HAL
open science

Design and Implementation of a Type System for a Knowledge Representation System

Cécile Capponi

To cite this version:

Cécile Capponi. Design and Implementation of a Type System for a Knowledge Representation System. RR-3096, INRIA. 1997. inria-00073595

HAL Id: inria-00073595

https://inria.hal.science/inria-00073595v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Design and Implementation of a Type System for a Knowledge Representation System

Cécile Capponi

N° 3096

Janvier 1997

THÈME 3

Rapport de recherche

ISSN 0249-6399

Design and Implementation of a Type System for a Knowledge Representation System

Cécile Capponi

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Sherpa

Rapport de recherche n° 3096 — Janvier 1997 — 32 pages

Abstract: A knowledge representation system (KRS) is made up of both a language to represent knowledge of a domain and well-defined reasoning facilities to infer new knowledge from known facts. This paper deals with KRSs close to frame-based systems, that include description logic systems and object-based systems. In these systems, the main relation that leads to inferences is *subsumption*. Knowledge terms are described through roles which refer to either other knowledge terms or data types. Subsumption between term descriptions is usually interpreted as data set inclusion, where data is either a knowledge term or an external term (integer, string, etc.). Although subsumption between knowledge terms is well-defined, its implementation on external data depends upon the host language since there are actually the data types of the KRS. As a consequence, no KRS is able to integrate a new data type (*e.g.* Matrix) such that its values can be safely involved in subsumption and further inferences. This is the problem addressed in this paper. The proposed solution is the design of a polymorphic type system connected to both the KRS and the host language. It is designed so that it can extend the KRS with any data type implementation available in the host language (library, user-coded). Meanwhile, the values of the new data type get safely involved in the KRS reasoning processes. The presented type system avoids the incompleteness of subsumption due to its incomplete processing on external data.

Key-words: Object-Based Knowledge Representation, Subsumption, Data Types, Type Checking, Sub-typing.

(Résumé : tsvp)

* Email : `Cecile.Capponi@inrialpes.fr`. Projet SHERPA, INRIA Rhône-Alpes, France. This work has been partly carried out at Intelligent Software Group, Simon Fraser University, Burnaby (B.C.), Canada.

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 04 76 61 52 00 – Télécopie : (33) 04 76 61 52 52

Conception et Réalisation d'un Système de Types pour un Système de Représentation des Connaissances

Résumé : Un système de représentation des connaissances (SRC) est constitué, d'une part d'un langage pour représenter et structurer les connaissances d'un domaine, et, d'autre part, de processus de raisonnement bien définis pour l'inférence de nouvelles connaissances à partir de celles déjà représentées. Ce rapport traite des SRCs issus des systèmes de *frames*, tels que les systèmes de descriptions logiques ou les systèmes à base d'objets. La relation principale qui mène à des inférences y est la *subsomption*. Les termes de connaissances sont décrits par le biais d'attributs qui font référence, ou bien à d'autres termes de connaissances, ou bien à des types de données externes. La subsomption entre descriptions de termes est souvent interprétée comme l'inclusion ensembliste, où les ensembles considérés peuvent contenir à la fois des termes de connaissances et des termes externes (entier, chaîne, etc.). Bien que la subsomption entre termes de connaissances soit bien définie dans le SRC, sa réalisation sur des données externes dépend du langage hôte puisque les types de données externes utilisés dans le SRC sont généralement implantés dans ce langage. En conséquence, aucun SRC n'est actuellement capable d'intégrer de nouveaux types de données (ex. Matrice) de telle façon que leurs valeurs puissent être correctement impliquées dans la subsomption, et, en aval, dans les inférences. Il s'agit du problème traité dans ce rapport. La solution proposée consiste à concevoir un système de types polymorphe couplé d'un côté au SRC (pour le typage des termes de connaissances), et de l'autre côté au langage hôte (pour l'importation contrôlée de types de données externes). Grâce à ce système, des ensembles de données pourront être construits en langage du SRC à partir de valeurs et opérateurs de tout type de données importé du langage hôte ; puis ces ensembles et leurs valeurs pourront être correctement et complètement impliqués dans les processus de vérification et d'inférence du SRC par délégation au système de types. Ce résultat obtenu grâce au développement du système de types permet d'éviter l'incomplétude de la subsomption inhérente à son traitement sur des données externes dont l'implémentation était jusqu'à présent une boîte noire.

Mots-clé : Représentation des connaissances par objets, Subsomption, Types de données, Vérifications de types, Sous-typage

1 Introduction

The development of knowledge representation systems (KRSs) aims at helping people in an application domain to both organize and describe the knowledge related to this domain. This requires the development of expressive representation languages. Description logic systems (DLSs, also called terminological languages, or KL-ONE family [BS85]) are close to object-based languages, because they both allow one to identify and describe knowledge as groups and individuals.

1.1 Why do we need a new kind of type system ?

The problem we address is to provide a knowledge representation system with an extensible collection of computable data types which can be used as parts of knowledge descriptions. Although our solution is drawn for object-based KRSs, it can be easily extended to any KRS stemming from either frame systems (*e.g.* KRS [BW77]) or semantic networks [Woo75]. The solution can be especially adapted to DLSs which also have to deal with a restricted collection of data types.

One may figure out that, in order to provide a KRS with an extensible type collection, it would be sufficient to connect to the KRS an extensible type system which already exists in the field of programming languages (*e.g.* this of ADA [Uni83]). Such a solution is not suitable though, because of the following specificity of KRSs: subsets of values of any available data type are constantly constructed — or infered during the resolution of constraint satisfaction problems — within a knowledge base. For example in the simplest case, the *age* of a person is an integer ranging between 0 and 150, domain of values which most likely is to be compared with this of the *age* of a child, *i.e.* an integer that ranges between 0 and 14. These subsets of values are then involved in type checking and inference processes which rely on set-based operations. For example, the KRS requires the type system to be able to compute the intersection (or inclusion, disjunction, etc.) of two lists of intervals of values that belong to any totally ordered data type. Yet lists of intervals are the simplest kind of expressions to deal with. Moreover, there exist many ways in a KRS to express the same subset of values because of the richness of the knowledge representation language. This compels the system to deal with normal forms of subsets of values.

As far as we know, no extensible type system designed for a programming language — or for a database system — is able to manage subsets of values which are explicitly and declaratively maintained. Since here is the main requirement of a KRS with regards to type checking, it turns out that it is relevant to design a type system which could meet that demand. Obviously, if a KRS does not need neither operations nor checking among represented subsets of values, the type system we present hereafter is not of interest for such a KRS.

1.2 Where do types arise in knowledge representation ?

Basically, an object-based knowledge representation system (OBKRS) is a frame system [KLW95] that explicitly differentiates groups from individuals. In OBKRSs, a frame that corresponds to an individual is named an instance, and a frame that corresponds to a group of individuals is named a class. A frame is described by means of attributes which are intended to characterize the groups and individuals.

In DLSs, groups and individuals are called (generic) concepts and individual concepts, respectively. They are described through roles (binary relationships that link two concepts), or attributes (to relate a concept to a data type or a basic value). For example, *child* `is-a` (`and` (*person*) (*age* `integer`) (`max` *age* 12) (`all` *friends child*)) means that a child is a person under the age of 12 and whose friends are only children (*child* and *person* are concepts, *friends* is a role, *age* is an attribute and `integer` is a data type). In the following, the *type collection* of a KRS means the set of all data types available to be associated with attributes, which is usually restricted to basic data types.

Concepts can also be linked through subsumption. There exist several definitions of subsumption between two concepts, whether we consider either extensions (sets of elements) of concepts, or their

intensions (descriptions) [Woo91]. Intuitively, concept A (*e.g. person*) subsumes concept B (*e.g. child*), written $B \leq A$, if all individuals of B are also individuals of A (*e.g.* any child is a person). This intuitive meaning of subsumption requires the satisfaction of formal inclusive conditions on involved term descriptions, leading to the definition of the so-called *intensional subsumption*. If $B \leq A$, intensional subsumption requires B to be described at least with the same roles and attributes as A, plus possibly some others, and attributes and roles in B must be more specialized than in A (*e.g.* the range of *age* in *person* includes the range of *age* in *child*). Therefore, the subsumption test between two given concepts is projected on their roles and attributes. Subsumption between roles is a system-defined process that operates on terms. Its completeness thus depends on the richness of the knowledge representation language. Although subsumption between attributes is usually predefined in the KRS, its implementation still depends on the host language that provides the KRS with data types. Usually, the host language is the implementation language of the KRS, which is distinguishable from the knowledge representation language used to describe knowledge. Since the implementation of subsumption eventually depends on the structure of host language data types, the KRS cannot dynamically integrate a new host data types unless there exists a way for the KRS to automatically generate the code of subsumption among subsets of values of the new data type. This functionnality does not exist in any KRS so far except in the KRS system [Gai93].

Intensional subsumption in DLSs applies between any kind of concepts, either generic or individual, whereas in OBKRSs subsumption is split between two relations, namely specialization (between two classes: inclusion) and attachment (between an instance and a class: membership). In the following, susbumption may stand for both specialization and attachment.

Several KRSs allow the user to add new data types in the type collection. This is achieved by a user-defined predicate between values and the new type (TEST-H in CLASSIC [BMPS+91], `:predicate` in LOOM [ISX91]). This predicate is then used in the description of attributes. For example in CLASSIC, the term (TEST-H `evenp`) is the concept `EVEN` that denotes the set of all even numbers, where `evenp` is a LISP predicate. Subsumption however cannot be performed on attributes related to the new data type, so it cannot be completely checked between concepts if such an attribute is a part of their description. For example, there is no way for CLASSIC to statically detect the inconsistent term (`and STRING EVEN`) which may lead to further inconsistencies, although only this syntax allows one to relate two data types in the knowledge base. This is related to the problems of static consistency checking arising whenever one tries to build a predicate hierarchy. As a consequence, subsumption is incomplete as soon as a new data type occurs in a description [RBB+93]. Such an incompleteness does not depend on the expressiveness of the knowledge representation language, but on the fact that the KRS cannot reason about the bodies of data type definitions for they are written in the host language. It is not important as long as the application domain does not require unavailable data types to describe knowledge. However, some application domains may need some complex data structures to be fully involved in reasoning processes. For example molecular biology requires the data constructor sequence as a basis to represent the computable features of genomic sequences. One may figure out that those complex data structures can be implemented using the knowledge representation language, although this solution is too complex, and also unsuitable for the knowledge representation language. Moreover, the processes of the KRS are not suited to fully and efficiently integrate executable programs.

The purpose of this work may be compared to any other KRS extensibility study that means to improve its expressivity and genericity power. In this connection, PROTODL provides a way to extend the vocabulary of the representation language [BB92] and the system KRS is designed and implemented such that new functionalities may be easily added [Gai93].

The problem of the extensibility of the KRS type collection could be solved by providing the system with the capacity of dynamically building a safe predicate hierarchy, where each predicate is associated with functions which apply to its value and which manage subsets of its values (inclusion, membership, etc.). This solution is equivalent to the construction of a dynamic data type hierarchy, where each

data type encapsulates not only the construction, properties and operations of its values, but also the management of its powersets.

1.3 Guidelines

We propose here to solve this problem of the extensibility of the type collection, in the particular case of OBKRSs. For this purpose, a structure is designed which is connected to both the KRS and the host language. This structure should dynamically provide the KRS with all type operations required by subsumption, namely subset inclusion and subset intersection. This structure is a type system; it is a bridge between data type implementations (host language) and subsumption checking processes among attributes and thus among classes. We show how each knowledge entity is associated with a type (or a value) in the type system so that subsumption is performed on types according to a common set-based semantics, whether types are external or related to knowledge entities.

The paper is organized as follows: in section 2 we introduce the main features of OBKRSs, which are used further on in section 3 to identify the ways data types are processed in knowledge bases. The presented results are important to sketch the type system satisfying our purpose. In the section 4, we present the main organization of the type system METEO we designed for OBKRSs. The main feature of METEO is its two-level organization, corresponding to the distinction *data type / subset of data* that is kept up in knowledge base descriptions. The first level of METEO is directly connected to the host language; the second level is issued from the typing of knowledge entities. Both levels are related and cooperate as well. We then present how METEO may be easily and dynamically used to extend the type collection of the KRS. Section 5 outlines an example of use of METEO in the domain of molecular biology. Finally, in section 6, we compare our study with previous results in the area of DLSs.

2 Object-based knowledge representation systems

Object-based knowledge representation systems (OBKRSs) stem from frame-based systems, except that they distinguish between frames that denote individuals (instances) and frames that denote sets (class).

2.1 Preliminaries

A knowledge base is made of several families. A *family* represents a set of real-world individuals it divides up among groups (*e.g.*). Groups are implemented as *classes*. Classes of the same family are ordered by *specialization* that represents inclusion relationships among sets of individuals which the classes denote in the application domain. When a class C specializes a class C' (the set denoted by C is a subset of the set denoted by C'; C is then a sub-class of C' and C' is a super-class of C). As inclusion, specialization is transitive. Specialization may be either single or multiple according to the number of direct super-classes a class is allowed to have. An individual is implemented as an *instance*, which belongs to a class of its family or possibly to several classes when multi-instantiation is allowed in the KRS. The relation between an instance and a class is called *attachment*.

Classes and instances are described by means of *attributes*. An attribute represents one particular characteristic some instances of a family have.

Each main feature offered by an OBKRS to organize knowledge can be associated with a corresponding feature in description logic systems (DLSs). Indeed, both allow users to identify and describe knowledge as individuals which belong to one or several groups, where the groups are partially ordered according to set inclusion. Organizing knowledge according to such a frame leads to the common identification of both kinds of systems as *classification-based systems* [Mac91a].

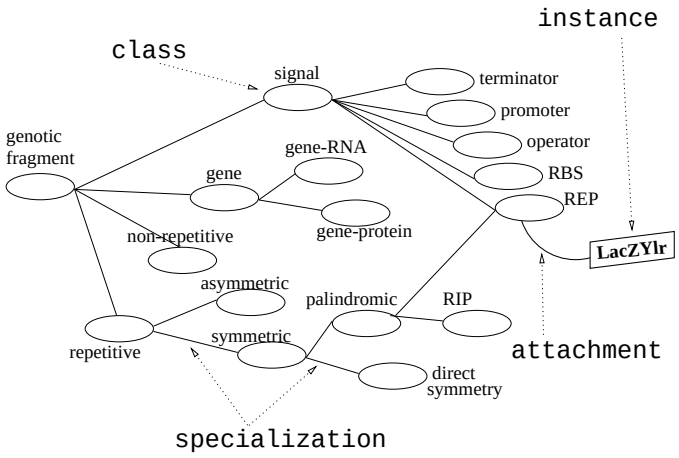

Figure 1: A class graph in the field of molecular biology. Specialization is a transitive relation, therefore the instance *LacZYlr* belongs to all super-classes of *REP*, such as *signal* or *symmetric*.

2.2 Knowledge entities

We talk about (knowledge) entities denoting classes, instances, and attributes: these are the three main structures used to describe knowledge in an OBKRS.

Each entity is implemented as a structure which is made up of an identifier and which is allowed to access by itself all the information it contains. Such a structure has two parts: it carries the *extension* of the entity it implements and its *intension*. When the entity is a class, its extension is the set of individuals it denotes; when the entity is an instance, its extension is the individual of the real world it represents. In both cases, the intension is the description of the entity. Hence, an instance I is a double ⟨`identifier;description`⟩ where the identifier is the name of the instance, representing the denoted individual. A class C is a triplet ⟨{`identifier`;`identifiers`};`description`⟩, where `identifiers` are those of the instances that belong to the class and `identifier` is that of the class. The description of any entity is an attempt to represent the properties of the corresponding part of the entity in the application domain.

Class descriptions are made up of conjunctions of attribute descriptions. An attribute is a double ⟨`name;domain`⟩, where `name` often carries the meaning the attribute gets in the real world (application domain), and `domain` is merely a set expression. Hence, two attributes, sharing the same name with two different domains are two different restrictions on the same feature. An instance of a class contains a value (possibly unknown) for each attribute of the class it belongs to.

In class descriptions, attributes are firstly given an initial type. Then, their descriptions may be refined such that their domains are restricted. The domain of an attribute is described by a conjunction of descriptors. A descriptor is a double ⟨`nature;expression`⟩, where `expression` is a set expressed over a syntax that depends on the `nature` of the descriptor. The attribute description below

attribute-name `set integer card` [6;9] `among` [1;49] `except` (1, 2, 3, 4, 5, 6)

denotes the whole set of sets containing between six and nine integers ranging from 1 to 49, excluding the set (1 2 3 4 5 6). Such a description could for example match this of the bet numbers of a *play* (or the result of a *draw*) at the 6/49 lottery.

Table 1 lists different kinds of descriptors which can be found in OBKRSs in order to describe class attributes. Only type descriptors are presented. The descriptor `compute-with-filter` (section 2.4) not only defines an inference mechanism, but it also restricts the type of the involved attribute, although we do not handle that possibility in this paper.

nature	**syntax**	**semantics**
attribute descriptors: specification of the attribute initial type		
`a`	$T \mid C_1, \cdots, C_n$	$\|\|T\|\| \mid \|\|C_1\|\| \cap \cdots \cap \|\|C_n\|\|$
`list`	$T \mid C_1, \cdots, C_n$	$\lambda^{\|\|T\|\|} \mid \lambda^{(\|\|C_1\|\| \cap \cdots \cap \|\|C_n\|\|)}$
`set`	$T \mid C_1, \cdots, C_n$	$2^{\|\|T\|\|} \mid 2^{(\|\|C_1\|\| \cap \cdots \cap \|\|C_n\|\|)}$
attribute descriptors		
`domain`	$v_1, \cdots, v_n$	$\{v_1, \cdots, v_n\}$
`except`	$v_1, \cdots, v_n$	$\mathcal{U} \backslash \{v_1, \cdots, v_n\}$
`value`	v	$\{v\}$
attribute descriptors, when initial type is totally ordered		
`intervals`	$i_1, \cdots, i_n$	$\|\|i_1\|\| \cup \cdots \cup \|\|i_n\|\|$
`except`	$i_1, \cdots, i_n$	$\mathcal{U} \backslash (\|\|i_1\|\| \cup \cdots \cup \|\|i_n\|\|)$
attribute descriptors, when initial type is multi-valued (prior application of either `list` or `set`)		
`among`	$v_1, \cdots, v_n$	$\gamma^{\{v_1, \cdots, v_n\}}$
`forbidden`	$v_1, \cdots, v_n$	$\mathcal{U} \backslash \gamma^{\{v_1, \cdots, v_n\}}$
`cardinal`	$[e_1, e_2]$	$\{s = \{v_i\} \mid \mathrm{card}(s) \in [e_1; e_2]\}$

Table 1: Attribute descriptors. Let $\|\|S\|\|$ be the subset of values of $\mathcal{U}$ denoted by the symbol S; $\gamma \in \{2, \lambda\}$; symbols v, v_i refer to values of $\mathcal{U}$; symbols i, i_i refer to intervals; T is an element of the type collection of the KRS, that is an available data type; symbols C_i refer to classes.

Entities of the knowledge base are given a denotational semantics. The interpretation domain is $\mathcal{U}$; it contains all type values and instances, which are unit elements, as well as lists and sets constructed from unit elements. Let us consider an attribute $\langle a; d \rangle$, where $d = \delta_1, \cdots, \delta_n$ (each δ_i is a descriptor). The denotation of the attribute in $\mathcal{U}$ is:

$$\|\langle a; d \rangle\| = \bigcap_{i \in [1;n]} \|\delta_i\|$$

(see table 1 for descriptor semantics).

The description of an instance is a conjunction of valued attributes, it is written $I = \{\langle a_i; v_i \rangle\}_{i \in [1;n]}$. The description of a class is written $C = \{\langle a_i; d_i \rangle\}_{i \in [1;n]}$. The denotation of the class C in $\mathcal{U}$ is:

$$\|C\| = \{I = \{\langle a_i; v_i \rangle\}_{i \in [1;m]} \in \mathcal{U} \mid m \leq n \text{ and } \forall i \in [1; m], v_i \in \|d_i\|\}$$

The description below is this of a class that is intended to group all the plays of a 6/49 lottery draw of a given day:

```
Class today-play-6/49
    Attributes:
        numbers set integer card [6;9] among [1;49]
        player in Person
            compute-with-filter { P Person,
                                  P.age ≥ P.citizenship.majority }
        outlay in integer interval [1;+∞]
        date in Date value (12,03,1996)
        won in integer interval [0;+∞]
```

where *Person* is another family of the knowledge base grouping all human individuals, `Date` is a data type which may exist in the type collection of the KRS; `compute-with-filter` is both a way to retrieve

instances of *Person* that satisfy the condition about the authorized age to play at the lottery, and a way to restrict — by the expression of a subset of potential instances of *Person* — the initial type of the attribute *player*.

Attributes may be either complex or elementary. An elementary attribute (*e.g. numbers* or *date*) gets its values from an external data type. A complex attribute (*e.g. player*) indicates that its values are instances of another family of the knowledge base (*e.g. Person*). Recursive descriptions, that is a class referring to itself in its description [Neb91], are not allowed in the kind of OBKRSs we consider in our study.

DLSs share many features of OBKRSs, although both the terminology and the internal implementation of entities are different. Generic concepts of DLSs are close to classes in OBKRSs. Both generic concepts and classes are supposed to capture sets of individuals (named "individual concepts" in DLSs) which are described through attributes (named roles in DLSs). A class (or an instance) in OBKRS, unlike a concept in DLSs, is defined as an object, therefore its description is encapsulated within the object. Concept descriptions in DLSs are distributed among two cooperative but different languages, namely the TBox (terminology) and the ABox (assertions) [BFL83].

2.3 Relations between entities

Two general relations exist among entities. They deal with a specific way of organizing knowledge that is usual in many application domains. Each relation has a meaning in the application domain and it induces conditions among entity descriptions.

Attachment is a binary relation, written $\in_\alpha$, defined between an instance and a class. It means that the individual the instance represents (*e.g.* Chet Baker) belongs to the group which the class denotes in the application domain (*e.g.* Trumpeters). The intensional attachment is defined on descriptions of involved entities. The restrictions on descriptions are necessary conditions for the whole attachment to be agreed upon in the knowledge base. Let $I = \{\langle a_i; v_i\rangle\}_{i\in[1;m]}$ be an instance. Let $C = \{\langle a_i; d_i\rangle\}_{i\in[1;n]}$ be a class. Then the attachment between two entity descriptions satisfies the following requirement:

$$i \in_\alpha C \Longrightarrow m \leq n \text{ and } \forall i \in [1;m] : v_i \in d_i$$

For example, the instance description below

```
Instance my-play
        Attributes:
            numbers =  { 12 5 32 31 7 9 }
            player =  Person#12008
            outlay =  4
            date =  (12,03,1996)
            won =  0
```

satisfies the conditions of the description of the class *today-play-6/49*, because each value of an attribute given in the instance belongs to the domain of the attribute described in the class. Thus the instance *my-play* could be attached to the class *today-play-6/49* according to the entity descriptions.

Attachment is distinct from instantiation as considered in any object language. Instantiation is the process that creates the data structure of an instance according to the data structure of a class. Attachment is a relation between any instance and any class, whatever their structures are. The distinction is essential in OBKRSs, because it permits migration of instances from one class to

another, without destroying the instance and having to re-create it. This distinction comes from the fact that an individual of the application domain (*e.g.* Chet Baker) exists independently of the groups of individuals it may belong to during its lifetime (*e.g.* Chet Baker starts his life as a child but fortunately he did not die when he became a teenager). Such considerations are close to these of G. Ghelli who insisted on the difference between both the type creation of a value and the actual type of this value, in order for the object migration process to get a well-defined semantics [Ghe90].

When multi-instantiation is allowed in a system, the above conditions are still necessary to check the attachment between an instance and each of the classes it may belong to. An alternative is, for each attribute a common to at least two classes of the instance, to compute the final domain of a as the result of the intersection of the different domains for a in each class. Thus, to be valid, the attachment relationship requires that the values of the instance belong to the intersected domains.

Specialization is a binary relation, written $\leq_\sigma$, defined between two classes. It means the inclusion of the group represented by the most specialized class (*e.g.* Trumpeters) in the group represented by the less specialized class (*e.g.* Musicians). As for attachment, intensional specialization leads to necessary restrictions on involved entity descriptions. Let $C_1 = \{\langle a_i; d_{1i}\rangle\}_{i\in[1;n_1]}$ and $C_2 = \{\langle a_i; d_{2i}\rangle\}_{i\in[1;n_2]}$ be two classes of a same family. Then specialization between class descriptions satisfies the following requirement:

$$C_1 \leq_\sigma C_2 \Longrightarrow n_2 \leq n_1 \text{ and } \forall i \in [1; n_2] : d_{1i} \subseteq d_{2i}$$

In the case of multiple inheritance (*e.g.* SHIRKA [RU91]), the sub-class must satisfy the above conditions for each specialization relationship with its super-classes. An alternative, close to that presented for attachment, is the computation of the intersection of all super-class descriptions, in order to check for inconsistency and/or to reduce the number of inclusion tests.

Specialization is a system-defined partial order. It is drawn as a class tree, or a graph in systems which permit multiple-specialization. The inheritance mechanism is processed along the class tree as a classic object-oriented way to factor the code out. Therefore, the complete description of an attribute in a class may be split among super-classes; the inheritance mechanism is used to retrieve its whole description. Some specific algorithms are studied in order to deal with multiple inheritance, such as linearization of the specialization order [DH91], or the renaming of attributes in case of a conflict. As attachment, specialization relationships regarding the descriptions may be inferred automatically.

In DLSS, both specialization and attachment are merged towards a unique relation, namely subsumption. As for specialization and attachment, intensional subsumption relationships may be inferred according to the concept descriptions available in the TBox. The definition of intensional subsumption is semantically equivalent to the definition of specialization between two class descriptions. Intensional subsumption inference or checking may be completed using terms of the ABox.

Some DLSS, such as BACK or CLASSIC allows the user to write rules [HKQ+93] in order to relate concepts independently of their descriptions. An equivalent relation exists in the OBKRS TROPES [Pro95], through the definition of points of view which cooperate by means of bridges [MRU90].

2.4 Inference mechanisms

Any OBKRS is provided with pre-defined inference mechanisms. They are activated either by the user or by another process, in order to deduce new knowledge according to already represented knowledge. Most of these inference mechanisms stem from frame language inference mechanisms, such as filters, procedural attachments, and default values. Yet the main inference mechanism is the so-called *classification*, also defined by DLSS.

Filters, procedural attachment and default are three inference mechanisms used to compute the value of an attribute for an instance, whenever the attribute is either elementary or complex. These inference mechanisms are called local inferences because they are attached to the attribute in the class, and are activated whenever the value of the attribute is unknown but needed. Both the activation of each of these mechanisms and the integration of their results in the knowledge base are totally controlled by the OBKRS. Among other conditions, the result must belong to the domain of the attribute the local inference mechanism is attached to. Relevant information about filters may be found in [Dek94]; [Win85, Rec93] contain information about procedural attachment; default are presented in [Bra85, Pag92, RN87] among others. Examples of filters (`compute-with-filter`), procedural attachments (`compute-with`), and default values (`default`), are given in the description of the class *protein-gene*, section 5.

Classification of an instance [Nap92, SL83, Mac91b] down a class hierarchy is aimed at finding the most specialized class the instance matches with. It is an inference process, because the more the class is specialized, the more there is knowledge about the individuals it groups. Hence, the lower an instance is attached to a class of the hierarchy, the more the user is provided with information about that instance. Instance classification is a top-down process. At each step, the process stops by a class, then descriptions of both the instance and the class are checked for attachment. If the conditions of attachment are satisfied, then the class is *sure* for attachment. If one of the conditions is violated, then the class is marked as *impossible*. Whenever a piece of information is missing, *i.e.* an attribute value, and even if every other attribute value respects conditions on the corresponding attribute domains, the class is marked as *possible* but not *sure*. Moreover, classification uses rules to propagate marks. For instance, whenever a class C is marked *impossible* for the instance, all the sub-classes of C are marked *impossible* as well.

Hence, both relations of an OBKRS are used during classification. Attachment is attempted to be inferred and specialization is used in order to propagate class marks according to extensional considerations.

There also exists a class classification process whose aim is to find the right place of a class description in the class tree or graph, according to the specialization relation. This is achieved through specialization rather than attachment inferences and tests [Cap94].

Classification in DLSs leads to the inference of subsumption relationships between a given concept and the existing concepts. Except for some systems such as LOOM [Mac91b], classification is a common process to both individual and generic concepts. However, OBKRSs distinguish between them, because attachment leads to membership and specialization leads to set inclusion.

Each inference mechanism we presented above involves matchings between attribute values and attribute domains. In other words, assuming that attribute domains are types, type checking and type inference exist in knowledge representation systems and furthermore, they are fundamental.

3 Where do types arise in knowledge representation systems ?

Despite some semantic choices or knowledge language expressivity that may differ from DLSs to OBKRSs, the solution we shall draw to type collection extensibility in the area of OBKRSs may basically be a solution to the same problem in the area of DLSs, because data types are used on the same way in both kinds of KRS. In both OBKRSs and DLSs, data types are explicitly used to specify an (elementary) attribute. For example, the *alcohol-degree* of a *drink* is a real value that can be specified to belong to a specific range. Data types arise in class descriptions too, although less obviously. The way types are expressed and manipulated in an OBKRS is identified below, in order to further outline a sound and complete solution to the extensibility to the type collection of a KRS.

3.1 Typed attributes

The description of an attribute is basically the expression of a set of values or instances. We generally name this set the *domain* of the attribute. The same attribute name can be associated with many domains within different class descriptions. The domain of a class attribute is specified in two main steps.

First, the domain is specified through the initial type descriptor, where the attribute name is associated with a data type or a class (see table 1). The initial type of an attribute is actually a reference to a set of elements which some known and accessible actions apply to, whether the initial type deals with host elements or knowledge entities. Consequently, the user who specifies an elementary attribute refers to an abstract data type whose specification and implementation is independent of the KRS features. For example, to choose whether the attribute *age* is an integer or a string, partly depends on the existence of a total order on integer and underlying operations, such as successor or addition, that fit well with the usual and basic way to measure time, whatever the implementation of integer is. In the case of a complex attribute, the initial type is the description of a set of individuals that are somehow similar and behave along a common way.

Second, more type descriptors may be applied on an attribute in more specialized classes, in order to restrict its initial set of values. For instance, in the class *beer*, the attribute *alcohol-degree* is a real value (initial type); in the class *trappist* (sub-class of *beer*), *alcohol-degree* is still a real, ranging in values from 6 to 15 (application of the descriptor `interval`). This restriction on the domain of *alcohol-degree* goes down the specialization hierarchy. Thus, the class *orval*, sub-class of *trappist*, specifies an unique value 6.2 (application of the descriptor `value`) for the attribute *alcohol-degree*. Whatever the domain of an attribute is, it is always expressed using the knowledge representation language.

In conclusion, two levels of type references thus appear in the whole attribute description:

1. selection of an abstract data type or selection of a knowledge class
2. expression of subset of values (or instances)

Whatever the initial type is, the attribute domain is involved in usual set operations required by both specialization and attachment tests. These set operations are mainly inclusion, membership, and intersection. Hence, whatever the initial type is, operations over attribute domains obey the same set-based semantics.

3.2 Set algebrae over data types and concepts

Given the definitions of necessary conditions for both attachment and specialization and the way attribute domains are used, we draw three algebrae which are shown to be closely related.

First, each data type $T = \langle V; O\rangle$, where V is a set of values, and O is a set of operations on these values, is associated with an algebraic structure over the power set of V. This algebraic structure specifies the way the KRS operates in a common way on all subsets of any data type. We write this algebra $\mathcal{A}_T = \langle 2^V; E\rangle_T$ where E is a set of set operations. It is important to notice that because of the predominancy of $\mathcal{A}_T$ in any OBKRS, whereas it is not managed in programming languages, the way types are managed in an OBKRS is different from the way they are in a programming language ; checking and inference processes do not rely on the same structures.

Second, each class C is made up of two parts. On one hand, considering class extensions (namely set of instances), a class $C = \langle\{I\}; O\rangle$ is the representation of a set $\{I\}$ of individuals of the application domain (*e.g.* Musicians) to which actions O of the application domain may apply (*e.g.* to win an award)[1]. A class is thus associated with an algebra over the power set of its extension. We write

[1] Actually, we do not consider the possibility of defining such actions within a class of a KRS, because it leads to considerations which are out of the scope of this paper. The set O of real-world actions has been introduced in order

this algebra $\mathcal{A}_C = \langle 2^{\{I\}}; E_C \rangle$ where E_C is a set of real-world actions that apply to sets of real-world individuals. Besides some elements of $2^{\{I\}}$ are represented by sub-classes of C. Among others, E_C contains actions that correspond to individual set inclusion, intersection or membership checking (*e.g.* Trumpeters are Musicians, Chet Baker is a Trumpeter). We shall focus on these set-based actions because they are semantically defined by the KRS to perform specialization and attachment. On the other hand, considering class intensions (namely descriptions), a class is a double $C = \langle D; O_d \rangle$, where D is a namely a set of record values thus represented by a record type, and O_d is a set of operations manipulating the record values. We then define the algebra over the power set of record values, written $\mathcal{A}_R = \langle 2^D; E_R \rangle$, where 2^D is the power set of record values, thus represented by the set of all record types, and E_R is a set of operations manipulating sets of record values, represented by operations manipulating record types. Among these record operations, sub-typing, membership and intersection are defined, because they closely match necessary conditions for specialization and attachment relations.

Together, the two latest algebrae $\mathcal{A}_C$ and $\mathcal{A}_R$ define the way classes and instances are organized in the knowledge base, *i.e.* according to the main relations defined by a set-based semantics. Each class or instance of the knowledge base is an element of each algebra. When an OBKRS does not consider term descriptions as definitions, the description of a term must at least cover its extension. This means that $\mathcal{A}_R$ and $\mathcal{A}_C$ are not equivalent. In addition, $\mathcal{A}_R$ is more likely an inclusive approximation of $\mathcal{A}_C$, since descriptions of classes are intended to capture their extensions.

It is interesting to notice that if one considers a record type as a particular data type, then $\mathcal{A}_R$ is a sub-algebra of $\mathcal{A}_T$. Besides, operations of $\mathcal{A}_R$ may call operations of $\mathcal{A}_T$ for elements of $\mathcal{A}_R$ are written from conjunction of labeled elements of $\mathcal{A}_T$. In other words, computations over class static descriptions (record types) may be performed independently of the KRS provided that $\mathcal{A}_T$ is not implemented in the KRS.

3.3 Implementations of algebrae

On one hand, data types are usually implemented in the host language, while the KRS usually implements $\mathcal{A}_T$ using the semantics of descriptors that actually build subsets. On the other hand, the KRS defines a language to represent both parts (extension and description) of classes, instances, and attributes by means of objects. Both $\mathcal{A}_R$ and $\mathcal{A}_C$ are thus implemented by the KRS, directly on top of the knowledge representation language.

Since the KRS cannot process on host language programs, the addition of a new data type implemented in the host language leads to incomplete KRS inferences. For example, assuming that `even-integer` is a data type implemented in the host language and is accessible in the KRS through a predicate (*i.e.* the LOOM way to integrate a new data type), let us then consider the four attribute domains below:

a_1 `in integer interval` $[0;+\infty]$
a_2 `in even-integer interval` $[4;+\infty]$
a_3 `in even-integer interval` $[0;+\infty]$ `except` 4 6 8
a_4 `in even-integer interval` $[0;2]$, $[10;+\infty]$

The KRS can infer neither $\|a_3\| = \|a_4\|$ nor $\|a_2\| \subset \|a_1\|$, unless `even-integer` has been defined as a sub-concept of `integer`. Yet in this last case, there is no way for the KRS to check the consistency of such a subsumption (or specialization) relationship, because the system cannot interpret host language predicates. Furthermore, the KRS cannot integrate the data type `date`, for example, together with its underlying total order and further use this order for the construction of subsets. Indeed, the property of total order is embedded in the definition of `integer` in the host language, or in the definition of

to point out the difference between the two cooperating aspects of a class; then, O must be compared to O_d which is introduced hereafter.

the concept `number` in the KRS. Therefore, no data type other than a number can be ordered. This is due to the fact that the KRS cannot access the specific properties of host data types, therefore it cannot take advantage of these properties for knowledge description purposes.

The KRS cannot guarantee the safety and the completeness of specialization over subsets constructed from new data types. Thus, specialization tests which involve classes containing these attribute descriptions are incomplete, so further is the result of a classification process.

3.4 Proposal

Avoiding the above incompleteness requires the KRS to be able to access properties and relevant features of any new data type. The way the new data type is related to existing data types must also be known by the KRS. We thus propose to design an extensible type language in addition to the knowledge representation language, that can express the relations among types and that can be somehow interpreted by the KRS. The type language is then intended to be the core of a two-level extensible type system. According to the two levels of types previously extracted through the two structures $T = \langle V; O \rangle$ and $\mathcal{A}_T$ (therefore $\mathcal{A}_R$), the type system must be connected to the KRS on one hand, and connected to the host language on the other hand. The two levels of the type system are closely related.

- The first level of types is connected to the host language. It is intended to capture, in a special format, the implementation of the host language defined data types, *i.e.* the implementation of each $T = \langle V; O \rangle$.
- The second level of the type system is intended to deal with the implementation of $\mathcal{A}_T$ (and further $\mathcal{A}_R$), for each data type defined at the first level. For this purpose, the second level of the type system includes an internal type language to express subsets from any type and set-based operations applying to these subsets.

The second level of types is meant to correspond to the typing of knowledge entities (classes and attributes), towards a normalized term language. It is the actual interface with the OBKRS, whereas the first level of types is independent of the OBKRS.

With this solution, the KRS will not have any access to the type language, but it does not need to. Indeed, all the set operations like inclusion, membership, etc., will be performed on terms of the second level of types (*i.e.* subsets) in a homogeneous way, whether the subsets are basic data subsets or are issued from the typing of the knowledge base.

4 The METEO type system

As pointed out in the previous section, a solution to type collection extensibility in an OBKRS is the design of a type system where both host data types and knowledge term types are to be handled. The type system METEO (Module of Extensible Types for Expressive Objects) resulting from our study is presented in this section. It is made up of two type levels:

- *C-types* are encapsulated and formated implementations of abstract data types (data structures + operations)
- *δ-types* are normalized terms. A δ-type represents a subset of C-type values. δ-types are intended to capture the result of knowledge term typing.

4.1 C-types as implementations of abstract data types

A C-type is an implementation of an abstract data type. The KRS user does not have access to implementation details, but he may refer to the signature. A C-type is defined, in a minimal fashion, by two operations. Let $\mathcal{U}$ be the set of universal values.

- $\in_T : \mathcal{U} \mapsto \{0;1\}$ is the membership predicate, from any value, to the type T
- $=_T : T \times T \mapsto \{0;1\}$ is the equality predicate between two values of T

A C-type is a data type, that is a double $\langle V; O \rangle$ where V is a set of values and O is a set of operations applying to values. O must include the two above operations, especially since the membership predicate actually characterizes V.

METEO contains, in its minimal setting, a whole set of C-types. The extensibility of this set is detailed in section 4.4.

4.1.1 Operations applying to values

In addition to the minimal operations above, many others may be implemented in the definition of a C-type, where parameters and/or results have to belong to one of the defined C-types. For example, the C-type `list` of METEO contains all the usual operations applying to lists, such as `map`, `union`, etc. We shall see later these operations may be used in a knowledge base in order to compute attribute values for instances through procedural attachment.

4.1.2 Hierarchical organization

C-types are ordered according to C-sub-typing. It is a set-based partial order. $T_1 = \langle V_1; O_1 \rangle$ may be a C-sub-type of $T_2 = \langle V_2; O_2 \rangle$, written $T_1 \preceq T_2$, if $V_1 \subseteq V_2$. A rich hierarchy of C-types is provided by METEO, as shown on figure 2.

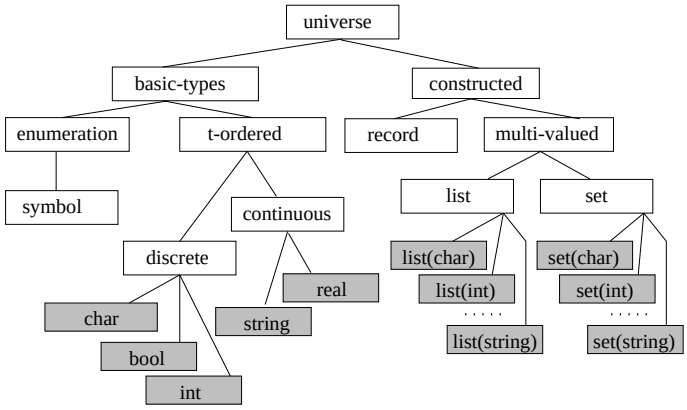

Figure 2: METEO C-type hierarchy. Only white C-types actually do exist in the minimal setting of METEO. Although grey tint C-types do not exist in the minimal setting, they can be easily defined by the user as addressed further on.

In the C-type hierarchy, middle C-types are mainly used to factor out generic properties about lower C-types. Hence, operations that are based on these properties are defined within middle types as second order functions and are inherited along the C-sub-typing order. For example, the C-type `t-ordered` factors out all kinds of totally ordered types. It implements some general algorithms, such as sorting. Then, whatever the total order relation may be, all its C-sub-types (`discrete`, `int`, etc.) inherit these general algorithms.

4.2 Internal language for subset expressions: EOLE

The second level of types in METEO deals with implementing normal forms to express subsets of C-type values. These types are named δ-types. They are intended to capture the results of knowledge

term typing, that is subsets of values. The way knowledge entities are typed in METEO is addressed in section 4.3. We focus here on the way METEO manages them.

4.2.1 C-types and δ-types

Each C-type $T = \langle V_T; O_T \rangle$ is associated with an algebraic structure $\mathcal{A}_T = \langle 2^{V_T}; E \rangle$ where E contains the usual set operations. METEO defines, for each C-type, the implementation of this structure. δ-types are elements of 2^{V_T}; they are terms of an internal type expression language named EOLE.

In other words, each C-type $T = \langle V; O \rangle$ is associated with a part of METEO that both defines the syntax of EOLE terms by taking into account the specific properties of T for a more optimal representation and implements set operations over this syntax. The set operations are mainly:

- $\in_{\delta,T}$: $2^V \times T \mapsto \{0; 1\}$, checks the membership of a value to a δ-type
- $\leq_{\delta,T}$: $2^V \times 2^V \mapsto \{0; 1\}$, checks whether the first δ-type is included in the second
- $\sqcap_{\delta,T} : 2^V \times 2^V \mapsto 2^V$, computes the greatest lower bound of any two δ-types (intersection)
- $\sqcup_{\delta,T} : 2^V \times 2^V \mapsto 2^V$, computes the least upper bound of any two δ-types (union)
- $\setminus_{\delta,T} : 2^V \times 2^V \mapsto 2^V$ computes the result of the first δ-type minus the second (difference)

These operations are implemented for each C-type, but they have the same set-based semantics. Moreover, each C-type must define the normalization operation used in order to ensure that two different δ-types from the same C-type, denote two different sets of values (section 4.2.3).

4.2.2 EOLE terms

EOLE terms are δ-types intended to express subsets of values. A δ-type is a double $\delta t = \langle T; e(T) \rangle$ where T is the C-type the δ-type is attached to and $e(T)$ is a syntactic expression representing the set denoted by δt. The expression $e(t)$ is a normal form, *i.e.* that two different δ-types from the same C-type will never denote the same set of values.

The syntax of EOLE is given for each C-type. It takes into account the given specific properties of the C-type that are useful in order to make the syntax optimal. For this purpose, C-types are grouped into classes of C-types, according to the useful properties they share. It is a second criterion for the C-sub-typing, the first criterion being set inclusion. For instance, `discrete` is a C-sub-type of `t-ordered` because both are totally ordered sets, but `discrete` is more specialized than `t-ordered` because it defines predecessor and successor operations, leading to a more optimal way to express subsets of values. They both belong to the same class of C-types which could be named `totally ordered types`. Actually, C-types could be compared somehow to the type classes as implemented in Haskell [HJe92, PJ93]. Indeed, a C-type captures the (extensible) way its δ-types — subsets of values — and the δ-types of its C-sub-types can be both represented and handled according to the properties and operations of the values.

EOLE terms express subsets through dynamic combinations of statically filled fields. Both the nature and the number of these fields depend on the properties of the C-types. For example, δ-types from the C-type `discrete` are made of a single field named `domain` that is filled with an ordered list of closed-bound intervals. Then, the set denoted by such a δ-type is merely the set denoted by the union of the intervals. This representation for δ-type is then inherited by all the C-sub-types of `discrete`. Hence, $\langle \texttt{int}; [6;18]+[45;+\infty_{\text{int}}] \rangle$ is a δ-type from the C-type `int`, whose denoted set is obvious. Note that each C-sub-type of `t-ordered` must define or inherit its own management of bound (maybe infinite) values.

The EOLE syntax concerning `constructed` C-types is far more complex. Indeed, several fields are used in order to guarantee that the δ-types are normal forms, while preserving the expressivity

of knowledge representation language descriptors. For example, the syntax of any $e(\texttt{record})$ is made up of three fields: $\langle \texttt{ref-typ}\ t_r\ ;\ \texttt{dom}\ d_1\ ;\ \texttt{dom-c}\ d_2\rangle$, where t_r is an actual record structure (set of labeled δ-types [CM91]), and d_1 and d_2 are explicit enumerators of record values. The way the three fields are combined for $\delta t = \langle \texttt{record}; e(\texttt{record})\rangle$ is defined by the following resulting set: $\|\delta t\| = \|t_r\| \cap \|d_1\| \backslash \|d_2\|$.

The complete syntax and denotational semantics of EOLE are formally described in [Cap95].

If $T_2 \preceq T_1$ in the C-type hierarchy, then fields of T_2 include those of T_1 and perhaps some additional ones as well. For instance, fields for `record` are inherited from `constructed`, although the field `ref-type` of `constructed` does not refer to a record type structure, but to any data structure. δ-types from `list` are expressed through the three fields of `constructed`, plus the field `card` that provides information about the cardinality of lists belonging to each δ-type. The δ-type $\langle$ `list`; $\star\langle$ `int`; [2;8] $\rangle$; $\top$; {(4, 5, 6) (4, 5, 7) }; $[2;+\infty]\rangle$ represents the set of all lists of at least two integer values which range between 2 and 8, excluding lists (4, 5, 6) and (4, 5, 7). Note that, for each C-type $T = \langle V; O\rangle$, the symbol $\top$ represents the powerset 2^V.

4.2.3 Normalization of EOLE terms

Normalization is an operation that computes the normal form δt^N associated with any newly created δ-type δt. A normal form is itself a δ-type. METEO deals only with normal forms. The normalization operation is used to guarantee that two different δ-types from the same C-type denote two different subsets of values as formally expressed below.

$$\forall \delta t_1^N = \langle T; e_1(T)\rangle, \delta t_2^N = \langle T; e_2(T)\rangle : e_1(T) \neq e_2(T) \Longleftrightarrow \|\delta t_1^N\| \neq \|\delta t_2^N\|$$

Since EOLE is distributed among C-types, the normalization operation is defined for each class of C-types, according to each syntax. Normalization is achieved through a syntactic rewritting system which is fully presented, as well as proven sound, complete, and confluent in [Cap95].

The normalization of METEO and the normalization in DLSS [Neb90, BPS94] both have the same purpose. However, normalization of DLSS is performed on knowledge entities while normalization of METEO is utterly independent of the KRS.

4.2.4 δ-sub-typing and δ-type lattices

δ-types from the same C-type $T = \langle V; O\rangle$ are partially ordered according to δ-sub-typing. δ-sub-typing means subset inclusion. It is then directly issued from the operation $\leq_{\delta,T}$ of $\mathcal{A}_T$ (section 4.2.1). Like normalization, the δ-sub-typing is implemented according to the distributed EOLE language, *i.e.* for each class of C-types. δ-sub-typing is then locally performed by combinations of the fields describing δ-types. For example, δ-sub-typing between two `discrete` δ-types leads to the computation of inclusion between two ordered sets of closed-bound intervals. δ-sub-typing between δ-types from `constructed` is obviously less simple since δ-types are expressed through three fields, and four in the case of `list` or `set`. Let $\delta t_i = \langle \texttt{constructed}; [t_i; D_i; C_i]\rangle$, $i = 1..2$, be two δ-types from `constructed`, where $\forall i = 1..2, t_i = \langle T; e_i(T)\rangle$. Then, δ-sub-typing among `constructed` δ-types is defined as follows:

$$\delta t_1 \leq_{\delta,\texttt{constructed}} \delta t_2 \Leftrightarrow \begin{cases} t_1 \leq_{\delta,T} t_2 \text{ and} \\ \text{if } v \in D_1 \text{ then } v \in D_2 \text{ and} \\ \text{if } v \in C_2 \text{ then } v \in C_1 \text{ or } v \notin D_1 \text{ or } v \notin_{\delta,T} t_1 \end{cases}$$

δ-sub-typing is fully presented, as well as proven sound and complete according to set-based semantics in [Cap95]. In other words, the following result has been demonstrated.

Theorem 1 *For each C-type T, $\forall \delta t_1$, $\delta t_2 \in \Delta(T)$, $t_1 \leq_{\delta,T} t_2 \Longleftrightarrow \|t_1\| \subseteq \|t_2\|$*

Let us recall that $\forall t, \|t\|$ is the set denoted by t, and $\Delta(T)$ is the set of all possible δ-types of the C-type T. if $T = \langle V; O\rangle$, $\Delta(T)$ is merely the implementation in METEO of the powerset 2^V.

As METEO implements δ-sub-typing between δ-types of each class of C-types by combinations of δ-type fields, operations $\sqcap_{\delta,T}$ and $\sqcup_{\delta,T}$ are defined. This allows METEO to manage homogeneously a δ-type lattice under each C-type, using a generic and dynamic process.

4.2.5 γ-sub-typing

In order to be able to check set inclusion between two δ-types from different C-types (*e.g.* `int` and `even`) which are related by C-sub-typing, δ-sub-typing is extended as γ-sub-typing. For this purpose, property 1 has been established. It shows that there exists a non-strict set-preserving mapping between δ-types from different but related C-types. The proof of this property is then presented for the reader to understand the way the mapping can be computed.

Property 1 *Let T_1, T_2 be two C-types such that $T_2 \preceq T_1$. There exists a homomorphism $h : \Delta(T_1) \rightarrow \Delta(T_2)$ such that*

$$\forall t_1, t'_1 \in \Delta(T_1) \text{ such that } t_1 \leq_{\delta,T_1} t'_1,$$
$$h(t_1) \in \Delta(T_2), h(t'_1) \in \Delta(T_2) \text{ and } h(t_1) \leq_{\delta,T_2} h(t'_1)$$

Let us define h such that $\|h(t_1)\| = \|t_1\| \backslash \{e \in \|t_1\| \mid e \notin \|T_2\|\}$. Therefore, the homomorphism h may be seen as a projection. Since $h(t_1)$ must be an expression that denotes only elements of T_2, $h(t_1)$ is expressed through the syntax defined for T_2. For the same reasons, $\|h(t'_1)\| = \|t'_1\| \backslash \{e \in \|t'_1\| \mid e \notin \|T_2\|\}$. Since $t_1 \leq_{\delta,T_1} t'_1$, then $\|t_1\| \subseteq \|t'_1\|$, thus $\{e \in \|t_1\| \mid e \notin \|T_2\|\} \subseteq \{e \in \|t'_1\| \mid e \notin \|T_2\|\}$. As a consequence, $h(t_1) \subseteq h(t'_1)$; according to the theorem 1, we thus prove that $h(t_1) \leq_{\delta,T_2} h(t'_1)$.

The homomorphism h represents, through the syntax defined for δ-types of T_2, the set of values denoted by any δt of T_1 without values that do not belong to T_2. For example, if ($T_2 =$`even`$\preceq$($T_1 =$`int`), if $\delta t = \langle$ `int`;[11;32]+[51;90]$\rangle$ then $h(\delta t) = \langle$`even`;[12;32]+[52;90]$\rangle$. Yet, as defined above, h does not always preserve the strictness of a δ-sub-typing relationship. Indeed, in the worst case, a strict δ-sub-typing between t_1 and t'_1 is mapped towards an equality, since elements may be removed during the mapping. For example, if $t_1 = \langle$`int`;[2;8]$\rangle$ and $t'_1 = \langle$`int`;[2;9]$\rangle$ then obviously, $t_1 \leq_{\delta,T_1} t'_1$ whereas $h(t_1) = h(t'_1) = \langle$`even`;[2;8]$\rangle$.

Property 1 is interesting because it shows that any two δ-types from related C-types may be compared according to set inclusion. Therefore, it is a first step towards the definition of a sub-typing that mixes both δ-sub-typing and C-sub-typing. This broader sub-typing is named γ-sub-typing and is defined in definition 1.

Definition 1 (γ-sub-typing) *γ-sub-typing is a relation that links two δ-types from different C-types. It is computed by the combination of both C-sub-typing and δ-sub-typing. Let $t_1 = \langle T_1; e_1(T_1)\rangle$ and $t_2 = \langle T_2; e_2(T_2)\rangle$ be two δ-types, and $T_2 \preceq T_1$. γ-sub-typing between t_2 and t_1 is written as $t_2 \preceq_\gamma t_1$ and is defined as follows:*

$$t_2 \preceq_\gamma t_1 \Leftrightarrow \begin{cases} T_1 = T_2 = T \text{ and } t_2 \leq_{\delta,T} t_1 \\ \text{or } T_2 \prec T_1 \text{ and } t_2 \leq_{\delta,T_2} h(t_1) \end{cases}$$

where h is the homomorphism previously defined.

Due to the γ-sub-typing, any two δ-types can now be compared according to subset inclusion, provided that the required homomorphisms are defined. Yet, whereas δ-sub-typing and C-sub-typing

are explicitly handled and maintained by METEO, γ-sub-typing relationships are only computed upon request, provided that required homomorphisms are defined.

γ-sub-typing has been defined in METEO in order to avoid any incompleteness in subsumption checking which would result from the non-assertion of actual inclusion links between involved data types. As a consequence of γ-sub-typing, the incompleteness of subsumption related in section 3.3, between attributes a_1 and a_2, does not hold anymore, since data types may be safely related in METEO by inclusion through C-types and further their δ-types are related by γ-sub-typing.

4.3 Connecting METEO to the OBKRS through δ-types

The METEO design is independent of the KRS, yet δ-types are the results of the typing of knowledge entities. In other words, METEO can be instantiated by the typing of a knowledge base. The typing of a knowledge base is thus the core of the interfacing between the KRS and METEO. This section outlines the features of this interface.

4.3.1 Typing of knowledge entities

Let us recall that each knowledge term is made of two components, its intension and its extension. δ-types of METEO have been designed in order to capture normal expressions of knowledge term intensions. The typing of a knowledge term is made up of three steps. Only the first step takes part in the interface between the KRS and METEO, while the two others are internal processes of METEO.

1. Translation from the knowledge representation language to EOLE. It is relevant to point out here that EOLE has been designed in order to be able to be at least as expressive as the knowledge representation language of the system we considered in our study (namely TROPES [Pro95]). It means that any subset of value expressed through the TROPES language can also be expressed in EOLE. As a consequence, EOLE may not be suited to more expressive knowledge representation languages for this could lead to unsafe further computations.

   - Whenever the knowledge term to be typed is an attribute, the translation needs to interpret the knowledge representation language descriptors towards fields of a δ-type. The C-type is chosen according to the initial type of the attribute (table 1). For example, if the attribute identified by *age* is firstly defined as an integer in a class, then it is typed by a δ-type of the C-type `int`. Any farther new description of *age* in a sub-class then will be typed towards a δ-sub-type or a γ-sub-type.
   - If the knowledge term to be typed is a class, it is typed towards a δ-type from `record`. The labels are the attribute identifiers and their associated types are δ-types corresponding to attribute typing results. Indeed, as pointed out in section 3.2, class descriptions may be viewed as record types, and instances descriptions are record values.

2. Normalization of the resulting δ-type.

3. Insertion in the corresponding δ-type lattice. This step is achieved only if the knowledge term does not lead to an inconsistency with regard to its relationships with other knowledge term types, and if it is validated in the knowledge base.

Figure 3 shows the dynamic links that exist between the knowledge base and the δ-types. These links allow δ-type lattices to be built and to evolve according to the evolution of the knowledge base, without any redundance or lack of information.

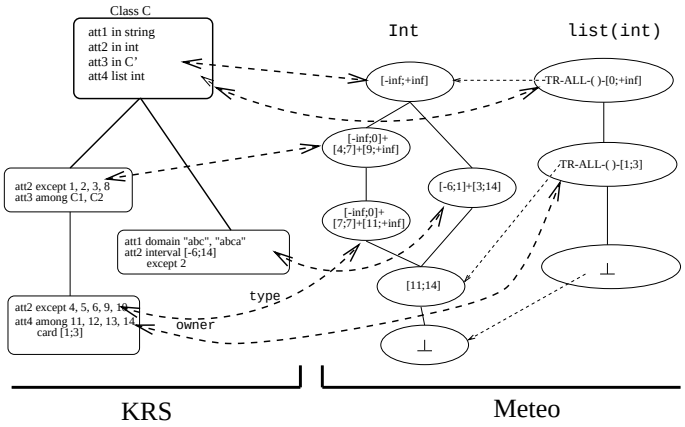

Figure 3: A specialization tree on the left; two δ-type lattices on the right. This picture shows both the links that exist between knowledge entities and METEO terms (dashed thick lines), and the internal links between METEO terms that stand for subordination of δ-types to others (dashed thin lines). For example, in the case of δ-types of `list(int)`, the first field of the δ-type syntax is an explicit reference towards an existing δ-type. Such a reference (which is called subordination) is written `TR` on the picture.

4.3.2 Isomorphism between δ-types and knowledge term descriptions

Due to the typing of the whole knowledge base, each knowledge term is associated with a δ-type. The set-based operations applying to δ-types are equivalent to the set-based operations of the KRS which are used to handle main relationships between knowledge entities. Consequently, each operation which manipulated knowledge term descriptions is now associated with a semantically equivalent operation in METEO that manipulates δ-types. For example, δ-sub-typing between record δ-types is equivalent to specialization between class descriptions. In other words, the OBKRS now delegates to METEO all its intensional operations, that is operations applying to knowledge term descriptions.

Since δ-types are normal forms, operations applying to δ-types rather than to knowledge term descriptions are more efficient. Indeed, the normalization is performed only once and the syntax of EOLE is more optimal than the syntax of the knowledge representation language which must be expressive, therefore permissive.

Since METEO carries out the intensional operations of the KRS, it also interferes with all the processes of the KRS which deal with entity descriptions. Error handling, explanation, knowledge base merging, knowledge base comparisons, management of the knowledge base dynamic evolution, categorization, and constraint management, are all examples of processes which somehow have to be connected to METEO and which take advantage of its properties.

Figure 4 shows where does METEO take place, as a bridge between the host language (where data structures are actually implemented) and the knowledge representation system (whose entities are typed towards δ-types which are manipulated by relations equivalent to specialization and attachment).

4.4 Extensibility of METEO

One main interest of METEO is its extensibility capacity that is independent of the KRS it is intended to be dedicated to. This section addresses the way METEO fully deals with type collection extensibility, as a new tool for the development of knowledge representation applications.

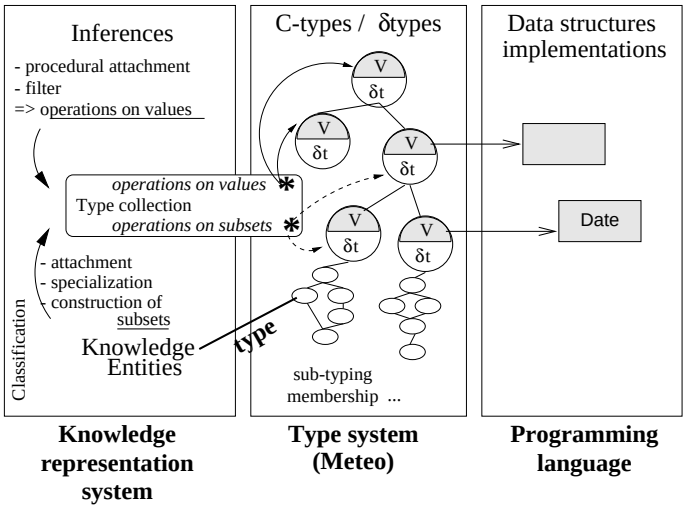

Figure 4: Connection of METEO to the knowledge representation system and to the host language. METEO gets the implementation of C-types from the host language (part V of each data type), and the implementation of δ-types is defined (or inherited) by each C-type (part δt). δ-types capture the types of knowledge terms. Operations on values of C-types may be used by the knowledge representation system through local inference mechanisms. Main relations among objects (specialization and attachment) are delegated to the type system which implements them as γ-sub-typing and membership on C-type values and δ-types.

4.4.1 Adding a new data type

Due to the C-type hierarchy, METEO is both a polymorphic and a extensible type system. Adding a C-type may be achieved dynamically by the user. First, the user must select a C-super-type. Second, he must declare links towards required operations that are implemented in the host language. Let us recall that these operations are mainly the membership and equality predicates, plus possibly some specific operations that represent properties required by the C-super-type. Once the previous two steps have been achieved, the new C-type is fully and immediatly integrated into METEO, as any other predefined C-type. The function used to link a host language data type to METEO is named `create-adt`. For example, adding the C-type `date`, as a `discrete` C-type, is performed by executing the following code:

```
(create-adt 'date  'discrete
     equal:  'dateq
     member:  'datep
     order:  'ldate
     succ:  '+day
     pred:  '-day )
```

where the quoted symbols are names of functions implemented in the host language. The user must define the `order` function on `date` values because it is required by the given C-super-type `t-ordered`. Similarly, functions `pred` and `succ` are required by the C-super-type `discrete`. Indeed, these three functions are mandatory for they allow δ-types to be fully, and in an optimal way, represented by means of lists of closed-bound intervals. In addition, the user may define any other function applying to `date` values that would be used in the knowledge base.

The formated implementation of the C-type `date` is then generated by METEO by setting up dynamic links between METEO predefined modules and the actual implementation of the data type.

This implementation may be either written by the user, or retrieved from a programming library. Basically, METEO encapsulates this implementation. The new C-type then inherits everything from its C-super-types. For example, the sorting operations defined in `t-ordered` do not have to be defined by the user for the new C-type `date`, for these operations are mostly generic. It is an illustration of the inclusion polymorphism of METEO.

Once the new data type is encapsulated in METEO, the resulting new C-type is considered as any other C-type of the hierarchy. In the case of the addition of a `basic` C-type, METEO generated automatically the C-types resulting from the application of each unary constructor to that new C-type. In the example above, METEO would generate two other C-types, namely `list(date)` and `set(date)`. Similarly, the addition of a new unary constructor (*e.g.* `sequence`) leads to the creation of all the C-types that result from the application of the constructor to the basic C-types.

Any data type introduced in METEO as a C-type using `create-adt` is associated to the KRS through its name and the name of its C-super-type. The descriptors authorized for the new data type are these which are authorized for its C-super-type.

4.4.2 Extensibility and δ-types management

This section addresses the actual capacity of METEO as a type system for an object-based knowledge representation system. Indeed, the two connected levels of METEO allow to strenghten its extensibility capacities. Let us recall that the C-type hierarchy has been designed from two main criteria:

- the set of values of a C-type is included in the set of values of any of its C-super-types,
- a C-type gets all the properties of any of its C-super-types; it may add some other properties which may represent the core of the way EOLE is designed.

Due to the second criterion, the extensibility of METEO is protected along δ-type management. Indeed, whenever a user adds a new data type, he does not have to define the EOLE sub-language associated to the new type; the syntax of δ-types is inherited from the C-super-type, as well as all the set-based operations applying to them. This possibility is due to the generic way these operations are defined from basic predicates over C-type values. In other words, for any new implemented data type T, the powerset algebra $\mathcal{A}_T$ is automatically generated. Therefore, the user does not have to deal with δ-type management and further attribute domain management, such as attachment or specialization tests. Nevertheless, the user must provide METEO with an appropriate link to the codes of operations, as previously indicated. If he does not, they will be inherited from the C-super-type, without any compatibility checking performed by METEO.

In the previous example of the new C-type `date`, assuming that a date value is a `int` triplet `(dd,mm,yyyy)`, the user can now express any subset of date values by using knowledge representation language in order to specify domain of so-typed attributes. For example, the knowledge representation language expression:

exam-period `a date interval` [(13,05,1996);(20,05,1996)[`except` (18,05,1996), (19,05,1996)

is typed in METEO by the EOLE term (δ-type):

$$\langle \texttt{date} \ ; \ [\ (13,05,1996);(17,05,1996)\ ] \rangle$$

Any attribute typed by `date` is associated with a δ-type from the C-type `date`. Therefore, the set-based management of the attribute domain is performed by METEO through its associated δ-type. In particular, specialization checking or inference involving attributes typed by `date` are fully handled by METEO, according to a semantics common to every data type. As a consequence, classes and instances that contain descriptions of this attribute can then be involved in specialization and attachment checking processes.

As previously said, no KRS is able to deal with the example page 12 (section 3.3) in a sound manner. With the connection of METEO to the KRS, not only the new data type `even-integer` can be added, but this addition goes with the automatic generation of all set-based operations that manage subsets of even values. The normalization process is one of these operations. Hence, METEO computes the type of all four attributes:

$$\text{type } (a_1) = \langle \texttt{ integer } ; [0;+\infty_{\texttt{int}}] \rangle$$
$$\text{type } (a_2) = \langle \texttt{ even-integer } ; [4;+\infty_{\texttt{even-int}}] \rangle$$
$$\text{type } (a_3) = \text{type } (a_4) = \langle \texttt{ even-integer } ; [0;2] + [10;+\infty_{\texttt{even-int}}] \rangle$$

Then METEO is syntactically able to infer that $\|a_3\| = \|a_4\|$, thanks to the normalization process which ensures a synctactic equality over δ-types whenever (and only when) there is a semantic equality. Moreover, γ-sub-typing is used to check whether or not $\|a_2\| \subseteq \|a_1\|$, where h is the homormorphism that translates δ-types of `integer` towards δ-types of `even-integer`:

$$h(\text{ type}(a_1) \text{ }) = \langle \texttt{ even-integer } ; [0;+\infty_{\texttt{even-int}}] \rangle$$

Then, using δ-sub-typing among δ-types of `even-integer` (which is automatically generated by METEO using the basic set-based operations of the new data type `even-integer`), it is straightforward for METEO to conclude that

$$h(\text{ type}(a_2) \text{ }) \leq_{\delta,\texttt{even-int}} \text{type } (a_1)$$

i.e. according to the definition of γ-sub-typing, type$(a_2) \leq_\gamma$ type (a_1) meaning $\|a_2\| \subseteq \|a_1\|$. As far as we know, no other KRS is able to check such a sub-typing relationship — although it is necessary to get a complete subsumption checking process — because both subsets are initially constructed from values and operators of different data types. Yet this is possible in METEO because of the hierarchical organization of C-types which permits polymorphism not only of operations among values of C-types, but also of operations among subsets of values.

4.5 About the implementation of METEO for the KRS TROPES

METEO has been implemented for TROPES [Pro95] with the programming language ILOG-TALK [ILO94]. ILOG-TALK is a LISP-like language, provided with an object management module, ILOG-TALK is therefore close to Common-lisp.

4.5.1 Implementation features

C-types of METEO are implemented as classes whose attributes are

- the fields describing δ-types
- the δ-sub-typing relationships (δ-type lattices are implemented as adjacency lists)
- the links between TROPES entities and δ-types

δ-types are obviously instances of these classes. Values are usual, possibly constructed, LISP values. Values may also be instances from a TALK class, if this class is used as a C-type membership predicate. Methods of C-type classes are, on one hand operations of the algebra $\mathcal{A}_T$ plus the normalization process, and, on the other hand, they are any user-defined functions associated with C-type values, including membership and equality predicates. Inheritance between classes stands for C-sub-typing.

4.5.2 METEO and other features of TROPES

TROPES is a multi-point of view OBKRS: a family, called a concept in TROPES, is split into several points of view. A point of view focuses on one aspect of the individuals which the concept denotes according to a given set of relevant properties. A point of view consists of a tree of classes ordered by single specialization. An instance belongs to exactly one class in each point of view. Classes from different points of view may be related by bridges [MRU90]. A bridge is a relationship between several different classes not linked by specialization: a bridge from classes $C_1, \cdots, C_n$ to class C means that whenever an instance belongs to every C_i ($i \in [1;n]$), it necessarily belongs to C as well. Like specialization or attachment, a bridge must obey set-based restrictions on term descriptions. Thus, as METEO sub-typing is intended to represent specialization between term descriptions, METEO defines another operation that is equivalent to the bridge checking [Cap95].

Since METEO carries out the intensional operations of TROPES, it has a strong impact on the whole internal organization and implementation of TROPES. METEO then cooperates with all inference and checking processes of TROPES, as well as with error handling or knowledge dynamics maintenance [CE96].

METEO is used during instance and class classifications [CEG95] through the checking and the inferences of membership and inclusion relations. It is also useful to process symbolic categorization, because similarity measures are defined within the definition of C-types, according to their specific properties [VE95].

At last, METEO weakly cooperates with MICRO, the constraint management system of TROPES [Gen95]. Constraints may apply to attribute domains and class extensions. In addition, constraints may also exist between attribute domains. Many predefined constraints apply to domains that are built from any existing initial data type, such as `list` or `int`. Although the existence of dynamic constraints in TROPES prevents the METEO static type checking from being sound and complete, METEO and MICRO cooperate in order to reduce dynamic constraint checking. Indeed, MICRO performs domain reductions by means of specific operations associated with the different kinds of predefined constraints. The typing process is thus completed by the integration of the results of attribute domain reductions computed by MICRO. For example, let `mic-eq(mic-add(`a`,`b`),`c`)` be a class constraint defined among the three attributes a, b, and c, meaning $a + b = c$, where the domains of these attributes are integers ranging respectively in intervals [0;6], [8;19] and [-2;29]. Then MICRO reduces the domain of c from [-2;29] to [0+8;6+19], *i.e.* [8;25]. This reduction is performed using MICRO predefined rules which call METEO operations on C-type values and δ-types. The final δ-type associated with the attribute c is the result of the intersection $\sqcap_{\delta,\text{int}}$ between its initial δ-type $\langle$`int`;[-2;29]$\rangle$ and the temporary δ-type $\langle$`int`;[8;25]$\rangle$.

5 Example

This section outlines an example of the usefulness of METEO extensibility during the design of complex applications. Here is addressed the field of molecular biology where biologists aim at representing the phenomenon of protein synthesis. The modelling of this phenomenon is described step by step, through the attributes of the class *protein* and *protein-gene* (figure 5).

The characterization of any gene is its associated, unique DNA sequence, usually represented as a sequence over the alphabet 'A' 'C' 'G' 'T'. The characterization of any protein is an unique sequence of amino-acids, represented as a sequence over an alphabet of twenty characters. The protein synthesis from a gene requires the production of messenger RNA, represented as a sequence over the alphabet 'A' 'C' 'G' 'U'. The protein synthesis is formally represented by the composition of several functions over sequences.

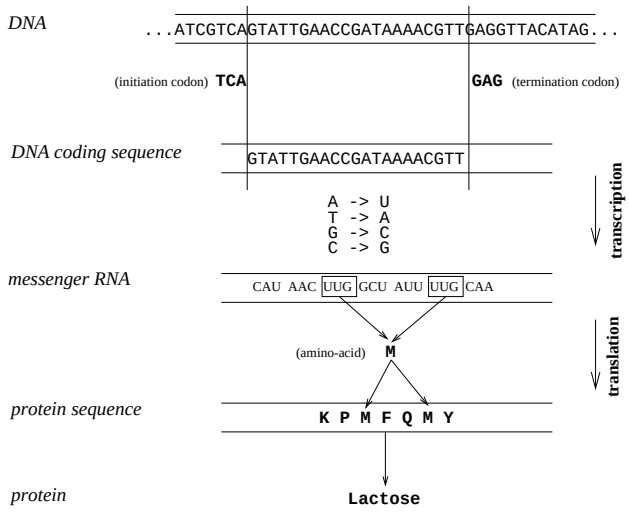

Figure 5: Simplified model of protein synthesis from DNA.

But the KRS does not provide the user with the implementation of the data constructor `sequence`, while it is rather useful to the description of protein gene and protein. There exist three ways to define such a data constructor.

1. The user may use the knowledge representation language. However, the data constructor `sequence` may not represent a direct and unique element in the application domain, namely molecular biology. The same sequence of characters could stand for both a RNA and a protein, which could be ambiguous, thus resulting in further erroneous computations of the KRS about these sequences and about what they mean to represent.

2. The user may create the new data constructor (C-type) `sequence`, as a C-sub-type of `list`, such that it inherits the syntax and handling of δ-types from `list`.

3. The user may use the host language to fully implement the data type `sequence`. Yet, this solution does not allow further attribute domains to be involved in KRS inferences and checking processes, such as specialization or attachment.

The second solution is obviously the most appropriate here, especially since operations over sequences can then be implemented and because δ-types from `sequence` will type attribute domains which then will be involved in set-based processes. When creating the C-type `sequence`, METEO automatically creates `sequence(char)`. The user can either use it or create C-sub-types of it for a better fit with alphabets which sequences of its application are constructed over.

Classes *protein-gene* and *proteins* are partly described below, using user-defined C-type and user-defined functions defined within new C-type definitions, which are used to infer unknown attribute values for particular instances of the classes (section 2.4).

```
Class protein-gene a-kind-of gene
    Attributes:
        promoter in sequence(char) among 'A' 'C' 'G' 'T' card [2;6]
        gene-type in string domain ("CDS" "ORF" "URF") default "CDS"
        DNAseq in sequence(char) among 'A' 'C' 'G' 'T'
        coding-seq in sequence(char) among 'A' 'C' 'G' 'T'
                compute-with    mt-extract-maxseq (
                                *.DNAseq,
```

```
                                ('T' 'A' 'C'),
                                ('T' 'T' 'G') )
prot-seq in sequence(char) among ['A';'Z'] forbidden 'B' 'J' 'O' 'U' 'X' 'Z'
          compute-with     mt-transcript (
                           mt-traduct (*.coding-seq ) )
protein in string
       compute-with-filter { P proteins, P.name:
                              P.AAseq = *.prot-seq }
function in string domain ("catalytic" "storage" ..."inhibitive")
rbs in RBS
```

where `compute-with-filter` corresponds to the filtering mechanism and `compute-with` is namely procedural attachment (section 2.4). Both are mainly ways to compute (or retrieve) the values of the attributes they are associated with. In the above example, *RBS*[2] is a class of the knowledge base. Functions prefixed by `mt-` are defined in METEO within the C-type `sequence(char)` and using the host language. They apply to particular sequences and correspond to basic manipulations of sequences such as extraction of sub-sequences, transformation from an alphabet to another, etc. These functions, when applied together in a particular way, represent the phenomenon of protein synthesis; the coding sequence of DNA is extracted from the whole DNA sequence of the gene, which is part of the process represented by attributes *DNAseq* and *coding-seq*. Then RNA is obtained through the application of transcription to the coding DNA sequence (represented by attribute *prot-seq*), and finally the protein associated with the gene can be retrieved using a filter which looks at all the proteins of the class *proteins* and keeps the names of those whose proteinic sequence corresponds to *prot-seq* of the gene. The class *proteins* is partly described below:

```
Class proteins a-kind-of biological-object
     Attributes:
         AAseq in sequence(char) among ['A';'Y'] forbidden 'B' 'J' 'O' 'U' 'X'
         name in string
```

Since METEO manages δ-types as attribute types, set-based relationships between attributes typed by `sequence` can now be soundly checked. For instance, the domains of both attributes *AAseq* of class *proteins* and *prot-seq* of class *protein-gene* are not expressed in the same way using the knowledge representation language. The KRS cannot find out by itself that these domains actually denote the same set of values. Among other consequences, this does not allow the KRS to check the soundness of the filter that computes the value of the attribute *protein* in the class *protein-gene*. However, these domains are typed in METEO by the same δ-type, using typing and normalization processes of the C-super-type `list` which are parameterized by `sequence` C-type mandatory specific functions:

$$\langle \texttt{sequence(char)}; \langle \texttt{char}; ['A';'A']+['C';'I']+...+['Y';'Y'] \rangle; \top; [0;+\infty] \rangle$$

Hence, the two domains are recognized to be equivalent. This would not have been possible without METEO which provides the new C-type `sequence` with all required set-based operations. Since even the attribute typed by a newly defined data type may be involved in set-based checking and inference processes through a sound and complete way, the classes that contain them can in their turn be involved in these processes. This is one of the main consequence of delegating KRS intensional operations to METEO.

6 Related works

The type system METEO contributes in two different area. With regards to other type systems, the contribution of METEO is its ability in dealing with arbitrary subsets of values (δ-types) in an extensible way. Then METEO must be compared to works carried out in the area of knowledge representation.

[2] *Ribosome Binding Site*

6.1 METEO vs. other type systems

The design of a type system dedicated to a language is not original at all. Besides a type system is one of the most basic components of programming languages or database systems. Yet no type system has been designed so far to deal with the specificities of a KRS that stem from frames. Those specificities come from the expressiveness of the knowledge representation language which complicates all type checking processes. Indeed, subsets of values of any data type can be constructed by the user — or by an inference mechanism — then are involved in type checking processes related to subsumption (section 2.3). One of the most expressive type systems dedicated to an object-oriented programming language is the type system of ADA [Uni83], which both permits and processes type definitions such as `subtype` *young-age* `is integer range 1..18`. Although, such a type definition is not usual in programming languages. It means exactly the same as the following expression of an attribute domain in an OBKRS: *young-age* `a integer interval` [1;18] (according to the syntax we used so far). Yet there are three main differences between the ways ADA and an OBKRS deal with types.

1. The type system of ADA does not permit to compare two types that are not related by the `subtype` relation. For example in ADA, values of the type *young-age* defined above are not comparable to values of the type defined as `subtype` *old-age* `is integer range 1..150`, although the set represented by the type *young-age* is included in the set represented by *age*. Such a comparison, which could be viewed as the inference of a sub-typing relationship, may be performed at any time in an OBKRS in order to check specialization relationships. Besides, all KRSs permit it; METEO permits it through δ-sub-typing and/or γ-sub-typing. ADA cannot permit such sub-typing inferences because it does not distinguish between a data type (values + operations) and a set of values by itself. METEO does distinguish between both notions through δ-types and C-types.

2. Not all predefined data types of ADA can be restricted as `integer` can be. Besides only the descriptor `range` can be applied to type in order to define a sub-type (strictly, a subset). For example, if one wants to create the type of roman numeral, he or she cannot specify it as a sub-type of `character` through the enumeration of the relevant characters 'I', 'V', 'X', etc. As a consequence, he or she cannot use input/output operations that apply on characters. The example of section 5 shows that it is necessary to have that possibility in knowledge representation because of the complexity of the structures that capture the knowledge. To be more general, we can say that despite its important expressivity, ADA is not expressive enough for its type system to be adapted to the knowledge representation requirements.

3. Since there is only one descriptor in ADA that allows one to restrict the set of values of a data type, the type system of ADA does not contain any way to add a normalization step during (or before) type checking. Because an OBKRS requires a deeper expressivity, the underlying type system must fully integrate normalization in order to ensure both the soundness and the completeness of both type checking and type inferences.

In order to compare the requirements of an object-oriented programming language and these of an OBKRS regarding type management, we chose ADA because it comes with one of the most expressive type system. Other type systems such as these of Simula or C++ perform type checking but they do not allow one to build subsets of values as domain of the attributes without having to program a class that would be the implementation of the sub-type. Consequently, those type systems do not handle the dimension corresponding to subsets of values (δ-types) of usual data types (C-type), whereas it is one of the specificities of METEO. This lack of subset management explains why we did not use an existing type system to be adapted and connected to the KRS.

Type systems are the basic of almost any system in computer science. Most of the theoretical proposals and system implementations of object-oriented data models are restricted to the possibility

of defining only enumerated sets [BKKK87, LR89]. However, the object-oriented data model TM [BdBZ93] is the first that is formally capable of dealing with arbitrary set expressions and powertypes in the context of sub-typing and multiple inheritance. For such a purpose, TM permits arbitrary set expressions (enumerated sets and predicative sets defined as $\{x : \sigma | \phi(x)\}$) as well-typed expressions. Therefore, METEO is comparable to TM. Indeed, METEO deals with predicative sets too — with enumerated sets as well —, through both the typing of filters and the storage of domain reductions computed by the constraint management module MICRO (section 4.5.2). The specifities of METEO which distinguishes between C-types and δ-types could actually be adapted to an object-oriented database system whose designers would like to integrate more expressive ways to specify the features of object classes.

6.2 METEO vs. KRS similar abilities to manage types

Type collection extensibility is still a problem in most KRSs in the area of description logic systems, for example. Despite some incomplete solutions, such as TEST-H in CLASSIC or `:predicate` in LOOM, only two systems provide the user with type collection facilities in such a way that type checking remains sound. The TEST-H predicate of CLASSIC is currently studied in order to deal with concept subsumption [BIM96].

The system K-REP [MDW91, OMW96] provides the user with a partial type collection extensibility, although no type system is associated with K-REP. In order to add a new data type, the user must define around fifteen functions within the KRS. These functions basically correspond to the handling of subsets of values of the new type that can be constructed with the knowledge representation language. Once these functions are written by the user, the new data type is connected to K-REP. Relationships such as subsumption can then be soundly checked or inferred. The manner in which K-REP integrates new data types is the result of the modular organization of the whole implementation of the KRS, that leads to the dynamic creation of links between host language programs and the KRS-defined checking processes. With this solution, the user must still program set-based operations, whereas both the hierarchical organization of METEO's C-types and the implementation of powerset algebrae as δ-types permit the inheritance of those operations. With METEO, only the basic operations have to be defined by the user. Besides the user can directly reuse existing programs from programming libraries, without any further adaptation.

The design of METEO has been achieved according to formal observations close to those B. Gaines reported in [Gai93] regarding data type extensibility and KRSs. Starting from the knowledge representation server KRS [Gai91], B. Gaines pointed out that descriptions of concepts in DLS may be considered as a specific kind of record types. Some set-based constraints which apply to this type, may be resolved using its basic operations. Individuals are then considered as variables ranging in record types. This approach is similar to the one of METEO which manages instances as values of statically constrained record types. Besides, B. Gaines pins down concept description management onto the same abstraction level of the management of any subset of values taken in a data type that is associated with a constraint algebra. In METEO, these data types are C-types, while the subsets, which are ideals in KRS formalism, are δ-types.

The description logic system PROTODL provides the user with the ability of soundly defining new type descriptors. To add a new descriptor, the user has to write the code of some specific normalization operations. The extensibility property of METEO is comparable, in some extent, to the descriptor extensibility of PROTODL [BB92]. Indeed, both systems are aimed at providing the user with a customizable representation language, thus improving its general expressivity according to the specific requirements of an application domain. Descriptor extensibility could be achieved with the presence of METEO, but not along the same way it is done in PROTODL. Indeed, normalization is a process of METEO, rather than a process of the KRS. Therefore, to add a new descriptor would require

the modification of the typing process (interpretation of the KRS towards fields of δ-types) instead of the addition of some normalization operations.

7 Conclusion

The usual data types available in a knowledge representation system, namely the type collection of the KRS, are not always sufficient, especially for the development of knowledge bases that require the handling of complex data structures (*e.g.* molecular biology). This problem relies partly on type checking and completeness of inference processes within the knowledge bases.

The type system METEO has been designed and implemented in order to provide an object-based KRS with full type collection extensibility. This is achieved through the two cooperative levels of METEO. The first level (C-types) deals with data types implemented in the host language. The second level (δ-types) deals with subsets of values that are built in the knowledge representation language then involved in the KRS checking and inference processes. This second level of types actually distinguishes METEO from all other type systems designed for programming languages. It corresponds to the result of the actual typing of a knowledge base. Its implementation is distributed among C-types according to their specific properties for the expression of subsets.

On one hand, METEO is a polymorphic type system due to the hierarchical and generic organization of C-types. This means that new C-types may be added through dynamic predefined links that METEO sets up between the implementation of the data types in the host language and the TROPES KRS. When a new C-type is added in METEO, the handling of its computed δ-types is fully and automatically generated by METEO.

On the other hand, we pointed out the existence of an isomorphism between the δ-type level of METEO and the intensional part of the KRS, which deals with representation and reasoning on descriptions of entities. Indeed, through δ-types as knowledge entity types, METEO carries out the whole intensional part of the KRS, except for dynamic constraint checking.

As a consequence of the two connected levels of METEO, knowledge entities that are typed using a new C-type can be soundly and completely involved in the KRS checking and inference processes. It is one of the most important originalities of METEO, because no existing KRS is able to warrant the completeness of set-based reasoning processes as soon as a new user-defined data type is used in the knowledge base. The KRS, together with METEO, is now able to provide users with a way to design knowledge bases using both knowledge representation and programming language facilities and libraries (executable programs), in a safe and complete way. These results have been empirically tested through the implementation of METEO for the OBKRS TROPES.

The type system METEO could be adapted to any description logic system, provided that the translation from the knowledge representation language, namely the TBox, to EOLE remains complete. Yet, EOLE has been designed with the same expressive power as that of the TROPES representation language. The adaptation of METEO to a more expressive knowledge representation language may thus lead to a non-sound typing, unless EOLE gets extended appropriately. The important point is that the main organization of METEO can be safely of use to DLS.

Many extensions of METEO are in progress. Among them, METEO is currently extended in order to integrate the management of recursive definitions and regular expression. In addition, a deeper cooperation between METEO and the constraint management system MICRO [Gen95] is planned which will allow MICRO to benefit from the extensibility property of METEO.

References

[BB92] A. Borgida and R.J. Brachman. Protodl : a customizable knowledge base management system. In *1st CIKM*, pages 482–490, Baltimore (MA, US), 1992.

[BdBZ93] H. Balsters, R.A. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In O.M. Nierstrasz, editor, *7th European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 161–184, Kaiserslautern (Germany), July 1993. Springer-Verlag.

[BFL83] R.J. Brachman, R.E. Fikes, and H.J. Levesque. KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, pages 67–73, October 1983.

[BIM96] A. Borgida, C.L. Isbell, and D.L. McGuinness. Reasoning with black boxes: handling test concepts in CLASSIC. In *International Description Logics Workshop*, Cambridge (MA, US), November 1996.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.K. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD Conference*, San Francisco (CA, US), 1987.

[BMPS+91] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Resnick, and A. Borgida. Living with classic, when and how to use a kl-one-like language. In J.F. Sowa, editor, *Principles of Semantic Networks*, chapter 14, pages 401–456. Morgan Kaufmann, 1991.

[BPS94] A. Borgida and P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the classic description logic. *Journal of Artificial Intelligence Research*, 1:277–308, June 1994.

[Bra85] R.J. Brachman. "i lied about the trees", or defaults and definitions in knowledge representation. *AI Magazine*, 6(3):80–93, 1985.

[BS85] R.J. Brachman and J.G. Schmoltze. An overview of the KL-ONE knowledge representation language. *Cognitive Science*, 9:171–216, 1985.

[BW77] D.G. Bobrow and T.W. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):3–46, 1977.

[Cap94] C. Capponi. Interactive class classification using types. In E. Diday *et al.*, editor, *New Approaches in Classification and Data Analysis*, Studies in Classification, Data Analysis and Knowledge Organization, pages 204–211. Springer-Verlag, July 1994.

[Cap95] C. Capponi. Identification et exploitation des types dans un modèle de connaissances à objets. Thèse de doctorat, Université Joseph Fourier, Grenoble (France), October 1995.

[CE96] Y. Crampé and J. Euzenat. Révision interactive dans une base de connaissances à objets. In *10ème congrès Reconnaissances des Formes et Intelligence Artificielle*, pages 615–623, Rennes (France), January 1996. AFCET-AFIA.

[CEG95] C. Capponi, J. Euzenat, and J. Gensel. Objects, types and constraints as classification schemes. In *Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz (CA, US), August 1995.

[CM91] L. Cardelli and J.C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, March 1991.

[Dek94] L. Dekker. *FROME: représentation multiple et classification d'objets avec points de vue.* Thèse de doctorat, LIFL, Université des Sciences et Technologies de Lille, Lille (France), June 1994.

[DH91] R. Ducournau and M. Habib. Masking and conflicts, or to inherit is not to own! In D. Nardi M. Lenzerini and M. Simi, editors, *Inheritance Hierarchies in Knowledge Representation and Programming Language*, pages 223–244. John Wiley and Sons, 1991.

[Gai91] B.R. Gaines. Empirical investigations of knowledge representation servers: design issues and applications experience with krs. *SIGART bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):45–56, June 1991.

[Gai93] B.G. Gaines. A class library implementation of a principled open architecture knowledge representation server with plug-in data types. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 504–509, Chambéry (France), September 1993. Morgan Kaufmann.

[Gen95] J. Gensel. Contraintes et représentation de connaissances par objets. application au modèle TROPES. Thèse de doctorat, Université Joseph Fourier, Grenoble, France, October 1995.

[Ghe90] G. Ghelli. A class abstraction for a hierarchical type system. In *3rd ICDT*, volume 470 of *Lecture Notes in Computer Science*, Paris (France), December 1990.

[HJe92] P. Hudak, S.L. Peyton Jones, and P. Wadler (eds.). Report on the programming language haskell, version 1.2. *ACM SIGPLAN Notices*, 27(3), May 1992.

[HKQ+93] T. Hoppe, C. Kindermann, J.J. Quantz, A. Schmiedel, and M. Fischer. BACK *V5, tutorial and Manual.* Technische Universität Berlin, March 1993.

[ILO94] ILOG, Gentilly (France). *ILOG* TALK*, version 3.1 (Beta 1)*, 1994.

[ISX91] ISX Corporation. LOOM *users Guide, version 1.4*, August 1991.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42, May 1995.

[LR89] C. Lécluse and P. Richard. The o_2 database programming language. In G. Wiederhold P.M.G. Apers, editor, *15th International Conference on Very Large Data Bases*, pages 411–422, Amsterdam (NL), August 1989. Morgan Kaufmann.

[Mac91a] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J.F. Sowa, editor, *Principles of Semantics Networks*, chapter 13, pages 385–400. Morgan Kaufmann, 1991.

[Mac91b] R. MacGregor. Inside the LOOM classifier. *SIGART Bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):88–92, June 1991.

[MDW91] E. Mays, R. Dionne, and R. Weida. K-rep system overview. *SIGART Bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):93–97, June 1991.

[MRU90] O. Marino, F. Rechenmann, and P. Uvietta. Multiple perspectives and classification mechanism in object-oriented representation. In *9th European Conference on Artificial Intelligence*, pages 425–430, Stockholm (Sweden), August 1990. Springer-Verlag.

[Nap92] A. Napoli. Subsumption and classification-based reasoning in object-based representations. In *10th European Conference on Artificial Intelligence*, Vienna (Austria), August 1992. John Wiley and Sons.

[Neb90] B. Nebel. *Reasoning and revision in hybrid representation systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin (Germany), 1990.

[Neb91] B. Nebel. Terminological cycles. In John Sowa, editor, *Principles of semantic networks*, chapter 11, pages 331–361. Morgan Kaufmann, 1991.

[OMW96] F.J. Oles, E.K. Mays, and R.A. Weida. The algebraic essence of K-REP. In *International Description Logics Workshop*, Cambridge (MA, US), November 1996.

[Pag92] L. Pagdham. Defeasible inheritance: a lattice-based approach. *Computer Math. Applic.*, 23(6-9):527–541, 1992.

[PJ93] J. Peterson and M. Jones. Implementing type classes. *ACM SIGPLAN Notices*, 28(6):227–236, June 1993.

[Pro95] Projet SHERPA, INRIA Rhône-Alpes, Grenoble (France). TROPES, *version 1.0 reference manual*, June 1995.

[RBB+93] L.A. Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, and K.C. Zalondek. CLASSIC *description and reference manual for the* COMMON LISP *implementation, Version 2.2*, December 1993.

[Rec93] F. Rechenmann. Integrating procedural and declarative knowledge in object-based knowledge models. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 98–101, Le Touquet (France), October 1993.

[RN87] R. Fikes R. Nado. Semantically sound inheritance for a formally defined frame language with defaults. In *Sixth National Conference on Artificial Intelligence, AAAI-87*, volume 2, pages 443–448, July 1987.

[RU91] F. Rechenmann and P. Uvietta. SHIRKA*: an object-centered knowledge based management system*, pages 9–23. Artificial intelligence in numerical and symbolic simulation (SCS conference on artificial intelligence in numerical and symbolic simulation, Lyon (France), 1989). ALEAS, Lyon (France), 1991.

[SL83] J.G. Schmolze and T.A. Lipkis. Classification in the KL-ONE knowledge representation system. In *8th International Joint Conference on Artificial Intelligence*, Karlsruhe (Germany), 1983.

[Uni83] United States Department of Defense. *Reference Manual for the Ada Programming Language ANSI/-MIL-std 1815-a*, 1983.

[VE95] P. Valtchev and J. Euzenat. Classification of concepts through products of concepts and abstract data types. In *1st International Conference on Data Analysis and Ordered Structures*, pages 131–134, Paris (France), June 1995.

[Win85] T. Winograd. *Frame representation and the declarative/procedural controversy*, chapter 20, pages 357–370. Readings in Knowledge Representation. Morgan Kaufmann, Los Altos (CA, US), 1985.

[Woo75] W.A. Woods. *What's in a link: foundations for semantics networks*, pages 35–82. Representation and Understanding: Studies in Cognitive Science. Academic Press, 1975.

[Woo91] W.A. Woods. Understanding subsumption and taxonomy: a framework for progress. In J.F. Sowa, editor, *Principles of Semantic Networks*, chapter 1, pages 45–94. Morgan Kaufmann, 1991.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399