declaration `f` : `d` $\rightarrow$ `r` is formalized as adding the formal constraint $f : d \rightarrow r$ into an $\mathcal{OSF}$ theory called Θ, a conjunctive set of such constraints which is part of a TBox. The empty theory is the all-permissive $\mathfrak{true}$ constraint. The inconsistent theory noted $\Theta_\perp$ is the all-forbidding $\mathfrak{false}$ constraint.

When declaring an $\mathcal{OSF}$ theory of n constraints $\Theta \overset{\text{def}}{=} \{f_i : d_i \rightarrow r_i\}_{i=0}^{n}$, it is necessary that it be verified to be self-consistent. This means that it must contain no feature that can be inferred to have an inconsistent *range* for any of the declared partial features.[7] This consistency check is what the classification preprocessing of the taxonomy amounts to. It is a constraint propagation and normalization process. Static taxonomy classification preprocessing also improves the operational performance of $\mathcal{OSF}$ normalization since it eliminates the need for dynamic taxonomic lookups for relevant feature domain/range constraints.

[6] *op. cit.*, pages 125–126.

[7] A feature may not be declared with inconsistent *domain* but consistent range. This would just be eliminated as a vacuous axiom (*i.e.*, $\mathfrak{true}$) since it is always verified. Indeed, all declared features apply vacuously to the empty set by inheritance.

The idea is simple: given a taxonomy defining a partial order on sorts that have been encoded as bit-vectors, a feature f's domain/range declaration of the form $f : d \to r$ is propagated to a subsort s of d as follows:

- if there is no declaration for f for the sort s, then we simply install the declaration $f : s \to r$ for the sort s, and iterate the process for subsorts of s;
- if there is already a declaration for $f : s \to r'$ for the sort s, then we normalize it to be $f : s \to r \wedge r'$, where $r \wedge r'$ is the (binary code of) the conjunction of (the binary codes of) the sorts r and r'. If this code is all 0's, this means that the feature declaration is inconsistent, and so is the taxonomy.

Clearly, this process always terminates: it is in fact linear in the number of declared features and the number of subsorts of their domains. If no inconsistency is found, the resulting taxonomy is then normalized into a consistent set of feature declarations.

For example, assume that the sort ordering on sorts is such that:

```
researchScientist is-a researcher
researchScientist is-a scientist
scientificResearch is-a research
scientificResearch is-a science
```

and that we have the feature declarations:

```
interestedIn : researcher → research
interestedIn : scientist → science
```

Feature propagation brings these declarations for feature **`interestedIn`** from the sorts **`researcher`** and **`scientist`** down to the sort **`researchScientist`**, for which they are normalized into the single declaration:

```
interestedIn : researchScientist → scientificResearch.
```

This normalization results in a consistent taxonomy by coercing the range sort of the feature declaration **`interestedIn`** on the domain sort **`researchScientist`** to the most general sort that is compatible with the declarations of this feature inherited from its supersorts. If there had been no compatible range sorts for this feature, this normalization would have reported an inconsistent feature declaration. The normalization rules for taxonomy consistency check are described more formally next.

Ensuring taxonomy consistency is a process called *classification*. It computes all implicit sort subsumption relationships and inheritance of properties. An efficient classification algorithm was proposed in our previous work [5]. This specification extends the classification performed in [5] with verifying feature consistency with respect to declared domain/range constraints for features. Using the rules of Figure 11, it propagates feature declarations down the taxonomy, normalizing these declarations to be consistent if need be, or reporting inconsistencies. A consistent normalized taxonomy

($\mathcal{O}_1$) **FEATURE INHERITANCE**:

$$\frac{\Theta \cup \{f : d \rightarrow r\}}{\Theta \cup \{f : d \rightarrow r, f : d' \rightarrow r\}} \quad [\textbf{if } d' \textbf{ is-a } d \textbf{ and } f : d' \rightarrow r \notin \Theta]$$

($\mathcal{O}_2$) **FEATURE-RANGE NORMALIZATION**:

$$\frac{\Theta \cup \{f : d \rightarrow r, f : d \rightarrow r'\}}{\Theta \cup \{f : d \rightarrow r \wedge r'\}}$$

($\mathcal{O}_3$) **INCONSISTENT FEATURE RANGE**:

$$\frac{\Theta \cup \{f : d \rightarrow \bot\}}{\Theta_\bot}$$

Figure 11: TBox propagation and normalization rules for $\mathcal{OSF}$ feature declarations

(or TBox) can then be used for normalizing queries in $\mathcal{HOOT}$ form, thus optimizing potential ABox instance retrieval. How this is done is described next.

Thus, a taxonomy's feature classification will propagate a declared feature constraint $f : d \rightarrow r$ to any subsort of its domain sort d down the taxonomy. Operationally, this is achieved by transforming an $\mathcal{OSF}$ theory Θ, applying the rules shown in Figure 11 until they do not change the $\mathcal{OSF}$ theory, or until they produce the inconsistent theory $\Theta_\bot$. Clearly, $\mathcal{OSF}$ theory classification can, and should, be carried out as a static preprocessing step to save runtime processing.

Once a fully classified and normalized consistent $\mathcal{OSF}$ theory is at hand, one can still use the basic syntax-directed $\mathcal{OSF}$-term normalization rules of Figure 8. Taking into account feature domain-range constraints from an $\mathcal{OSF}$ theory is done with the addition of a single rule to this rule set—the one we give below in Figure 12.

Before we do so, let us first come back to pseudocode to define how we can organize an actual implementation of $\mathcal{OSF}$-term normalization modulo a domain/range feature theory. So we now explain the data-structure architecture and pseudocode specification for processing feature declarations into a normalized consistent taxonomy. Again, all the algorithms we specify simply implement systematically the declarative rules of Figure 11 in a Java-like programming style.

Each `Sort` is given a new field, called `features`, a hash table associating any feature name declaring the sort as its domain to its range, a `Sort` object. We will also assume a global structure, called `definedFeatures`, a hash table associating to each declared feature the set of maximal sorts for which it is defined.

Processing a feature declaration `f : d` $\rightarrow$ `r` then consists in the pseudocode shown as Algorithm 5. It uses a global `Boolean` to indicate whether the feature declaration was not found to be inconsistent. It consists in two steps: (1) propagating and checking for consistency the declaration for the specified domain and all its subsorts; and, (2) if it is consistent, update the global `definedFeatures` table.

```
procedure DECLAREFEATURE (String feature, Sort domain, Sort range)
    consistentDeclaration ← true;
    PROPAGATEFEATURE (feature, domain, range);
    if consistentDeclaration then
        UPDATEDEFINEDFEATURES (feature,domain);
    end
end
```

Algorithm 5: Processing a sort declaration

The procedure PROPAGATEFEATURE proceeds as follows: if the specified domain sort is $\bot$, it exits—since there is nothing to be declared for the bottom sort. Otherwise, if the feature is not defined for the specified domain sort, it puts the specified range sort in the domain's feature table, then propagates the declaration recursively to the domain's children. Otherwise, it intersects the range with the range sort that was previously defined for this domain. If this yields an inconsistent range, the procedure aborts all further work and reports an inconsistent feature declaration. Otherwise, if this intersection is actually different from the previously defined range, it sets the range to be this intersection and propagates the declaration with this coerced range to the domain's children.

Updating the defined feature table is specified as Algorithm 7, which keeps only maximal domains for every declared feature. Note that there is no need to record also the ranges in the table `definedFeatures` because retrieving the range that corresponds to a given feature and domain is accessible as the domain's `features` table's value for the given feature.

Coming back to formal constraint normalization rules, we still need to explain how to take feature declarations into account.

In a normalized consistent $\mathcal{OSF}$ theory Θ, for any declared feature, f we will denote as $\mathbf{Dom}_{\Theta}(f)$ the set of maximal sorts d in $\mathcal{S}$ such that $f : d \to r \in \Theta$ for some sort r, and $\mathbf{Ran}^{d}_{\Theta}(f) \in \mathcal{S}$ for $d \in \mathbf{Dom}(f)$ the (necessarily unique) range sort corresponding to d.

Then, in the context of such a normalized consistent $\mathcal{OSF}$ theory, the rule shown in Figure 12 can be used in conjunction with the basic $\mathcal{OSF}$-term normalization rules of Figure 8 to enforce a declared feature's domain/range constraints whenever this feature appears in an $\mathcal{OSF}$-term expression. Note that, contrary to the rules in Figure 8, this new rule is *non-deterministic*; *i.e.*, it involves making a choice among several potential domains $d \in \mathbf{Dom}_{\Theta}(f)$ for given feature f. This is is due to the fact that there may be several maximal domains for which a given feature is defined.

As we have done before, we next explicate how to implement algorithmically $\mathcal{OSF}$-term normalization in the context of declared features in the form of pseudocode. Given a normalized sort taxonomy with feature declarations (*i.e.*, in which these have

```
procedure PropagateFeature (String feature, Sort domain,
Sort range)
    if domain = ⊥;
    then
        exit;
    end
    Sort sort ← domain.features.get(feature);
    if sort = null then
        domain.features.put(feature,range);
        foreach Sort child ∈ domain.children do
            PropagateFeature (feature,child,range);
        end
    else
        sort ← sort ∧ range;
        if sort = ⊥ then
            consistentDeclaration ← false;
            abort ("inconsistent feature declaration");
        else
            if sort ≠ range then
                domain.features.put(feature,sort);
                foreach Sort child ∈ domain.children do
                    PropagateFeature (feature,child,sort);
                end
            end
        end
    end
end
```

Algorithm 6: Propagating a feature declaration in the taxonomy

```
procedure UPDATEDEFINEDFEATURES (String feature, Sort domain)
    Set domains ← definedFeatures.get(feature);
    if domains = null then
        definedFeatures.put(feature,new Set().add(domain));
    else
        foreach Sort sort ∈ domains do
            if domain.isSubsortOf(sort) then
                exit;
            else
                if sort.isSubsortOf(domain) then
                    domains.remove(sort);
                end
            end
        end
        domains.add(domain);
    end
end
```

Algorithm 7: Updating the defined feature table to its maximal domains

(5) **DECLARED FEATURE**:

$$\frac{\phi \ \& \ X.f \doteq Y}{\phi \ \& \ X.f \doteq Y \ \& \ X : d \ \& \ Y : r} \ [\textbf{if } d \in \textbf{Dom}_{\Theta}(f) \textbf{ and } r = \textbf{Ran}^{d}_{\Theta}(f)]$$

Figure 12: $\mathcal{OSF}$-term normalization involving declared features

been verified consistent and propagated), an $\mathcal{OSF}$-term that has been normalized into a ψ-term object of class `PsiTerm` must be further normalized to abide by the domain/range constraints of features that occur in it.

There are two modes in which this may be done:

- a *permissive mode*, whereby any non-declared feature `f` is implicitly assumed as if declared with `f` : $\top \rightarrow \top$; and,
- a *strict mode*, whereby any non-declared feature `f` is implicitly assumed as if declared with `f` : $\top \rightarrow \bot$.

Thus, in permissive mode, only declared features are made to obey the domain/range constraints specified in their declarations, while all others are considered defined for all sorts with no range constraints. Whereas, in strict mode, any undeclared feature will be deemed inconsistent.

We will assume that feature-consistency normalization mode can be set at the discretion of a user since there are advantages and drawbacks to both modes. Indeed, permissive mode has the advantage of relieving the user from having to be exhaustive in specifying domain/range declarations for *all* features. However, it will also prevent catching common errors such as misspelled features. On the other hand, strict mode will guarantee that no feature is ever used with a domain or range that was not meant for it. But it will also require that all features be declared—even if only as f : $\top \rightarrow \top$. The pseudocode we specify here provides both possibilities. For this, we assume a global (static) Boolean flag called strictMode, which can be set at the discretion of the user.

```
procedure NormalizeFeatures (Tag tag, Set dejavu)
  if tag ∉ dejavu then
    dejavu.add(tag);
    foreach ⟨feature,subtag⟩ ∈ tag.term.subterms do
      if ⟨feature,domains⟩ ∈ definedFeatures then
        Sort domain ← domains.pick();
        domain ← domain ∧ tag.term.sort;
        if domain = ⊥ then
          abort ("inconsistent feature domain");
        end
        tag.term.sort ← domain;
        Sort range ← domain.features.get(feature);
        range ← range ∧ subtag.term.sort;
        if range = ⊥ then
          abort ("inconsistent feature range");
        end
        subtag.term.sort ← range;
      else
        if strictMode then
          abort ("undefined feature in strict mode");
        end
      end
      NormalizeFeatures (subtag,dejavu);
    end
  end
end
```

Algorithm 8: Normalizing a ψ-term *w.r.t.* defined features

Normalizing a ψ-term with respect to a consistent normalized set of feature declaration that has been propagated through a taxonomy is done by the NormalizeFeatures procedure whose pseudocode is given as Algorithm 8. This procedure is a straightforward algorithmic adaptation of the normalization rule **Declared Feature** of Figure 12. It recursively traces the features attached to a given tagged term, enforcing domain/range constraints of defined features on the corresponding subtags in all the tagged term's subterms. For the recursion to be well-founded even in the presence of cycles, it passes along as argument a Set of Tag objects, called dejavu, recording previously processed tags.

Note that, just like the rule **Declared Feature**, the NormalizeFeatures procedure is also a *non-deterministic* algorithm. This is reflected on Line 6 where a domain is picked as a choice among several potentially contained in the set of maximal domains on which the feature `feature` is defined. In a real implementation, this could be handled as a possible backtracking point to come back to in case a domain choice leads to inconsistency. Implementing such a backtracking process should then involve saving information to be undone for making another domain choice for a declared feature, if there is any more, to be tried upon inconsistency. A Prolog-style architecture may then be used for this purpose [2]. We will not give such details here. Suffice it to say that every structure modification between such choice points must then be recorded as a state of computation to be potentially recovered upon inconsistency before making another choice (if any is available), or report definitive inconsistency. Of course, this entails more involved book-keeping, but is not difficult to set up. In addition, the number of choices being finite, the process is clearly decidable. Collecting the set of all resulting feature-consistent ψ-terms can therefore be done in this way. This is not as expensive at it may appear since the number of resulting consistent ψ-terms for meaningful structures can be expected to be very small, and often just one, due to the fact that multiple-domain features rarely relate to compatible structures.

The reader is encouraged to trace the effect of Algorithm 8 on the ψ-term of Figure 2 and verify that it actually yields the ψ-term of Figure 10, provided that the sort `married-person` was declared as a subsort or the sort `person` along with the feature declaration `spouse` : `married-person` $\rightarrow$ `married-person`, in either permissive or strict mode (assuming, for the latter, other needed feature declarations such as `id` : `person` $\rightarrow$ `name`, `first` : `name` $\rightarrow$ `string`, *etc.*).

4.4 Data—$\mathcal{HOOT}$ ABox

This section describes the $\mathcal{HOOT}$ language assertional part—the form and contents of the data it expects. As elsewhere, a $\mathcal{HOOT}$ ABox is a set of $\mathcal{RDF}$ triples. It may be a (set of possibly distributed) triplestore(s) (*i.e.*, one or several possibly distributed databases of such objects) or an in-memory dataset. It is populated with triples representing object instances abiding by axioms (sorts and features) defined in one or several (self- and mutually consistent) TBoxes.

4.4.1 Data syntax

The elements populating an ABox are *ground* objects. Informally, a ground object denotes the value of a structured element. Syntactically, a ground object is also represented as an $\mathcal{OSF}$ term as specified in the grammar of Figure 4, but with the additional syntactic constraints given by the more specific BNF rules of Figure 13. In other words, an ABox object can only be, or contain, an *extensional value*-denoting (as opposed to an *intensional set*-denoting) $\mathcal{OSF}$ term expression. Note that a *set of ground objects* is deemed a *extensional value* in this setting.

Yet another syntactic well-formedness for a *ground set* object is that it may contain an *`OTag`* as one of its elements only if this *`OTag`* occurs somewhere else in the ABox as the root tag of a ground object.

```
           GROUNDOSFTERM  ::=  UNTAGGEDGROUNDOSFTERM
                            |  OTag [ `:' UNTAGGEDGROUNDOSFTERM ]

   UNTAGGEDGROUNDOSFTERM  ::=  BuiltInValue
                            |  SETVALUE
                            |  OBJECTSORT `(' SUBGROUNDTERMLIST `)'

       SUBGROUNDTERMLIST  ::=  SUBGROUNDTERM [ `,' SUBGROUNDTERMLIST ]

           SUBGROUNDTERM  ::=  [ Feature `→' ] GROUNDOSFTERM
```

Figure 13: BNF grammar specific to ground $\mathcal{OSF}$ terms—*i.e.*, ABox values

Finally, an `OTag` may not be nested inside a set value unless it is inside the subterm under a feature. For example:

```
#X : { person(age → 25), ..., #X, ...}
```

is not syntactically well-formed; whereas the following:

```
#X : { person(age → 25, likes → #X), ...}
```

is well-formed.

4.4.2 Data semantics

We will assume such a TBox always to be consistent and normalized; namely, all feature declarations are checked for consistency and propagated through the taxonomy. Data populating an ABox with objects whose structure uses sorts and features from a TBox must abide by TBox axioms constraining them. Thus adding an object to an ABox may be done only if this object is normalized and checked consistent with respect to the TBox types and features that occur in its structure. For example, let us assume that the following axioms are declared in the TBox:

```
married-person is-a person.

person ( id → name
       , age → integer
       ).

married-person ( spouse → married-person
               ).

name ( first → string
     , last → string
     ).
```

and the following ground objects:

```
#P2753 :  person ( id → #N691
                 , spouse → #P3902
                 )

#P3902 :  person ( id → #N873
                 , age → 33
                 , spouse → #P2753
                 )

#N691 :  @ ( first → "John"
           )

#N873 :  @ ( first → "Jane"
           , last → "Doe"
           )
```

In order to be added to the ABox, these objects must be normalized to be:

```
#P2753 :  married-person ( id → #N691
                         , spouse → #P3902
                         )

#P3902 :  married-person ( id → #N873
                         , age → 33
                         , spouse → #P2753
                         )

#N691 :  name ( first → "John"
              )

#N873 :  name ( first → "Jane"
              , last → "Doe"
              )
```

Note that object normalization does not mind that some objects may be incomplete with respect to declared features. However, it does check that features that are present are correctly typed, and infers correct type information wherever required.

Operationally, the sorting semantics of the features from the ABox may only be populated with objects that are structurally conform to the TBox declarations. This is done at ABox-population time simply by normalizing a candidate ABox object (*i.e.*, a *ground* $\mathcal{OSF}$ graph) using the rules in Figure 8 on Page 9 and Figure 11 on Page 18, which are given in pseudocode respectively as Algorithm 1 on Page 10 and Algorithm 8 on Page 22.

4.5 Queries—$\mathcal{HOOT}$ QBox

This section describes the $\mathcal{HOOT}$ language query part—the QBox.

4.5.1 Query syntax

Just like a TBOX or an ABox, a QBox can be expressed using $\mathcal{OSF}$ terms. If involving ABox data, some such $\mathcal{OSF}$ queries have a corresponding interpretation as a type-normalized type-indexed $\mathcal{SPARQL}$ query [10].

QBox $\mathcal{OSF}$-terms may use all three kinds of tags: (1) `QTags` (**`?...`**), (2) `ETags` (**`!...`**), and (3) `OTags` (**`#...`**). All are tags—the only difference between all three kinds to indicate, respectively, (1) requested query results, (2) equality constraints that must be verified, (3) specific ABox object instances or builtin constants.

Informally, the semantics of a $\mathcal{HOOT}$ query consisting of a conjunctive collection of $\mathcal{OSF}$ constraints of the form **`?X:t`** in the context of a (normalized consistent) TBox and (normalized consistent) ABox, is to find the set of all maximal TBox-consistent ψ-terms (including ABox instances) that also verify the $\mathcal{OSF}$ graph constraints expressed by the query. Operationally, this consists in:

1. normalizing the query to determine whether it may be proven consistent with the TBox; and,
2. if needed, collecting all the ABox instances that verify the constraint that results as the normalized query; collecting such instances from an ABox is achieved by compiling the normalized $\mathcal{HOOT}$ query's original $\mathcal{OSF}$ form into an equivalent $\mathcal{SPARQL}$ query in $\mathcal{RDF}$ form;
3. displaying the $\mathcal{RDF}$ graphs with roots corresponding to QTags that verify the query in their equivalent $\mathcal{OSF}$ syntax—which is simpler to manipulate and read than its underlying $\mathcal{XML}$-based $\mathcal{RDF}$ syntax.

The operational semantics for interpreting a $\mathcal{HOOT}$ query term in the context of a (collection of) TBox(es) and ABox(es) consists in:

1. using TBox axioms to normalize the query term;
2. if the query is strictly a TBox query that does not involve any ABox access, returning the ψ-terms bound to all QTags in the query;
3. if the query involves accessing ABox data, collecting ABox objects into sets.

In the latter case, the ψ-term corresponding to the normalized query is compiled into an equivalent $\mathcal{SPARQL}$ query assuming an $\mathcal{RDF}$ representation of the data in the ABox.

4.5.2 ABox $\mathcal{RDF}$ representation and processing

Since a ψ-term is a notation for a labeled graph, an $\mathcal{RDF}$ notation for it can be readily derived. Such a representation must account for differences between conventions of both notations. The essential difference is that, whereas in a ψ-term all nodes are labeled uniformly with sort symbols or values, $\mathcal{RDF}$ makes a difference between nodes that are labeled with URIs, blank nodes, and value nodes, and arcs are labeled with URIs. Also, rather than labeling typed nodes with their types, it uses an arc labeled with the specific URI **`rdf:type`** pointing to such a node (which is then a unique representation for a type, and thus shared by all so-typed nodes). Hence, an $\mathcal{RDF}$ representation for a ψ-term is obtained as a straightforward adaptation accounting for these differences. For example, the $\mathcal{RDF}$ graph corresponding to the $\mathcal{OSF}$ graph of Figure 1 (which is that of a ψ-term since in normal form) is as shown in Figure 14, where the prefix **`osfex`** refers to a (hypothetical) URI where the specific sort and feature symbols it uses could be defined as identifiers (`osfex:person`, for example).

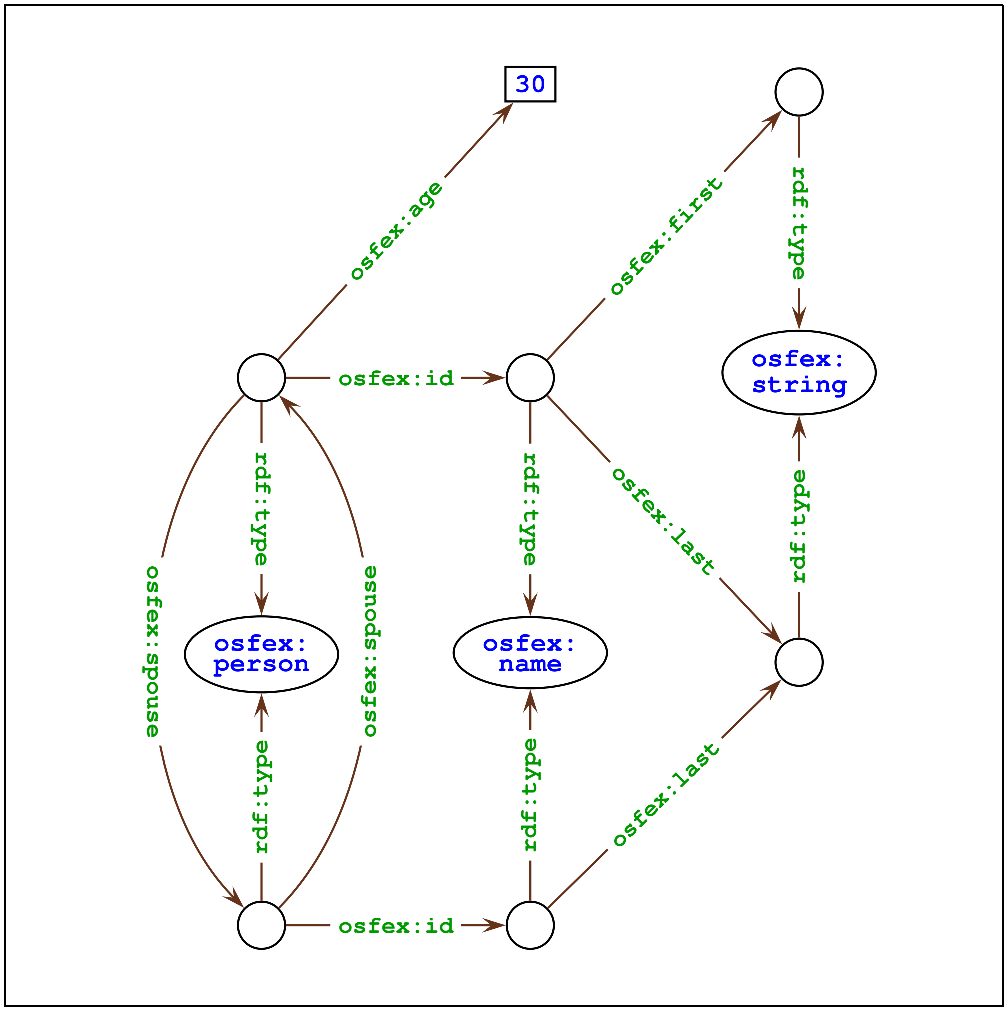

Figure 14: $\mathcal{RDF}$ graph version of the $\mathcal{OSF}$ graph of Figure 1

$\mathcal{RDF}$ **format** Just as we make explicit the relationship between the labeled-graph representation of a ψ-term and an $\mathcal{RDF}$ representation for it on the example shown in Figure 14, we must also define its corresponding serialization syntax in $\mathcal{RDF}$.[8] It is given, for that same example, in Figure 15. The pseudo-code performing this is given as Algorithm 9.

Because the relation between a ψ-term and its $\mathcal{RDF}$ representation is formally a structure homomorphism, a bi-directional translation can be realized as a pair of mappings, each definable as a structure-directed recursive algorithm. The first one takes the in-

[8] `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/`

```
procedure PsiTermToRdf (Tag tag, Set dejavu, Array xmlArray)
    tag ← tag.deref();
    String xml ← "<rdf:Description";
    if tag ∈ dejavu then
        xml ← xml + " rdf:resource=\"";
        xml ← xml + "#" + tag.name + "\"/>";
        xmlArray.add(xml);
    else
        if not tag.isAnomymous() then
            xml ← xml + " rdf:about=\"";
            xml ← xml + tag.name + "\"";
        end
        xml ← xml + ">";
        xmlArray.add(xml);
        xml ← "<rdf:type rdf:resource=\"";
        xml ← xml + tag.term.sort + "\"/>";
        xmlArray.add(xml);
        dejavu.add(tag);
        foreach ⟨feature,subtag⟩ ∈ tag.term.subterms do
            switch type-of(subtag.term.sort) do
                case integer
                    xmlArray.add("<" + feature
                    + "rdf:datatype=\"&xsd;integer\"" + ">"
                    + subtag.term.sort + </" + feature + ">");
                end
                ⋮
                otherwise
                    xmlArray.add("<" + feature + ">");
                    PsiTermToRdf (subtag,dejavu,xmlArray);
                    xmlArray.add("</" + feature + ">");
                end
            endsw
        end
        xmlArray.add("</rdf:Description>");
    end
end
```

Algorithm 9: From ψ-term syntax to $\mathcal{RDF}$ as an array of $\mathcal{XML}$ strings

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:osfex="http://cedar.liris.cnrs.fr/ns/osfex#">
  <rdf:Description rdf:about="P">
    <rdf:type rdf:resource="osfex:person"/>
    <osfex:id>
      <rdf:Description>
        <rdf:type rdf:resource="osfex:name"/>
        <osfex:first>
          <rdf:Description>
            <rdf:type rdf:resource="osfex:string"/>
          </rdf:Description>
         </osfex:first>
         <osfex:last>
          <rdf:Description rdf:about="S">
            <rdf:type rdf:resource="osfex:string"/>
          </rdf:Description>
         </osfex:last>
       </rdf:Description>
     </osfex:id>
     <osfex:age rdf:datatype="&xsd;integer">30</osfex:age>
     <osfex:spouse>
       <rdf:Description>
         <rdf:type rdf:resource="osfex:person"/>
         <osfex:id>
           <rdf:Description>
             <rdf:type rdf:resource="osfex:name"/>
             <osfex:last>
               <rdf:Description rdf:resource="#S"/>
             </osfex:last>
           </rdf:Description>
         </osfex:id>
         <osfex:spouse>
           <rdf:Description rdf:resource="#P"/>
             </osfex:spouse>
           </osfex:Description>
         </osfex:spouse>
       </rdf:Description>
     </osfex:spouse>
   </rdf:Description>
 </rdf:RDF>
```

Figure 15: $\mathcal{RDF}$ serialization of the ψ-term in Figure 2

ternal structure of a root tag of a ψ-term structure as defined above and generates an array of strings using a method `PsiTerm.toXmlArray(Set dejavu)`, each part of the $\mathcal{XML}$ rendition of the equivalent $\mathcal{RDF}$ structure. The other taking an $\mathcal{XML}$ structure corresponding to a ψ-term into this ψ-term's actual $\mathcal{OSF}$ graph form as internal structure representation.

We can also define a mapping from either representation to ψ-term *syntax*, than can be parsed into its structure—*i.e.*, a `Tag.toString()` method. It can also be used as a format to save on secondary storage. Note incidentally that a similar `XmlPsiTerm.toString(Set dejavu)` method is not needed if we have a `XmlPsiTerm.toPsiTerm(Set dejavu)`. Note also that the **switch/case** statement on Line 20 of Algorithm 9 assumes a function `type-of` which applies to a ψ-term and returns its type among predefined built-in types (`integer`, `float`, `string`, ..., or `object`—the default **otherwise** case).

4.5.3 Feature consistency and query optimization

This is a simple but powerful static query optimization done on the $\mathcal{OSF}$ formulation of a query in $\mathcal{OSF}$ syntax, prior to being translated into $\mathcal{SPARQL}$ form. It consists in a feature-type verification realized as a normalization of the query using the rules of Figure 12 and Figure 8. In effect, it is a semantic type checking of consistency of all features involved in a query with their declared domain/range sorts, inferring missing sorts.

This semantic type inference may involve changing the query feature domain or range type. This is a type coercion by intersection, a *narrowing* correction of the missing or less precise type information specified in the query. This is by intersecting the known (from TBox) and implicit (in query) type information using taxonomic knowledge from the TBox. This is because it is both a type-checking and a type-inference process.

Once classification is performed, the encoded ontology is saved on disk once and for all. There are three important advantages for proceeding so:

1. a saved classified ontology can be reused without the need to be reclassified for every new query sessions;
2. checking a query's consistency with respect to a classified ontology before executing it prevents useless scanning of the ABox for instance retrieval if the query is not consistent; and,
3. normalizing a query with respect to a classified ontology drastically reduces the ABox retrieval search space focusing only on relevant instances.

Normalization of queries *w.r.t.* a normalized consistent $\mathcal{OSF}$ taxonomy is therefore a step that should be performed prior to submitting queries for execution—*i.e.*, before actual ABox instance retrieval. This has for effect to focus data retrieval.

Let us illustrate this on an example. Figure 16 shows an ontology describing academic workers and institutions. A bit-vector encoding of this partial order is given in Table 1.

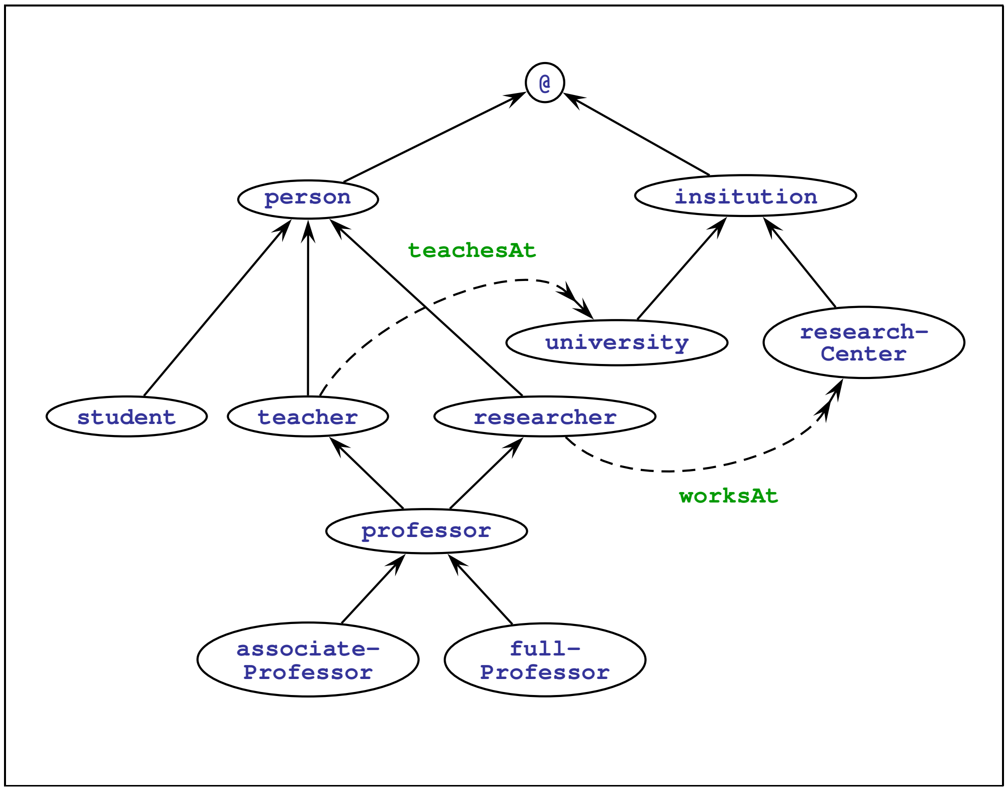

Figure 16: Example of a taxonomy with attribute features

Sort	**Code**
`@`	11111111111
`institution`	01110000000
`university`	00100000000
`researchCenter`	00010000000
`person`	00001111111
`researcher`	00000100111
`teacher`	00000010111
`student`	00000001000
`professor`	00000000111
`associateProfessor`	00000000010
`fullProfessor`	00000000001

Table 1: Binary codes for the poset shown in Figure 16

Besides the concept taxonomy, let us assume the two following set-valued feature declarations:

`teachesAt` : `teacher` $\to$ ***`setOf`***(`university`)

`worksAt` : `researcher` $\to$ ***`setOf`***(`researchCenter`)

Let us now consider the query Q_1 corresponding to the ψ-term:

Query Q_1:

```
?X : person ( worksAt ⇒ setOf(researchCenter)
            , teachesAt ⇒ setOf(university)
            ).
```

This query aims to retrieve all instances of persons teaching at a university, and working at a research center as well. Rather than submitting this query as is for retrieving instances from the ABox that verify it, we first normalize it to be consistent with the knowledge in the TBox. Doing so, we identify the sort `professor` as being the most specific one. In fact, the sort corresponding to `professor` is the Maximal Lower Bound (MLB) of the root sort (`person`) and the sorts which are the domains of the two features presents in Q_1; namely, `researcher` and `teacher`. The MLB is the intersection (conjunction) of the binary code corresponding to `person`, `teacher` and `researcher` which are represented by `00001111111`, `00000010111`, and `00000100111`. The result of that intersection is `00000000111`, which corresponds to the sort `professor`. Thus, the normalized query is Q'_1:

Normalized Query Q'_1:

```
?X : professor ( worksAt ⇒ setOf(researchCenter)
               , teachesAt ⇒ setOf(university)
               ).
```

Following the same reasoning, normalizing the query Q_2 expressed as the ψ-term:

Query Q_2:

```
?X : student ( worksAt ⇒ setOf(researchCenter)
             )
```

yields the inconsistent (*i.e.*, empty) sort represented by the code `00000000000`. In this case, the query is considered inconsistent with the TBox because the sort `student` has no subsort that is compatible with a known domain for the feature `worksAt`. Thus, there is no need to search the ABox for any instance of this query.

4.5.4 $\mathcal{SPARQL}$ query generation

Once a query expressed as in $\mathcal{OSF}$ syntax is normalized with respect to a TBox and found consistent, it is compiled into $\mathcal{SPARQL}$ for efficient instance retrieval. The translation is straightforward, and need not even be detailed as pseudo-code. Examples will suffice.

Figure 17 shows the $\mathcal{SPARQL}$ query corresponding to the query Q_1 without prior normalization, and Figure 18 shows the $\mathcal{SPARQL}$ query for the same query after normalization.

```
SELECT  ?X
 WHERE
     {
        ?X  rdf:type   person.
        ?X  worksAt    ?Y.
        ?Y  rdf:type   researchCenter.
        ?X  teachesAt  ?Z.
        ?Z  rdf:type   university.
     }
```

Figure 17: Generated $\mathcal{SPARQL}$ from Query Q_1 (without normalization)

```
SELECT  ?X
 WHERE
     {
        ?X  rdf:type  professor.
     }
```

Figure 18: Generated $\mathcal{SPARQL}$ from Query Q'_1 (with normalization)

One can clearly see that the $\mathcal{SPARQL}$ query in the normalized format has many less constraints than the first one. Not only could the `rdf:type` of query variable **`?X`** be narrowed to the more specific sort **`professor`**, but also *the domain/range feature constraints could be eliminated altogether!* This is because they were already verified to be consistent by normalization, and since all instances in the ABox are necessarily consistent with the knowledge of the TBox (in the same manner as all data in a database obey its schema), it can be safely assumed that all relevant instances of sort **`professor`** in the ABox *already abide by those feature constraints!* This not only reduces the search space in the ABox, but also greatly improves query evaluation by removing useless costly joins. In addition, evaluating this query can be made even more efficient if a datatype indexing is already performed by the triplestore (see Section 4.5.5).

Note that it is not always possible to eliminate feature constraints from a generated $\mathcal{SPARQL}$ query. This is the case in particular when a feature constraint in a query specifies a *value* as opposed to a sort as the range of a feature. In that case, these features must be included in the generated $\mathcal{SPARQL}$ query. Consider for example:

Query Q_3:

```
?X : person ( school ⇒ "Stanford"
            )
```

(assuming feature declaration **`school`** : **`student`** $\rightarrow$ **`string`**). The generated $\mathcal{SPARQL}$ query shown in Figure 19 does normalize the sort **`person`** to **`student`**, but it must keep the feature **`school`** with specific value **`"Stanford"`**.

```
      SELECT  ?X
       WHERE
           {
              ?X  rdf:type  student.
              ?X  school    "Stanford"
           }
```

Figure 19: Generated $\mathcal{SPARQL}$ query from normalized Query Q_3 with valued feature

Finally, it is important to mention that for each sort occurring in the query, sending its binary code along with the query to the triplestore allows efficient filtering of eligible answer instances. For example, for the sort **`professor`**, the binary code `00000000111` allows to filter instances of the sorts **`associateProfessor`** and **`fullProfessor`** as eligible answers since their sorts are subsorts of **`professor`**. This is explained next.

4.5.5 TBox-based triplestore indexing

This describes a powerful static query optimization which is possible if $\mathcal{HOOT}$'s underlying triplestore management system offers the means to index its sets of triples to be queried using $\mathcal{HOOT}$'s bit-vector encoding of a concept taxonomy. Indeed, a simple type-indexing scheme taking advantage of the bit-vector encoding can focus the retrieval of triples strictly and only on the concerned ABox instances.

This is achieved as follows. Organize the ABox in memory so that triples of a given sort ("root" triples of this sort) are all stored contiguously. Then, in the taxonomy array containing each sort and its properties (such as name, binary code, *etc.*), add two integer fields: one indicating the ABox index of the first triple of this sort, and the other indicating the last such index. In this way, it is possible to iterate only over those triples in the ABox sorted with subsorts of a given query simply by using the binary code of the query's root sort. This is possible since its "`1`" bits' positions correspond to the indices of its subsorts in the taxonomy array [5]. Thus, the relevant ranges of triples stored at these indices in this array are readily accessible. Such an indexing scheme is illustrated in Figure 20.

Of course, when used, indexing is performed once and for all as an offline step, to be reused on the same ABox as often as needed. Using such a scheme, we could verify experimentally that performance of query processing could be further divided by a factor in the order of thousands [10]. This indicates that building type-indexing using encoded sorts in an actual triplestore management system is likely to provide similar results—even if lessened to a factor of tens for complex queries.

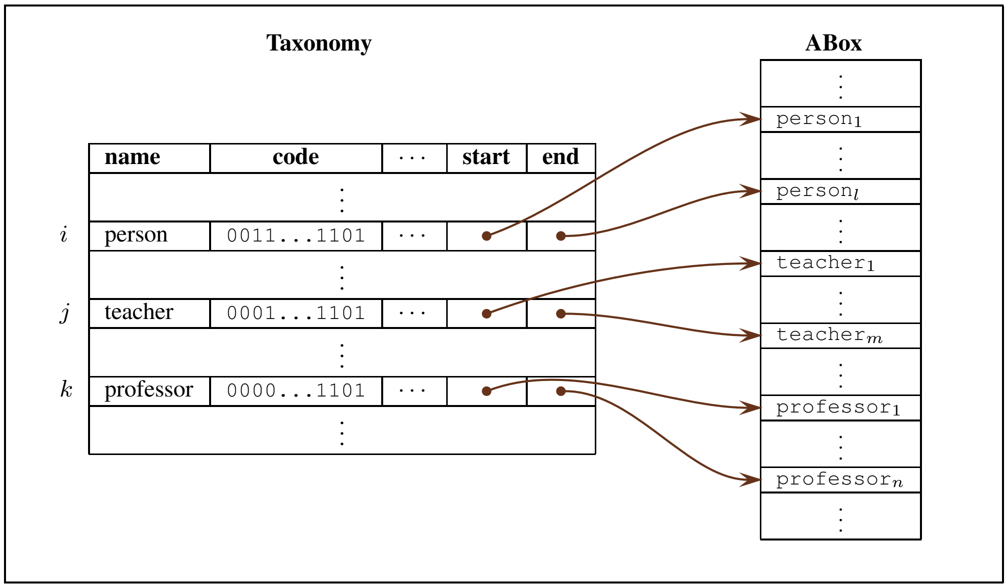

Figure 20: Illustration of ABox-indexing scheme using TBox sort encoding

5 Conclusion

This document has justified the design of, and detailed an implementation specification for, $\mathcal{HOOT}$, a computer language designed for processing taxonomic attributed ontologies. We put this work in the context of the Semantic Web. It is a contribution in the nascent area of the intelligent processing of distributed knowledge and data represented as $\mathcal{RDF}$ triples. An initial prototype of this specification was realized as the $\mathcal{CEDAR}$ reasoner.[9] Initial performance comparison tests with the state of the art show that the ideas developed in this specification are valuable in boosting TBox and ABox querying by orders of magnitudes [10].

This initial design is meant as a starter. Later versions of $\mathcal{HOOT}$ can be developed to support more sophisticated $\mathcal{OSF}$ theories (ontologies of concepts abiding by rule-defined constraints), focusing always on efficiency. In the eventuality that such theories may lead to undecidable inference, we will stay on the pragmatic side and opt for an incomplete but efficient inference system. Indeed, this is a trade-off worth making as it enhances semantic expressiveness while retaining efficient reasoning—just as Prolog is *w.r.t.* first-order Horn logic: while semi-decidable it can be implemented efficiently, and has much greater expressive power than plain (decidable, but NP-complete) Propositional Logic.

References

[1] Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures*. PhD thesis, Technical Report MS-CIS-84-62,

[9] http://cedar.liris.cnrs.fr/cedar-software/cedar/

Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA, September 1984. [ACM reference[10]].

[2] Hassan Aït-Kaci. *Warren's Abstract Machine—A Tutorial Reconstruction*. Logic Programming. MIT Press, Cambridge, MA, USA, 1991. [See online[11]].

[3] Hassan Aït-Kaci. Data models as constraint systems: A key to the semantic web. *Constraint Processing Letters*, 1:33–88, November 2007. [See online[12]].

[4] Hassan Aït-Kaci. A set-complete domain construction for order-sorted set-valued features. CEDAR Technical Report Number 11, CEDAR Project, LIRIS, Département d'Informatique, Université Claude Bernard Lyon 1, Villeurbanne, France, October 2014. [See online[13]].

[5] Hassan Aït-Kaci and Samir Amir. Classifying and querying very large taxonomies—a comparative study to the best of our knowledge. CEDAR Technical Report Number 2, $\mathcal{CEDAR}$ Project, LIRIS, *Département d'Informatique, Université Claude Bernard Lyon 1*, Villeurbanne, France, May 2013. [See online[14]].

[6] Hassan Aït-Kaci, Robert Boyer, Patrick Lincoln, and Roger Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages and Systems*, 11(1):115–146, January 1989. [See online[15]].

[7] Hassan Aït-Kaci and Roger Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986. [See online[16]].

[8] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3-4):195–234, 1993. [See online[17]].

[9] Hassan Aït-Kaci, Andreas Podelski, and Seth C. Goldstein. Order-sorted feature theory unification. *Journal of Logic Programming*, 30(2):99–124, 1997. [See online[18]].

[10] Samir Amir and Hassan Aït-Kaci. Design and implementation of an efficient semantic web reasoner. CEDAR Technical Report Number 12, $\mathcal{CEDAR}$ Project, LIRIS, *Département d'Informatique, Université Claude Bernard Lyon 1*, Villeurbanne, France, October 2014. [See online[19]].

[11] Ronald J. Brachman. *A Structural Paradigm for Representing Knowledge*. PhD thesis, Artificial Intelligence, Harvard University, Cambridge, MA, USA, 1977. Available as BBN Technical Report 3605 from Bolt Beranek and Newman Inc. (1978).

[10] `http://dl.acm.org/citation.cfm?id=911837`
[11] `http://wambook.sourceforge.net/`
[12] `http://hassan-ait-kaci.net/pdf/cpl-article.pdf`
[13] `http://cedar.liris.cnrs.fr/papers/ctr11.pdf`
[14] `http://cedar.liris.cnrs.fr/papers/ctr2.pdf`
[15] `http://hassan-ait-kaci.net/pdf/encoding-toplas-89.pdf`
[16] `http://hassan-ait-kaci.net/pdf/login-jlp-86.pdf`
[17] `http://hassan-ait-kaci.net/pdf/meaningoflife.pdf`
[18] `http://hassan-ait-kaci.net/pdf/osf-theory-unification.pdf`
[19] `http://cedar.liris.cnrs.fr/papers/ctr12.pdf`

Technical Report Number 16

$_{\mathcal{H}}\boldsymbol{\mathcal{OO}}_{\mathcal{T}}$

A Language for Expressing and Querying
$\mathcal{H}$ierarchical $\mathcal{O}$ntologies, $\mathcal{O}$bjects, and $\mathcal{T}$ypes

Hassan Aït-Kaci

December 2014