

# FROM TYPE SYSTEMS TO KNOWLEDGE REPRESENTATION: NATURAL SEMANTICS SPECIFICATIONS FOR DESCRIPTION LOGICS\*

ALEXANDER BORGIDA  
*Dept. of Computer Science, Rutgers University*  
*New Brunswick, NJ 08903, USA*  
*borgida@cs.rutgers.edu*

Received                      October 1991  
Revised                        January 1992

## ABSTRACT

We first explore the similarities and differences between concept definitions in *description/terminological logics* such as KL-ONE, Classic, Back, Loom, etc. and the types normally encountered in programming languages. The similarities lead us to consider the application of natural semantics – the mechanism most frequently used to describe type systems – to the definition of knowledge base management systems that use such description logics. The paper presents inference rules in the natural semantics style for a variety of judgments involving descriptions, such as “subsumption” and “object membership”, and provides the full definition of subsumption in the Classic KBMS as a proof system. One of our objectives is to document some advantages of this approach, including the utility of multiple complementary semantics, and especially the characterization of implementations that are computationally tractable but are incomplete relative to standard denotational semantics.

*Keywords:* Terminological logics; description logics; type systems; natural semantics; proof-theoretic semantics.

## 1. Introduction

A database or knowledge-base management system (KBMS) is charged with maintaining a computer model of some application domain. In order to describe and access such a model, the user needs a language – actually a collection of languages – to express updates, queries and even answers. In many cases, the foundations of such languages rest on viewing the world to be described as being populated by *individual objects* that are related to each other by binary relations, known as *properties/roles*; in addition, individual objects are grouped into abstract *classes*, that are themselves organized in the by-now familiar *subclass hierarchy*.

Semantic data models and object-oriented data bases are two examples of KBMS families that follow this general pattern. For example, the following class declaration would prescribe that objects stated to be instances of the class *PERSON* must also be instances of the class *LIVING\_THING* and must have exactly one filler for the attribute *name*, which belongs to the class *STRING*, and zero or more fillers for the attribute *children*, which must all belong to the class *PERSON*:

```
class PERSON is-a LIVING_THING with
    name [1,1] : STRING;
    children [0,*] : PERSON;
```

Such a class declaration basically expresses necessary conditions for members of the class, and is used to perform error checking and optimize physical storage allocation. Thus, a class functions more or less as a type in a standard Algol-like language.

---

\*TO APPEAR IN *INT. J. ON INTELLIGENT AND CO-OPERATIVE INFORMATION SYSTEMS 1(1)*, WORLD SCIENTIFIC PUBLISHING, SINGAPORE, 1992.

Another family of KBMS is based on so-called “KL-ONE style” languages, more recently termed *description logics (DLs)*, exemplified by KL-ONE [9], Nikl [15], Kandor [23], Back [30], Classic [6], and Loom [18], among others. What distinguishes this family is the fact that in addition to primitive classes, which behave more or less like the one above, it is possible to specify classes using intensional *descriptions* phrased in terms of the necessary and sufficient properties that must be satisfied by their instances. This is accomplished using a language for declaring primitive concepts and then combining them into composite ones. For example, consider the description in Figure 1, expressed in the Classic language [6].

```
(AND PERSON
  (AT-LEAST 1 degree)
  (ALL degree (ONE-OF Ba Bs Ms Phd)))
```

Figure 1: Description in Classic

This description delimits the set of all objects in the intersection (“AND”) of three sub-concepts:

**PERSON** — the identifier of a concept defined elsewhere (in this case a primitive);

**(AT-LEAST 1 degree)** — individuals with at least one filler for the *degree* role;

**(ALL degree (ONE-OF Ba, Bs, Ms, Phd))** — individuals whose *degree* fillers are all among the values *Ba, Bs, Ms, Phd*.

This description could be used, among others, as the definition of the concept *UNIVERSITY\_GRAD*. The key difference between such descriptions and the earlier class specifications is that such a definition can be used to *recognize* instances of the concept, or to *infer answers* to queries in situations where there is incomplete knowledge.

We therefore see an advance from the constrained utilization of class specification in semantic data models or object-oriented databases to KL-ONE style languages where concepts, viewed as composite terms/descriptions, can be reasoned with and are the source of inferences – hence the name “description logics”.

It is interesting to note that modern type systems have also evolved to include notions such as type inference/reconstruction and subtyping. The goal of this paper is to explore these similarities between two notions – types and descriptions – that have been developed entirely independently, and to exploit the analogy in at least one way: by applying a technique used in defining type systems to the definition of description logics.

In Section 2 we present a summary of the features of KBMS that use concept descriptions while in Section 3 we provide an in-depth analysis of their similarities to and differences from programming language types. In Section 4 we adapt the notation and technique of natural semantics for defining description logics, and after considering some of its advantages in Section 5, we apply this technique to the description of the Classic 1.0 knowledge base management system.

## 2. Description Logics and KBMS

Suppose we start with a database of individuals and their inter-relationships. The facts would be represented in the database by recording for every individual the objects that fill its various roles. For example, it may be known that Calvin has age 8, has among his enemies Susie and Moe, and has a Bachelor’s and a Master’s degree; this would be represented by the individual *Calvin* having filler 8 for the role *age*, the objects *Susie* and *Moe* as fillers for role *enemies*, and the values *Bs* and *Ms* as fillers for role *degree*.

Descriptions such as the one in Figure 1 can then be used in a KBMS to provide several services to users:

- *Query language:*

Since a query is just a definition of the properties required to be satisfied by the objects listed in the answer, it is entirely reasonable to treat descriptions as queries. As demonstrated in systems such as

Argon [26] and Candide [2], concepts can be used for information retrieval, returning as answer the set of individuals in the current database which satisfy the description.

For example, the description in Figure 1 could be seen as a query for individuals with at least one university degree; and individual *Calvin*, from above, would satisfy this query if it was also known that *Calvin* was an instance of concept *PERSON*.

- *Storing partial information about individuals:*

Arbitrary descriptions can be ascribed to an individual, thus providing considerable power in representing partial knowledge about individuals. For example, we can say that *JoeCool* received at least two degrees, all granted by *TahitiU*, without knowing the exact identity of those degrees, by ascribing the following term to him:

```
(AND
  (AT-LEAST 2 degree)
  (ALL degree
    (FILLS grantedBy TahitiU)))
```

In this case, **FILLS** is a description constructor used to characterize objects having as one of their fillers for the role *grantedBy* the value *TahitiU*. Note that this can be accomplished without having to name and define *a priori* the above description, as would have been the case if we tried to use view updates in a database environment.

- *Definition of concepts:*

If some description appears to be repeatedly useful, either as a query or as a descriptor of individuals, it can be given a name and then added to the knowledge base. (*PERSON* was an example of such a named concept used in Figure 1.) Thus the KBMS manages schema-like definitions with the same ease as individuals.

- *Subsumption of concepts:*

One of the most significant consequences of this approach is the fact that we can take two descriptions and decide if, by virtue of their definitions, the set of instances satisfying one would in all circumstances include the instances of the other (in which case the first term is said to **subsume** the other). For example, *UNIVERSITY\_GRAD* would subsume the concept below, which essentially describes someone with exactly 2 degrees, one of which is a *PhD*, and the other is either *Ba* or *Bs*<sup>1</sup>:

```
(AND PERSON
  (AT-LEAST 2 degree)
  (AT-MOST 2 degree)
  (FILLS degree Phd))
  (ALL degree (ONE-OF Ba Bs Phd)) )
```

This subsumption relation can be exploited to several ends:

- the system can be charged with maintaining the subclass hierarchy of concepts itself; because generalization/specialization is known to be an important form of data abstraction[5], this is useful in dealing with large collections of definitions;
- a query concept can retrieve individuals, such as *JoeCool* above, that were (incompletely) depicted with descriptions, by testing for subsumption;
- in case there are many queries, as in data exploration, the queries can themselves be *organized* in a subsumption hierarchy.

---

<sup>1</sup> This fact is not explicitly stated in the definition but it is a logical consequence since all (two) of the degrees must be in the set {Ba,Bs,Phd}, and one of these values (*Phd*) is already known.

- *Answer language:*  
As demonstrated in [6], concepts can be used as *intensional* descriptions of answers, thus providing an alternative to extensional listings of values.
- *Integrity checking:*  
The system can detect whether the properties ascribed to an individual object are inconsistent (e.g., if some person is required to have at most one degree, but is in fact being assigned more than one).
- *Concept incoherence and disjointness:*  
Given a description, the KBMS can determine if it is incoherent, in the sense that it cannot be ascribed to any individual, or if it is incompatible with some other description.

KBMS based on description logics have been used in a number of practical situations, including software information bases [12], financial management [19], configuration management [22], and data exploration. Additional signs that DLs are significant subjects of study are the several recent workshops on DLs.

### 3. Descriptions and Programming Language Types

#### 3.1. Types in Programming Languages

Logicians introduced the notion of “type” in order to avoid the antinomies that were threatening the foundations of mathematics. Thereafter, researchers in the foundations of computation, such as the lambda-calculus, and then programming languages, have used types as a way of avoiding nonsensical computations (e.g., adding an integer and a string). In fact, without much exaggeration, it can be said that type systems are used to judge valid those expressions where a function is applied only to values in its domain.<sup>2</sup>

The type system of a modern programming language starts with a few primitive types, such as *integer*, *real*, *character* and *boolean*, and type constructors (e.g., **ARRAY**, **LIST**, **FUNCN**) which can be combined orthogonally to create complex types, such as arrays of integers, or lists of functions from integers to boolean values; the latter would be denoted by the type expression **LIST**(**FUNCN**(*integer*, *boolean*)), or more commonly **LIST**(*integer* → *boolean*).

This brief description already allows us to make a number of observations about the similarities between types and descriptions:

- Both act as unary predicates: in one case over the space of values manipulated by the programming language, in the other, the set of individuals in the database.
- Both type and description systems have pre-defined primitives, such as integers.
- Both allow the definition of new primitive types, whose elements must be explicitly specified by users. In programming languages, these primitives are usually called abstract types (e.g., **STACK**), while in DLs they correspond to natural kinds found in the domain (e.g., **DOG**, **SPANIEL**) – concepts which cannot be defined. In both cases, it is possible to specify that instances of the new primitive type must have some *necessary* properties: stacks cannot be popped when empty, dogs must be animals. In modern type systems, for object-oriented languages for example, primitive types can be organized in a subtype hierarchy, just like primitive terms: a **DEQUEUE** is a subtype of **STACK**, since it has the stack’s operations plus more; **DOG** is a primitive subconcept of **ANIMAL**.
- Although the original KL-ONE language provided mostly operators to manipulate data structures corresponding to structured descriptions, the syntax of the definition in Figure 1 shows that in languages like Classic descriptions are in fact built up compositionally from identifiers using what we will call **description constructors**, such as **AND**, **AT-LEAST**, **ALL** and **ONE-OF**. Types have also have been recognized to have a compositional term-like structure, built up using type constructors.

We take a closer look at types and their utility, by surveying and paralleling their use with that of descriptions.

---

<sup>2</sup>See [10] for a good overview of recent progress in the field.

### 3.2. Similarities in Processing Types and Descriptions

#### 3.2.1. Type checking and integrity checking.

A programming language’s type system is used to ensure that programs do not perform erroneous operations. Traditionally, this is done by starting with a set of pre-defined operations and constants of known type, requiring the programmer to declare the type of all variables and programmer-defined functions, and then checking that every expression in the program is “well-typed”. For example, the expression  $(y + 1) * 4$  is incorrect (not well-typed) unless  $y + 1$  can be checked to have type integer, which itself depends on  $y$  having been declared to be of type integer (or some subtype of it); so if  $y$  had been declared boolean, then the above expression would raise a type error. Type checking, especially at compile time, is therefore a very significant tool for detecting programming errors.

The equivalent of type checking in DLs occurs when one asserts an individual to be an instance of some concept that carries necessary conditions for its membership. For example, if *Alice* is “declared” to have a description that includes the term **(AT-MOST 0 enemy)**<sup>3</sup>, but *Alice* is already known to be related to *RedQueen* by the role *enemy*, then the KB is inconsistent, i.e., it is in an erroneous state, and the declaration can be disallowed.

#### 3.2.2. Type coercion and concept instantiation.

In some programming languages, such as PL/I, implicit function calls are inserted by the compiler or interpreter in order to make an expression type correct, even when it is not so at the beginning. For example, in the expression  $y + 1$ , if  $y$  is of type boolean then a conversion function might first be invoked, so that the final expression being evaluated is in fact `convertBooleanToInteger(y)+1`.

The corresponding phenomenon occurs in the Classic KBMS: if *Alice* is asserted to have description **(ALL friends ANIMALS)**, and *Frodo* is known to be one of Alice’s friends (i.e., a filler for the *friends* role) then even if *Frodo* was not known to be an instance of the concept ANIMALS before, this fact is now asserted about it, in order to bring in line the knowledge about *Frodo* and the necessary properties of *Alice*. In a traditional object-oriented database, such an update would have been simply rejected as not satisfying the integrity constraint that all fillers of *friends* must be *known* instances of ANIMALS.

#### 3.2.3. Type inference and individual classification.

Note that in determining the type of the expression  $f(y)$ , it is redundant to know both the type of the domain of  $f$  and the type of  $y$ : if the domain of  $f$  is the type *STRING*, then  $y$  also needs to have type *STRING* in order for the expression to type check correctly. Languages such as ML use this to dispense with type declarations for variables by requiring primitive functions to have unique types and then “inferring/reconstructing” the type of the variables used in the program. For example, the function definition `function g(x) = x+2`; implies that  $x$  must be an integer, since it is being added to the integer constant 2; hence  $g$  has domain of type integer, and returns a value of type integer, since  $+$  has range integer; i.e., the type of  $g$  is inferred to be  $integer \rightarrow integer$ .

In other languages, such as Ada and CLU, function identifiers may have multiple meanings (they are **overloaded**), so that for example `size` applied to numbers computes absolute value, but when applied to strings it returns their length, etc. In such languages, the type of a function argument must be uniquely determined, and then the language implementation may “deduce” the correct type for the function, and hence the code to be actually invoked.

The process of type reconstruction can in fact be applied with both overloaded functions and partially declared variables, by essentially setting up type equations that must hold in order for the program to be correct and then solving these; if there is no unique solution, the program is considered ill-typed.

Similarly, DLs also infer the “type” of values being manipulated by the KB, through the process of individual classification: if *JoeCool* is known to be an instance of *PERSON* and has two fillers for the *degree* role, namely *Ba* and *PhD*, then *JoeCool* will automatically be recognized/inferred to be an instance of the concept *UNIVERSITY\_GRAD*, defined in Figure 1.

<sup>3</sup>Which means that *Alice* can have no fillers for the role *enemy*.

### 3.2.4. Subtypes and subsumption.

Programming language (PL) research has recently focused on the subtype relationship (sometimes known as subtype polymorphism). The basis of most definitions of subtype revolves around the notion of record subtype: a record type is an unordered collection of labeled and typed fields, and R1 is a subtype of R2 if R1 has at least those fields that R2 has, and for every such field p, its type in R1 is a subtype of p's type in R2. This is then used inductively to define a subtype relationship for other type constructors such as list, function, etc.: e.g., **LIST**( $\sigma$ ) is a subtype of **LIST**( $\tau$ ) iff  $\sigma$  is a subtype of  $\tau$ .

Subtyping strongly resembles the subsumption relationship in DLs. In fact, the **ALL** and **AND** constructors can be seen to define essentially record types. However, in DLs subsumption is usually determined by comparing directly the constructors appearing in the normalized forms of the concepts, where normalization makes explicit certain implicit constraints. In effect, for many DL description constructors subsumption is not defined inductively, e.g., (**AT-LEAST**  $n$  friend) subsumes (**AT-LEAST**  $m$  friend) iff  $n \leq m$ .

### 3.3. Some Differences between Types and Descriptions

The similarity between types and descriptions outlined above is not absolute. To begin with, there is a general bias in work on types to perform analyses statically, at compile time, rather than at run time. So, for example, type inference is performed before the program is run, and by the time the values manipulated by the program actually materialize there usually is no longer any explicit sign of the types. In contrast, the classification of individuals has the feeling of being an inherently “run-time” activity – if we think of run-time as the time when updates to the knowledge base occur – and the concepts under which individuals are classified coexist with the individuals being classified.

The following is a list of further distinctions that should be kept in mind:

- *Multiple typing*: In general, it is considered a highly desirable property for a programming language type systems to provide a unique, or at least “principal” type<sup>4</sup> to every value or expression. Thus type systems do not normally permit a value to be in several incomparable primitive types, in the way that an entity might be considered both a **STUDENT** and an **EMPLOYEE** (without having to define a new class of **STUDENT\_EMPLOYEES**). A particular effect of this philosophy is that primitive types are by default disjoint, though they may be made to contain each other through subtyping, while the extension of primitive concepts is not restricted in any way by default.
- *Composite types*: PLs usually concentrate on defining aggregate data structures (sets, lists, arrays) or functions from more primitive ones, and in this sense are **constructive**. In contrast, description constructors in DLs are more often set-theoretic in nature, creating objects whose denotations are subsets of previously existing sets, and can thus be called **restrictive**.
- *Evolution of state*: In DLs, the model of the world maintained by the KBMS is not assumed to be permanent or complete: one can at least acquire new information, and in some cases even be told that old information is out of date. For example, someone who was known to be a **STUDENT** can be later asserted to be a **GRADUATE\_STUDENT**, or a **SPORTS\_FAN**, or even said to stop being a **STUDENT**, and continue being only a **PERSON**. In PLs, changes are modeled by re-binding identifier names, and one normally does not speak of a stack also becoming some other kind of value. In other words, state changes do not lead to type changes *for the same value*.
- *Type non-membership*: Type systems do not usually pass judgments about objects **not** having certain type. This is in part due to the above convention that types are either disjoint or subtypes, so that knowing the principal type of a value also tells us all the types it is not in. In DLs, the question of whether some value  $b$  has type  $T$  can have 3 answers: *Yes*, *No*, and *Unknown* – the latter case arising when the state of the KB could evolve in different directions, where the answer could be *Yes* or *No* respectively. For example, if *Frodo* is a **DOG**, and **DOGS** have been declared to be disjoint from **HORSES**, then *Frodo* is not a **HORSE**; but *Frodo* may or may not be a **FRIENDLY-ANIMAL**.

---

<sup>4</sup>A single type from which all others can be obtained by some formal operation such as substitution.

### 3.4. Individuals, Objects and Records

Knowledge bases primarily record relationships between objects through the notion of a binary relationship – the role. The closest notion in PLs is the idea of *records*, which have named attributes that can store other values. At first glance, it would seem easy to equate the two notions, and thus make the record type

$$[name : String; age : Integer]$$

be equivalent to the description

$$(\mathbf{AND} (\mathbf{ALL} \textit{ name STRING})(\mathbf{ALL} \textit{ age INTEGER})).$$

However, there are a surprising number of differences between the two notions which should be kept in mind:

- *Object vs. record identity:*

Programming language records are normally assumed to be identical if they have the same fillers for their attributes – i.e., they have the same *structure*. Thus in Pascal or SML two records — N.B. not *pointers* to records — with fields `name` and `age` having the same values ('Bob' and 23 say) will be determined to be equal.

Objects, on the other hand, have an intrinsic, immutable identity established at creation time, which is not dependent in any way on the values of roles and attributes (somewhat like pointers in PLs).

- *Role domains:* Record types specify exactly to what objects it is legal to apply some attribute, in order to get or set its filler. For example, it is illegal to try to evaluate `JoeCool.studentNumber`, unless Joe is known to have type `[studentNumber :  $\alpha$ ]` for some  $\alpha$ , which represents the information that Joe does have a field for `studentNumber`. In contrast, several DLs assume that every role is applicable to every object, although there may be ways to specify that some role is not to have fillers for some class of objects. This leads to complications in modeling, where one needs to specify explicitly, in an employee knowledge base for example, that departments, addresses, products, etc. do not have *salary* attribute fillers.
- *Multiple role fillers:* In most DLs a role can have zero or more fillers, which are considered to form a set (rather than a multiset): for example, a person can have 0 or more *siblings*. PL records on the other hand have exactly one attribute filler in every case. If the type system of the language is rich enough, this one value can be a set/list (which itself can be empty or not). The DLs' view of roles as binary relations rather than functions leads to a number of other differences.
- *Cardinality restrictions as term constructors:* Since a role can have zero or more fillers, there are several term constructors based on the number of fillers (e.g., `(AT-MOST 2 parents)`) or number of fillers of a certain kind (e.g., `(A-MIN-OF 1 parents WAGE_EARNER)`). Essentially these are short forms for rather complex quantified formula schemas in First Order Predicate Calculus.
- *Role hierarchies and constructed roles:* Since roles can have multiple fillers, it is possible to express certain constraints about roles which resemble constraints about primitive classes: all sisters and brothers are siblings and relatives of a person, so *sister* and *brother* are specified to be **subroles** of *sibling*, which in turn is a subrole of *relative*.

Just as one can have primitive and composite concepts, DLs allow primitive and composite roles – i.e., roles can be also be represented by terms. Composite role constructors might include **inverse**, **range**, **androle**, and **transitive\_closure**. For example, the term

$$(\mathbf{AT-LEAST} \ 2 \ \mathbf{inverse}(\mathbf{parent}))$$

refers to objects with at least 2 children – objects which are parent fillers of at least two other objects; and the term `(range sibling FEMALE)` can be used to define the “sister” relationship, which might itself then be used in the description

(**ALL** (range sibling FEMALE) Doctor)

characterizing objects with all sisters (who are the female siblings) being doctors.

As a result of the previous two ideas, DLs support both a concept/type hierarchy and a role/attribute hierarchy, with concomitant notions of subsumption.

- *Incomplete individual descriptions*: In order to further support the incomplete description of some state of affairs, DLs sometime allow roles to be specified as “open” (as opposed to “closed”) thereby indicating that some further fillers may be added later. In such situations certain type judgments must therefore be deferred, e.g., we cannot tell if an individual has at most 3 role fillers for some role until that role is declared or inferred to be “closed”.

## 4. Natural Semantics and the Definition of DLs

The preceding section has provided evidence that type systems in programming languages have much in common with DLs. This similarity can be exploited by applying techniques developed in one field to problems arising in the other. In this paper, we will focus on one such case of “technology transfer”, relating to the specification of the semantics of DLs.

In addition to the existing DLs, new ones are being proposed continually depending on the particular application being contemplated (e.g., reasoning about time [28] or plans [13]), or on the desired efficiency of classification and subsumption [8, 24].

Any DL (new or old) needs to be defined clearly and unambiguously. This is essential if users are to describe the concepts that they mean and get the inferences they desire, and if implementors are to know what is expected of them.

Originally, the definition of a language was embodied in its interpreter, which of course was a difficult “manual” to consult. More recently, the semantics of DLs have been specified model theoretically, by assigning set-theoretic denotations to various constructors (e.g., [8, 24]). For example, the denotation of the **ALL** construct would be expressed by

$$\mathcal{D}[(\mathbf{ALL} \ p \ C)] = \{x | \forall y (x, y) \in \mathcal{D}[p] \Rightarrow y \in \mathcal{D}[C]\}$$

As an alternative, we propose to explore here *proof-theoretic* definitions for DLs. In particular, based on their resemblance to types in programming languages, we will adapt for describing DLs the notation of *Natural Semantics*, presented most fully in [16].

A definition will consist of a collection of inference rules and axioms allowing sentences of the form  $\langle \textit{antecedent} \rangle \vdash \langle \textit{consequent} \rangle$ , called **sequents**, to be deduced. For example, the following is a typical inference rule dealing with types:

$$\frac{\rho \vdash f : \sigma \rightarrow \eta, \quad \rho \vdash e : \sigma', \quad \vdash \sigma' \ll \sigma}{\rho \vdash f(e) : \eta}$$

It essentially states that (in some environment  $\rho$ , which assigns types to variables) if  $f$  is a function whose domain is type  $\sigma$ , and  $e$  is an expression of type  $\sigma'$ , which itself is a subtype of  $\sigma$ , then the application of function  $f$  to  $e$  will have type  $\eta$ .

A sequent can deal with a variety of different **judgments** (i.e., predicates) in its consequent, and these judgments are usually presented using an infix notation for ease of reading. For example, the above rule contained two kinds of judgments: one about the type of expressions (e.g.,  $e : \sigma'$ ) used the colon symbol (:), another about the subtype relationship between types (e.g.,  $\sigma' \ll \sigma$ ) used  $\ll$ .

Every rule has a numerator – an unordered collection of formulae – and a denominator – a single sequent. Intuitively, the rule permits the construction of a proof tree yielding the denominator from proof trees yielding the numerator formulae. Rules can have variables and the denominator of a rule consists of sequents as well as general conditions usually involving the variables in the rule.

When there are many rules, it becomes convenient to collect them into localized sets, forming named mini-theories, which can be referred to by appending the name as a superscript of the  $\vdash$  symbol.

In applying this approach to DLs, we will convert the LISP style notation of descriptions to the more usual term notation, where the constructor appears as a functor before the parenthesis. Thus the definition in Figure 1 will be written as

*and*(PERSON, atleast(1, degree), all(degree, oneof(Ba, Bs, Ms, Phd)))

With this notation, we can present a simple rule dealing with the **subsumption judgment** ( $\vdash C \implies D$ ) involving the **ALL** constructor:

$$\frac{\vdash c \implies d}{\vdash \text{all}(p, c) \implies \text{all}(p, d)}$$

The rule, involving variables  $c, d$  and  $p$ , expresses the fact that an **ALL**-restriction is more specialized if the concept restricting the role is itself more specialized.

A number of very important term constructors, including **AND** are associative; rather than clutter the axiom sets with rules for this property, we shall allow such constructors to have as arguments sequences of values, and will assume some underlying mechanism which collapses unnecessary nestings of such operators. The ellipsis sign  $\dots$  will be used to match the remaining elements of such a sequence, once we have singled out some elements which are of interest. Similarly, there are operators for which the sequence can be treated as an unordered set (e.g., **ONE-OF**, as well as **AND**). Once again, we shall assume that there is some way of declaring such facts a priori, to avoid the need for commutativity axioms.

To begin with, we shall concentrate on the subsumption relationship between pairs of descriptions, which will be expressed by the judgment  $\vdash C \implies D$ .

#### 4.1. A small example: Subsumption Rules for a CoreDL

To begin with, we give a syntax for a language CoreDL that is a subset of most description logics discussed in the literature:

```
<descr> ::=
  THING                /* the universal concept */
  NOTHING              /* the empty concept */
  and( <descr> + )     /* conjunction */
  all( <role> , <descr> ) /* restriction on role fillers */
  atleast( <integer> , <role> ) /* minimum number of fillers */
  atmost( <integer> , <role> ) /* maximum number of fillers */
  prim(<identifier>)   /* primitive concept */

<role> ::= <identifier>
```

This language is somewhat more powerful than the  $\mathcal{FL}^-$  language presented in [8]: through the use of number restrictions, it can actually express conflicting constraints that lead to inconsistent concepts.

Figure 2 shows one possible set of rules for determining the subsumption relationship between descriptions in CoreDL. In its rules,  $c, d$ , and  $e$  will be variables ranging over concepts, while  $p, q, r$  range over role identifiers. Each rule is labeled (on the left side) to make future references to it easier, and has some commentary on the right, in small font. The rules are presented in two lists. Intuitively, the first set of rules expresses what is sometimes known as “structural subsumption”: for every constructor, we present rules indicating when another description built using that constructor describes a more specific concept. The second set of rules expresses equivalences that are useful for “massaging” descriptions into *normal forms* such that the structural subsumption rules can be applied directly.

isa-REF	$\vdash c \Longrightarrow c$	Reflexivity of subsumption
isa-TRANS	$\frac{\vdash c \Longrightarrow d, \vdash d \Longrightarrow e}{\vdash c \Longrightarrow e}$	Transitivity of subsumption
isa-NOUGHT	$\vdash NOTHING \Longrightarrow c$	Empty set
isa-THING	$\vdash c \Longrightarrow THING$	Universal set
isa-AND-RT	$\frac{\vdash c \Longrightarrow d, \vdash c \Longrightarrow \text{and}(ee)}{\vdash c \Longrightarrow \text{and}(d,ee)}$	Intersection on the right side of ISA
isa-AND-LFT	$\frac{\vdash c \Longrightarrow e}{\vdash \text{and}(c,\dots) \Longrightarrow e}$	Intersection on the left side
isa-ALL	$\frac{\vdash c \Longrightarrow d}{\vdash \text{all}(p,c) \Longrightarrow \text{all}(p,d)}$	Specializing the role range leads to a more specialized concept
isa-MIN	$\frac{n > m}{\vdash \text{atleast}(n,p) \Longrightarrow \text{atleast}(m,p)}$	Higher lower bounds are more restrictive, hence lower in subsumption
isa-MAX	$\frac{n > m}{\vdash \text{atmost}(m,p) \Longrightarrow \text{atmost}(n,p)}$	Higher upper bounds are less restrictive
isa-PRIM	$\vdash \text{prim}(id) \Longrightarrow \text{prim}(id)$	A primitive only subsumes itself. Redundant, given rule REF.

Figure 2: (a) Theory  $ISA_0$ : structural subsumption rules for CoreDL

AND0	$\vdash \text{and}() \equiv THING$	Intersection of empty collection is the universal set
MIN0	$\vdash \text{atleast}(0, p) \equiv THING$	Zero as lower bound is no constraint
ALL-THG	$\vdash \text{all}(p, THING) \equiv THING$	The universal set is no restriction
MAX0	$\vdash \text{atmost}(0, p) \equiv \text{all}(p, NOTHING)$	If you can't have any fillers, all fillers must belong to the empty set
ALL-AND	$\vdash \text{and}(\text{all}(p, c), \text{all}(p, d), \dots) \equiv \text{all}(p, \text{and}(c, d), \dots)$	<b>AND</b> is distributive over <b>ALL</b>
M-MAX	$\frac{n > m}{\vdash \text{and}(\text{atleast}(p,n), \text{atmost}(m,p), \dots) \equiv NOTHING}$	Inconsistent lower and upper bounds

Figure 2: (b) Theory  $ISA_0$ : Equivalence rules for simplification and inconsistency in CoreDL

In order to shorten the presentation of the rules, we have introduced a second judgment, namely  $\vdash C \equiv D$ , concerning the equivalence of two descriptions. This judgment is in fact derived from subsumption:  $\vdash C \equiv D$  if and only if  $\vdash C \Rightarrow D$  and  $\vdash D \Rightarrow C$ .

The rules in Figure 2 define in a “generative” manner what the subsumption relationship is: anything we can conclude using the rules of inference.

The following is an example of a (back-chaining) proof using the above rules, showing that the description (**AND (PRIM person), (AT-MOST 0 degree)**) is subsumed by (**ALL degree FUNNY**)<sup>5</sup>. Every line is annotated by the name of the rule used to derive it as a consequence of the lines at the next indentation level below it, which themselves may have subproofs. For brevity, we will use the string *Nongrad* to represent the term *and(prim(person), atmost(0, degree))*. Then we have

<i>Judgment</i>	<i>Rule used</i>
$\vdash \text{Nongrad} \Rightarrow \text{all}(\text{degree}, \text{FUNNY})$	isa-AND-LFT
$\vdash \text{atmost}(0, \text{degree}) \Rightarrow \text{all}(\text{degree}, \text{FUNNY})$	TRANS
$\vdash \text{atmost}(0, \text{degree}) \Rightarrow \text{all}(\text{degree}, \text{NOTHING})$	MAX0
$\vdash \text{all}(\text{degree}, \text{NOTHING}) \Rightarrow \text{all}(\text{degree}, \text{FUNNY})$	isa-ALL
$\vdash \text{NOTHING} \Rightarrow \text{FUNNY}$	isa-NOUGHT

The above specification of subsumption happened not to use the “antecedent” part of sequents. If we allowed identifiers to be associated with descriptions, e.g., *NO\_DEGREES*  $\cong$  *atmost(0, degree)* then we could put such definitions in an “environment”  $\rho$ , in whose context subsumption would take place. We would then be able to prove  $\rho \vdash \text{NO\_DEGREES} \Rightarrow \text{all}(\text{degree}, \text{NOTHING})$ . More interestingly, we may allow declarations which state that certain primitive concepts are said by the user to be related in a subclass lattice (e.g., *DOG*  $\triangleright$  *ANIMAL*). Such facts would then be gathered in an environment, and may be exploited by some inference rule of the form

$$\frac{\overset{\text{lattice}}{\rho \vdash \alpha \triangleright \beta}}{\rho \vdash \alpha \Rightarrow \beta}$$

where  $\overset{\text{lattice}}{\vdash}$  is axiomatized in a separate theory to derive transitivity facts from the lattice.

#### 4.2. Rules for Recognizing Instances

Suppose that the database can store two kinds of facts about individuals: fillers for roles and membership in concepts/descriptions. We will encode the contents of the database as a sequence of terms of the form *hasfiller*(*(individual), (role), (value)*) and *isin*(*(individual), (concept)*), where *hasfiller* and *isin* are term constructors that only appear in the data base, rather than arbitrary descriptions.

For example, we may know the following sequence of facts

hasfiller(joy, friend, bob) · isin(bob, atleast(home, 1))  
· hasfiller(bob, age, 25) · hasfiller(joy, friend, mary)

To reason about individuals, we introduce the judgment  $\mathcal{D} \vdash b \longrightarrow C$  indicating that individual  $b$  is a known instance of description  $C$  in the context of database facts  $\mathcal{D}$ .

Clearly the classification of individuals depends on the values in the database of facts, so we must be able to “access” it somehow. To do this, we axiomatize **retrieval from the database**,  $\overset{\text{get}}{\vdash}$ , so that we can find out the set of all known fillers for a role and the conjunction of descriptions which have been explicitly asserted to hold of the individual. The former is recorded by a predicate of the form  $\langle \text{individual} \rangle \xrightarrow{\phi} \text{fillers}(\langle \text{role} \rangle, \langle \text{set of values} \rangle)$ , where we use a new, special term-constructor *fillers*, which does not appear in the description language itself. For example, from the above database we would like to be able to conclude

$$\text{db} \overset{\text{get}}{\vdash} \text{joy} \xrightarrow{\phi} \text{fillers}(\text{friend}, \{\text{bob}, \text{mary}\})$$

<sup>5</sup>The meaning of FUNNY is irrelevant here.

### Rules for gathering filler information

$empty \vdash x \xrightarrow{\phi} fillers(p, \{\})$	Start out with no known fillers
$\frac{\rho \vdash x \xrightarrow{\phi} fillers(p, \{aa\})}{hasfiller(x,p,b) \cdot \rho \vdash x \xrightarrow{\phi} fillers(p, \{b,aa\})}$	accumulate
$\frac{\rho \vdash x \xrightarrow{\phi} fillers(p,jj)}{isin(\_,\_) \cdot \rho \vdash x \xrightarrow{\phi} fillers(p,jj)}$	Ignore description membership information
$\frac{\rho \vdash x \xrightarrow{\phi} fillers(p,jj), \quad y \neq x \text{ or } q \neq p}{hasfiller(y,q,\_) \cdot \rho \vdash x \xrightarrow{\phi} fillers(p, \{jj\})}$	Ignore fillers for other roles/objects

### Rules for gathering descriptor information

$\rho \vdash x \xrightarrow{\delta} and()$	Every object is in class <b>THING</b>
$\frac{\rho \vdash x \xrightarrow{\delta} and(cc)}{isin(x,d) \cdot \rho \vdash x \xrightarrow{\delta} and(d,cc)}$	Accumulate descriptions
$\frac{\rho \vdash x \xrightarrow{\delta} and(cc), \quad y \neq x}{isin(y,d) \cdot \rho \vdash x \xrightarrow{\delta} and(cc)}$	Ignore information about other individuals
$\frac{\rho \vdash x \xrightarrow{\delta} and(cc), \quad y \neq x}{hasfiller(y,\_,\_) \cdot \rho \vdash x \xrightarrow{\delta} and(cc)}$	Ignore information about other individuals

Figure 3: Theory **GET**<sub>0</sub>: rules for gathering database facts

To collect the object's explicitly asserted membership in a most specific concept we use the predicate  $\langle individual \rangle \xrightarrow{\delta} \langle description \rangle$ , as in the following statement that should be derivable from the above database:

$$db \stackrel{get}{\vdash} bob \xrightarrow{\delta} and(atleast(home, 1))$$

Figure 3 then shows the rules in the subtheory **GET** performing the above inferences.

Using this, we can now present in Figure 4 rules for determining if some individual is an instance of some concept in the CoreDL language. Although at first glance this would appear to be correct, the rules concerning **AT-MOST** and **ALL** are in fact not valid since there is no guarantee that the GET-theory is used to obtain *all possible fillers* for the role before applying rules like in-MAX<sub>0</sub>; therefore such a rule could ignore some fillers to jump to unwarranted conclusions. Essentially, we are confronting the problem of the “closed world assumption” – when do we know everything? To deal with this problem, we introduce another term that can be derived from the database, namely  $allfillers(\langle role \rangle, \langle set \ of \ fillers \rangle)$ , which describes the complete set of fillers for the role for some individual and then replace rules in-MAX<sub>0</sub> and in-ALL<sub>0</sub> by the following rules

	$\frac{\text{db} \vdash^{\text{get}} x \longrightarrow c}{\text{db} \vdash x \longrightarrow c}$	Obtain info from the database
	$\frac{\text{db} \vdash x \longrightarrow c, \vdash^{\text{isa}} c \Longrightarrow d}{\text{db} \vdash x \longrightarrow d}$	Membership is transitive via Isa
in-THING	$\text{db} \vdash x \longrightarrow \text{THING}$	Redundant: provided by theory GET and and()
in-AND	$\frac{\text{db} \vdash x \longrightarrow c, \text{db} \vdash x \longrightarrow \text{and}(cc)}{\text{db} \vdash x \longrightarrow \text{and}(c,cc)}$	<b>AND</b> is like intersection
in-MIN	$\frac{\text{db} \vdash^{\text{get}} x \xrightarrow{\phi} \text{fillers}(p,jj), \text{card}(jj)=n}{\text{db} \vdash x \longrightarrow \text{atleast}(n,p)}$	MIN considers cardinality of role fillers
in-MAX <sub>0</sub>	$\frac{\text{db} \vdash^{\text{get}} x \xrightarrow{\phi} \text{fillers}(p,jj), \text{card}(jj)=n}{\text{db} \vdash x \longrightarrow \text{atmost}(n,p)}$	MAX also considers cardinality – but see below for correction!
in-ALL <sub>0</sub>	$\frac{\text{db} \vdash^{\text{get}} x \xrightarrow{\phi} \text{fillers}(p,jj), \text{db} \vdash jj \longrightarrow c}{\text{db} \vdash x \longrightarrow \text{all}(p,c)}$	If fillers are known to be in c then the <b>ALL</b> constraint is satisfied
in-PRIM	— — —	No rule: membership in a primitive concept must be stated in the db.

Figure 4: Theory **IS-IN**<sub>0</sub>: rules for deducing membership in a description

$$\begin{array}{l}
\text{in-MAX} \quad \frac{\text{db} \vdash \mathbf{x} \xrightarrow{\text{get } \phi} \text{allfillers}(\mathbf{p}, \mathbf{jj}), \quad \text{size}(\mathbf{jj}) = n}{\text{db} \vdash \mathbf{x} \longrightarrow \text{atmost}(n, \mathbf{p})} \\
\text{in-ALL} \quad \frac{\text{db} \vdash \mathbf{x} \xrightarrow{\text{get } \phi} \text{fillers}(\mathbf{p}, \mathbf{jj}), \quad \text{db} \vdash \mathbf{x} \longrightarrow \text{atmost}(\text{size}(\mathbf{jj}), \mathbf{p}), \quad \text{db} \vdash \mathbf{jj} \longrightarrow \mathbf{c},}{\text{db} \vdash \mathbf{x} \longrightarrow \text{all}(\mathbf{p}, \mathbf{c})}
\end{array}$$

which consider only those cases where all possible fillers are known.

To be able to determine *allfillers*, we need some marker in the database to indicate that we have complete knowledge of these fillers. One way to proceed is to “terminate” each database with a marker CWA which permits making the closed world assumption at that point – i.e., all information is now known to the knowledge base. (The terminator must therefore appear at the front of the sequence of facts, since this sequence is built up as a queue from the empty sequence.) In this case, the rules for theory GET need to be augmented by

$$\frac{\rho \vdash \mathbf{x} \xrightarrow{\phi} \text{fillers}(\mathbf{p}, \mathbf{jj})}{\text{CWA} \cdot \rho \vdash \mathbf{x} \xrightarrow{\phi} \text{allfillers}(\mathbf{p}, \mathbf{jj})} \quad \begin{array}{l} \text{Close the role when en-} \\ \text{countering CWA marker} \end{array}$$

An alternative technique is to allow each role for each individual to be declared closed separately. In this case, the database would also contain assertions of the form *closed*((*individual*), (*role*)) (e.g., *closed*(*bob*, *age*)), and the theory GET would have to be augmented with the following rules

$$\begin{array}{l}
\frac{\rho \vdash \mathbf{x} \xrightarrow{\phi} \text{fillers}(\mathbf{p}, \mathbf{jj})}{\text{closed}(\mathbf{x}, \mathbf{p}) \cdot \rho \vdash \mathbf{x} \xrightarrow{\phi} \text{allfillers}(\mathbf{p}, \mathbf{jj})} \quad \begin{array}{l} \text{Close the role if encoun-} \\ \text{tering marker} \end{array} \\
\frac{\rho \vdash \mathbf{x} \xrightarrow{\delta} \alpha}{\text{closed}(\_, \_) \cdot \rho \vdash \mathbf{x} \xrightarrow{\delta} \alpha} \quad \begin{array}{l} \text{Ignore info about role} \\ \text{closing for concept mem-} \\ \text{bership} \end{array} \\
\frac{\rho \vdash \mathbf{x} \xrightarrow{\phi} \beta}{\text{closed}(\_, \_) \cdot \rho \vdash \mathbf{x} \xrightarrow{\phi} \beta} \quad \begin{array}{l} \text{Pass out filler informa-} \\ \text{tion too} \end{array}
\end{array}$$

Note that neither of these techniques truly implement the default meaning of “closed world assumption”, namely that “*there are no more fillers other than the ones that we can deduce*”. The distinction would arise in cases where fillers other than the ones explicitly present in the database could be inferred from the rules of reasoning about individuals (see Appendix A). In such cases “closing the knowledge bases” is an auto-epistemic operation, which normally requires access to the “*not provable*” relation (as in Prolog’s negation-by-failure mechanism).

## 5. Some Advantages of the Deductive Approach

### 5.1. Multiple Semantics

In many areas of computer science dealing with “languages” there appear to be multiple ways of describing formal objects:

- formal logic: the semantics of a logic, its axiomatization, and theorem-proving algorithms all characterize the set of “valid/true formulae”, but in different ways;
- the type system of a programming language can be described by the denotational semantics of types (e.g., [20]), by type deduction rules of the kind illustrated in Section 4. (e.g., [10]), and by the type checking algorithm implemented in the compiler (e.g., [1]);

- the semantics of a programming language can be presented, among others, in the denotational framework, or as axiomatic rules for verification, or as operational semantics describing state changes [14].

As argued in [14] and elsewhere, it is beneficial to provide multiple ways to define the meaning of languages because by approaching the subject from different angles we are less likely to leave things out. If we can in fact prove the equivalence of the various formalizations, we have also significantly increased our confidence in the correctness, as well as completeness of our various definitions. Multiple semantics are also useful if the various descriptive mechanisms have differing strengths and weaknesses: where one might be obscure the other may be more concise or clear. Natural semantics specifications complete this picture for DLs, were we already had denotational specifications, and implementations (that are essentially specialized theorem-provers).

Finally, in all of the above areas it is generally felt that developing implementations – which is our ultimate goal as computer scientists – is easier **after** some kind of axiomatization has occurred. Personal experience with several language features (e.g., combining equality and number constraints in Classic) indicates that this will be the case for the subsumption and classification algorithms in DLs.

We remark here that proof-theoretic descriptions of certain DLs have been recently presented in [27]. Those rules axiomatize reasoning about equality with concepts (though not with individuals) in languages that are proven to be equivalent in expressive power to classical logics such as modal and dynamic logics; hence the axiomatization is in some sense inherited from formal logic, with no special adaptation for DLs. The approach presented therein is less than fully general because it relies on the existence of a concept-complementation constructor (which leads to intractable reasoning), it reasons directly with concept equality rather than subsumption, and most significantly, it does not reason about individuals.

### 5.2. Rapid Prototyping

One of the potential advantages of the proof-theoretic approach presented here is the possibility of developing rapid prototypes for reasoners using DLs, by essentially translating the inference rules into a theorem-proving system augmented by some guidance. This approach appears to have been tried successfully with natural semantics specifications of programming languages [11], though it does necessitate adherence to certain constraints on the forms of the rules. This topic remains open for future study.

### 5.3. Characterizing Incomplete Reasoners

Starting with [8], there have been a series of results indicating that reasoning with many of the constructors naturally encountered in modeling applications is an inherently intractable problem.

For example, adding a constructor **SOME**, which essentially provides the power of a sorted existential quantifier — (**SOME** friend CHILDLESS) denotes objects with at least one filler for the *friend* role that is an instance of *CHILDLESS* — is known to make the subsumption problem NP-hard [21]. The reason is basically that if one concept has several such restrictions that are incompatible then one can begin to reason about the role requiring many different fillers; for example, having some friend being childless, and some friend who is a parent, implies at least two friends. In general, one may then have to check exponentially many combinations of terms constructed with **SOME**.

Given that such constructs are still desired in the language, yet we wish to have reasonably efficient algorithms, we can implement a subset of the inferences warranted by classical semantics. The crucial difficulty here is how to characterize the implemented set to the users of the system – or how to describe the inferences that will not be made by the system, though they would be entailed by the obvious semantics of the construct.

One solution would be to find some *non-standard denotational semantics*, as in [24] for example, under which complete reasoning is in fact tractable. Unfortunately, in some situations no intuitive interpretation can be found for the purely mathematical devices, thus rendering this kind of semantics rather useless.

Axiomatic descriptions on the other hand may provide a clear way to characterize the deductions made by an implementation which is not complete with respect to the standard semantics because of computational intractability.

So, for example, the following subsumption rules for the **SOME** concept constructor express what are its more obvious semantics:

### (Incomplete) Inference Rules for SOME

isa- SOME	$\frac{\vdash C \implies D}{\vdash \text{some}(p,C) \implies \text{some}(p,D)}$
SOME- NIL	$\vdash \text{some}(p, \text{NOTHING}) \equiv \text{NOTHING}$
SOME- 1	$\vdash \text{some}(p, \text{THING}) \equiv \text{atleast}(1, p)$
ALL-1- SOME	$\vdash \text{and}(\text{atleast}(1, p), \text{all}(p, C), \dots) \implies \text{some}(p, C)$
SOME- AND	$\vdash \text{and}(\text{some}(p, C), \text{all}(p, D), \dots) \implies \text{some}(p, \text{and}(C, D))$
SOME- MAX-1	$\vdash \text{and}(\text{some}(p, C), \text{atmost}(1, p), \dots) \implies \text{all}(p, C)$

The above set of rules is incomplete: there are logical consequences of the intuitive meaning of the **SOME** constructor that are not captured. For example, any individual satisfying the condition

(**AND**

(**SOME** friends (**AT-MOST** 0 children))  
 (**SOME** friends (**AT-LEAST** 1 children)))

would have to have at least two fillers for the *friends* role; hence this description should be subsumed by (**AT-LEAST** 2 friends).

Therefore a more complete reasoner would also implement something like the following inference rule:

$$\frac{\vdash \text{and}(C_i, C_j) \implies \text{NOTHING} \quad 1 \leq i, j \leq r \quad \text{but the } C_i, C_j \text{ are coherent}}{\vdash \text{and}(\text{some}(p, C_1), \text{some}(p, C_2), \dots) \implies \text{atleast}(r, p)}$$

Even more generally, we may recognize, as in [21], that while pairs of concepts may not be disjoint, 3 or more terms may be conjoined to yield an inconsistent concept. Therefore the previous rule could be replaced by one where in the antecedent, instead of using the terms  $C_i$  directly, we partition them into clusters, which themselves are internally consistent but are pairwise inconsistent.

A set of axioms, if properly chosen, may also be able to point out the consequences of different (denotational) semantics. For example, suppose we added a constructor that allowed the specification of mutually disjoint primitive concepts:  $\text{disjPrim}(\langle \text{identifier} \rangle, \langle \text{groupId} \rangle)$  (as in *Classic*, for example). The intuitive meaning of such a construct would be that  $\text{disjPrim}(\text{DOG}, \text{animal})$  and  $\text{disjPrim}(\text{CAT}, \text{animal})$  would be assumed to have no common instances. For the traditional semantics, this would result in the addition of the following subsumption rule:

$$\frac{x \neq y}{\vdash \text{and}(\text{disjPrim}(x, g), \text{disjPrim}(y, g), \dots) \implies \text{NOTHING}}$$

On the other hand, in the 4-valued semantics of Patel-Schneider [24], contradictions of this sort do not arise — one of the truth values essentially corresponds to contradictory information having been given — and in fact no further rule would need to be added, as disjoint primitives would behave just like primitives.

## 6. Applications to the Classic KBMS

### 6.1. The Classic KBMS

The *Classic* system is a DL developed at AT&T Bell Laboratories and used in AT&T on a variety of projects, both research and industrial applications. It is also available for free use in educational institutions. We

propose to use the mechanisms developed above to describe the reasoning performed (or not performed) by the standard system (version 1.0), in order to demonstrate the efficacy of the approach advocated in this paper in a realistic setting.

The complete syntax and semantics of Classic 1.0 is presented in Appendix A; we will content ourselves with highlighting here a couple of the novel aspects of Classic over and above those present in CoreDL.

Concerning the syntax of Classic 1.0, we note the following additions:

- The constructors **FILLS** (and **ONE-OF**), already introduced in the text, are permitted in Classic.
- The **SAME-AS** constructor describes individuals which are restricted to have the same filler for the two chains of roles given to it as arguments. For example, (**SAME-AS** *mother* *mother-spouse-spouse*) expresses the condition that the filler of the *mother* role is the same as the *spouse* of the *spouse* of the *mother*. The roles in the two paths are required to have exactly one filler; such roles are called *attributes* in Classic.

## 6.2. Rules of Inference for Subsumption in Classic 1.0

As usual, there are many different sets of axioms that specify the same theory. We have chosen to present here rules that parallel as much as possible the operations performed by the interpreter of the Classic system. We are currently exploring whether this might have the advantage of making it easier to explain to the user the actions and error messages of the system. At the same time we have tried to be systematic in the search for rules, in the hope that this will lead to a more complete semantics.

Thus, we recommend for every constructor one or more axioms defining the “structural subsumption” relationship. The following are illustrative rules for the **FILLS** and **SAME-AS** constructors:

isa- FILLS	$\frac{\{\text{ii}\} \subseteq \{\text{jj}\}}{\vdash \text{fills}(p, \text{jj}) \implies \text{fills}(p, \text{ii})}$	Requiring a larger set of fillers is more restrictive.
isa- SAME	$\vdash \text{sameas}(pp, qq) \implies \text{sameas}(pp \cdot r, qq \cdot r)$	Equalities extend to the right

Next, there are rules that define special equivalences which can be used to put a concept into a normal form where only the structural subsumption rules need to be tried. Some of these rules define equivalences with the special constants representing the universal or empty concepts:

EQ-TH	$\vdash \text{sameas}(pp, pp) \equiv C - \text{THING}$	Reflexivity of identity for roles
FIL- NIL	$\frac{\text{not}(\{\text{ii}\} \subseteq \{\text{jj}\})}{\vdash \text{and}(\text{fills}(p, \text{ii}), \text{all}(p, \text{and}(\text{oneof}(\text{jj}), \dots))) \equiv \text{NOTHING}}$	The required set of fillers has to be a subset of the set of potential fillers.

while others reorganize the terms:

FIL- AND	$\vdash \text{and}(\text{fills}(p, \text{ii}), \text{fills}(p, \text{jj}), \dots) \equiv \text{and}(\text{fills}(p, \text{ii} \cup \text{jj}), \dots)$	<b>AND</b> of <b>FILLS</b> is like union
ALL- EQ	$\vdash \text{sameas}(p \cdot qq, p \cdot rr) \equiv \text{all}(p, \text{sameas}(qq, rr))$	This rule crucially depends on roles being functions

Finally, there are rules that may infer possibly stronger constraints about a concept from its components, and hence are used to obtain a normal form which makes explicit some conditions that are implicit in the description’s constraints. (Again, we can make the search more systematic by taking every constructor and looking for rules that imply its presence (e.g., **ONE=FIL** below), or are implied by its presence (e.g., **FIL-SIZ**):

FIL-SIZ	$\frac{\text{size}(\{ii\})=n}{\vdash \text{fills}(p,ii) \implies \text{atleast}(n,p)}$	<p>Knowing a subset of the possible set of fillers for a role gives a lower bound on the number of fillers.</p>
ONE=FIL	$\frac{\text{size}(\{ii\})=n}{\vdash \text{and}(\text{all}(p,\text{oneof}(ii)), \text{atleast}(n,p), \dots) \implies \text{fills}(p,ii)}$	<p>If one needs <math>n</math> fillers and the superset containing the fillers has <math>n</math> elements, then we know exactly all the fillers.</p>
FIL-EQ	$\frac{\vdash \text{and}(\text{sameas}(pp \cdot p, qq \cdot q), \text{all}(pp, \text{fills}(p, b)), \dots)}{\implies \vdash \text{all}(qq, \text{fills}(q, b))}$	<p>If two role paths are identical then their fillers must be identical at the end.</p>

Note that such inference rules could also be presented as normalization rules about equivalent concepts, by taking a judgment of the form  $c \implies d$  and replacing it by the judgment  $c \equiv \text{and}(d, c)$

All rules concerning the semantics of reasoning about individuals in *Classic* have been relegated to Appendix A3.. Finally, we remark that the *Classic* system supports additional operations (such as retraction of told facts) and triggers (forward chaining rules) which are not strictly part of the description *logic*, and have not been formalized here.

### 6.3. Incomplete Reasoning in *Classic*

It may be worth pointing how the preceding has put into practice the advantages claimed for natural semantics in Section 5.

Considering the subsumption rules, we have shown in [4] that *Classic* individuals appearing in definitions with **ONE-OF** require case-by-case reasoning and make the subsumption problem become NP-hard. The source of the difficulty can be glimpsed by trying to decide whether the description (**ALL friends CHILDLESS**) subsumes or not (**ALL friends (ONE-OF JoeCool)**); intuitively, this decision depends on the properties of *JoeCool*: is he or isn't he childless? Unfortunately such case analysis explodes combinatorially with larger sets of individuals.

The design of *Classic* has consciously chosen to avoid such reasoning, in part because it seems somewhat unsettling to have the hierarchy of definitions depend on accidental facts in the world. The point is that the rules presented in Section A2. describe the subsumption rules actually implemented in the system. As it happens, this version of subsumption can also be given a denotational semantics, with a non-standard treatment of individuals as “sets of potential referents”. This semantics is presented in [25] and can be proven to be equivalent to the natural semantics presented here. We therefore have an example of complementary semantics.

This match with the denotational semantics comes however at a certain price: some standard inferences about subsumption do not require analyzing properties of individuals, but have not been implemented because they are not supported by the modified semantics in [25]. Users can be warned about such missing inferences by presenting rules that are not part of the definition. For example, according to the intuitive semantics of **SAME-AS**, if we can deduce a description contains the conjunction of (**FILLS bestFriend Moe**) and (**FILLS worstEnemy Moe**), where *bestFriend* and *worstEnemy* are attributes, then in fact this description entails (**SAME-AS bestFriend worstEnemy**). More generally, if two paths  $rr$  and  $qq$  are equal, and at the end of path  $qq$  attribute  $p$  must be filled by the same value,  $b$ , as does attribute  $s$  at the end of path  $rr$ , then the paths  $qq \cdot p$  and  $rr \cdot s$  would also be equal. This can be elegantly expressed by the rule

$$\vdash \text{and}(\text{sameas}(qq, rr), \text{all}(qq, \text{fills}(p, b)), \text{all}(rr, \text{fills}(s, b)), \dots) \implies \text{sameas}(qq \cdot p, rr \cdot s)$$

If we were to implement such inferences, then the natural semantics rules would seem to remain the only way to describe cleanly and formally the *Classic* KBMS.

Similar considerations apply to *Classic*'s incomplete reasoning about individuals.

## 7. Conclusions

We have presented a comparison of some similar notions occurring in the type theory of programming languages and KBMS based on descriptions/terminologies. In particular, we have argued that despite entirely disparate origins, modern “types” resemble modern “descriptions” in many ways, including (i) grouping elements of a value space, (ii) having a term-like compositional nature, and (iii) being related by sub-type/subsumption relationship. Moreover the processes that involve types, such as type checking, type coercion, type inference and subtyping, have close correlates in the domain of description language knowledge bases, such as integrity checking, propagation, individual recognition, and subsumption.

We have used this analogy to apply some of the machinery of PL types to the process of describing DLs and their (incomplete) implementations. In particular, we adapted the mechanism of natural semantics to the specification of DLs, and claimed that this helps in providing additional insight into new languages, and may be the only means to characterize succinctly reasoners that, because of computational intractability, are incomplete with respect to their obvious denotational semantics

To buttress this claim, we presented an annotated set of inference rules characterizing the reasoning performed by the Classic system, version 1.0, released by AT&T Bell Laboratories — an implementation that is in fact incomplete with respect to the obvious semantics.

This paper has concentrated on the use of inference rules in the style of natural semantics for the description of DLs. In this context, a significant topic to explore is the appropriate choice of axiomatizations for DLs. In mathematics, the shortest axiomatization is preferred because it makes inductive proofs of meta-theorems easier. In our field, brevity of axiomatization is a disadvantage: we want to use the rules of inference to explain to language users what computations the system will perform, or how it has arrived at an answer. Therefore an important area of research will be finding different styles of axiomatizations, and empirically deciding which are better for various purposes, including rapid prototyping and explanation. The present work must also be extended to apply the natural semantics approach to DLs where there are not only primitive but also composite roles, with their own role constructors and classification hierarchy (as in the original KL-ONE language).

For the purposes of explanation, as well as closed-world reasoning relating to role fillers, it appears necessary to reason about non-derivability in DLs. This is of course a harder and more difficult issue than derivability, and model-based refutation as well as other approaches need to be explored in this context.

More generally, we plan to explore further the synergies that can be obtained from the similarity of descriptions and types, such as their similarity to terms with variables.

**Acknowledgments:** I am deeply grateful to Ron Brachman for giving me the opportunity to join him in the Classic adventure. I also thank Ravi Sethi for exposing me to Natural Semantics – without that exposure, this paper would not have been conceived. Finally, I have received very useful comments from Klaus Schild, Peter Patel-Schneider, Robert MacGregor and Tim Griffin.

## References

- [1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: principles, tools and techniques*, Addison-Wesley, 1986.
- [2] H. Beck, H. Gala and S. Navathe, "Classification as a query processing technique in the CANDIDE semantic model" in *Proc. Data Engineering Conf.*, Los Angeles, CA, 1989, 572–581.
- [3] A. Borgida, "Modeling class hierarchies with contradictions", in *Proc. ACM SIGMOD Conference*, Chicago, IL, 1988, 434–443.
- [4] A. Borgida, "Terminologic Frames as Types: Inference rules and prospective applications", Technical Report, Department of Computer Science, Rutgers University, May 1991.
- [5] A. Borgida, J. Mylopoulos and H.K.T. Wong, "Generalization/ Specialization as a basis for software specification", in *On Conceptual Modelling*, M. Brodie et al. (eds.), Springer Verlag, 1984, 87–114.
- [6] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick, "CLASSIC: A structural data model for objects", in *Proceedings ACM SIGMOD-89*, Portland, Oregon (1989) 58–67.
- [7] R. J. Brachman, R. E. Fikes and H. J. Levesque, "Krypton: A functional approach to knowledge representation", *IEEE Computer, Special Issue on Knowledge Representation* **16** (10) (1983) 67–73.
- [8] R. J. Brachman and H. J. Levesque, "The tractability of subsumption in frame-based description languages", in *Proceedings AAAI-84*, Austin, Texas (1984) 34–37.
- [9] R. J. Brachman and J. G. Schmolze, "An overview of the KL-ONE knowledge representation system", *Cognitive Science* **9** (2) (1985) 171–216.
- [10] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Computing Surveys*, **17**(4), 1985, 471–522.
- [11] T. Despeyroux, "Executable specification of static semantics", *Semantics of Data Types*, LNCS Vol. 173, June 1984.
- [12] P. Devanbu, P. Selfridge, B. Ballard and R.J. Brachman, "Steps towards a knowledge-based software information system", *Proc. IJCAI-89*, Detroit, MI, August 1989.
- [13] P. Devanbu and D. Litman, "Plan-based terminological reasoning", *Proc. KR'91*, Boston, MA., 1991.
- [14] C.A.R. Hoare and P.E. Lauer, "Consistent and complementary formal theories of the semantics of programming languages", *Acta Informatica*, **3**(2), 1974, 135–154
- [15] T. S. Kaczmarek, R. Bates and G. Robins, "Recent developments in NIKL", in *Proceedings AAAI-86*, Philadelphia, Pennsylvania (1986) 978–985.
- [16] Kahn, G. "Natural Semantics", Rapport de Recherche No. 601, INRIA, Sophia Antipolis, France.
- [17] H. J. Levesque and R.J. Brachman, "Expressiveness and tractability in knowledge representation and reasoning", *Computational Intelligence* **3** (2) (1987) 78–93.
- [18] R. M. MacGregor, "A deductive pattern matcher", in *Proceedings AAAI-87*, St. Paul, Minnesota (1987) 403–408.
- [19] E. Mays, C. Apte, J. Griesmer, J. Kastner. "Organizing knowledge in a complex financial domain", *IEEE Expert*, Fall 1987, 61–70.
- [20] D.B. MacQueen, G.D. Plotkin and R. Sethi, "An ideal model for recursive polymorphic types", in *Proceedings POPL-86*, Salt Lake City, Utah, pp.165–174.
- [21] B. Nebel, "Computational complexity of terminologic reasoning in BACK", *Artificial Intelligence* **34**(3), April 1988.

- [22] B. Owsnicki-Klewe, “Configuration as a Consistency Maintenance Task”, in *Proc. GWAI-88*, W. Hoepfner (ed.), Springer Verlag Berlin, September 1988, pp. 77-87.
- [23] P. F. Patel-Schneider, “Small can be beautiful in knowledge representation”, in *Proceedings IEEE Workshop on Principles of Knowledge-Based Systems*, Denver, Colorado (1984) 11–16. Extended version appears as AI Technical Report No. 37, Schlumberger Palo Alto Research, Palo Alto, California (1984).
- [24] P. F. Patel-Schneider, “A four-valued semantics for terminological logics”, *Artificial Intelligence* **38** (1989) 319–351.
- [25] P. F. Patel-Schneider, and A. Borgida, “A non-classic semantics for CLASSIC”, *Unpublished manuscript*, AT&T Bell Laboratories, 1989.
- [26] P. F. Patel-Schneider, R. J. Brachman and H. J. Levesque, “ARGON: Knowledge representation meets information retrieval”, in: *Proceedings First Conference on Artificial Intelligence Applications*, Denver, Colorado (1984) 280–286.
- [27] K. Schild, “A Correspondence Theory for Terminological Logics: Preliminary Report”, *Proc. IJCAI’91*, Sydney, Australia, August 1991, 464–471.
- [28] A. Schmiedel, “A Temporal Terminologic Reasoner”, in *Proceedings AAAI-90*, Boston, MA, August 1990, 641–645.
- [29] M. Schmidt-Schauss, “Subsumption in KL-ONE is undecidable”, in *Proceedings KR ’91*, Toronto, Canada, May 1989, 421–431.
- [30] K. von Luck, B. Nebel, C. Peltason and A. Schmiedel, “The anatomy of the BACK system”, *KIT (Kunstliche Intelligenz und Textverstehen) Report 41*, Technical University of Berlin, Berlin, F.R.G. (1987).

## A The Natural Semantics of Classic 1.0

### A1. Syntax

The following are additional constructs, not appearing in CoreDL:

- The constructors **FILLS** and **ONE-OF** already introduced in the text, are permitted in Classic.
- The **SAME-AS** constructor describes individuals which are restricted to have the same filler for the two chains of roles given to it as arguments. For example, **SAME-AS**( *mother*, *mother-spouse-spouse*) expresses the condition that the filler of the *mother* role is the same as the *spouse* of the *spouse* of the *mother*. The roles in the two paths are required to have exactly one filler; such roles are called attributes in Classic.
- The test constructors provide an escape from the expressive limitations of the language: they allow the instances of the concept to be recognized by a host-language (e.g., LISP or C) boolean function. However, this function behaves like a black-box as far as subsumption is concerned.
- Primitive concepts are not simply atomic identifiers, but instead can be defined to be subconcepts of other concepts – i.e., they can be given necessary conditions. Similarly for disjoint primitive concepts.

```
<concept> ::=
  THING                /* the universal concept */
  C-THING              /* all classic objects */
  H-THING              /* all host individuals */
  NOTHING              /* the empty concept */
  and( <concept> + )   /* conjunction */
  all( <role> , <concept> ) /* restriction on role fillers */
  atleast( <integer> , <role> ) /* minimum cardinality on fillers */
  atmost( <integer> , <role> ) /* maximum cardinality on fillers */
  sameas(<attr-path> , <attr-path>) /* equality of paths */
  oneof( <individual> + ) /* enumerated concept */
  fills( <role>, <individual> + ) /* concept with required
                                   fillers for role */
  c-test(f)            /* test-defined classic concept */
  h-test(f)            /* test-defined host concept
  prim(<concept> , <id>) /* primitive concept */
  disj-prim(<concept>,<gp-id>,<id>) /*concept that is disjoint
                                   from others in the same group */

<role> ::= <identifier>
<attr-path> ::= <identifier> | <identifier> . <attr-path>
```

### A2. Subsumption axioms

We use the following notational conventions for variable names appearing in rules: *c, d, e* are concepts; *ee* is a list of concepts; *p, q, r* are roles; *pp, qq, rr, vv, ww* are chains of roles; *k, n, m* are integers; *ii, jj* are lists of individuals.

## STRUCTURAL SUBSUMPTION

isa- NOUGHT	$\vdash \text{NOTHING} \Rightarrow c$	Empty set
isa- THING	$\vdash c \Rightarrow \text{THING}$	Universal set
isa- AND- RT	$\frac{\vdash c \Rightarrow d, \vdash c \Rightarrow \text{and}(ee)}{\vdash c \Rightarrow \text{and}(d, ee)}$	Intersection on the right side of ISA
isa- AND- LFT	$\frac{\vdash c \Rightarrow e}{\vdash \text{and}(c, \dots) \Rightarrow e}$	Intersection on the left side
isa-ALL	$\frac{\vdash c \Rightarrow d}{\vdash \text{all}(p, c) \Rightarrow \text{all}(p, d)}$	Specializing the role range leads to a more specialized concept
isa-MIN	$\frac{n \geq m}{\vdash \text{atleast}(n, p) \Rightarrow \text{atleast}(m, p)}$	Higher lower bounds are more restrictive, hence lower in subsumption
isa- MAX	$\frac{n \geq m}{\vdash \text{atmost}(m, p) \Rightarrow \text{atmost}(n, p)}$	Higher upper bounds are less restrictive
isa- ONE	$\frac{(\{ii\} \subseteq \{jj\})}{\vdash \text{oneof}(ii) \Rightarrow \text{oneof}(jj)}$	Subsumption is just subset for <b>ONE-OF</b>
isa- FILLS	$\frac{(\{ii\} \subseteq \{jj\})}{\vdash \text{fills}(p, jj) \Rightarrow \text{fills}(p, ii)}$	Conversely, requiring a larger set of fillers is more restrictive.
isa- SAME	$\vdash \text{sameas}(pp, qq) \Rightarrow \text{sameas}(pp \cdot r, qq \cdot r)$	Equalities extend to the right
isa- PRIM	$\vdash \text{prim}(C, k) \Rightarrow C$	Primitives are subclasses of their parent class
isa- DISJPRIM	$\vdash \text{disjprim}(C, g, k) \Rightarrow C$	Ditto for disjoint primitives
isa-C- TEST	$\vdash c\text{-test}(f) \Rightarrow C\text{-THING}$	c-test concepts have Classic individuals as instances
isa-H- TEST	$\vdash h\text{-test}(f) \Rightarrow H\text{-THING}$	h-test concepts have host individuals as instances
REF	$\vdash c \Rightarrow c$	Reflexivity of ISA
TRANS	$\frac{\vdash c \Rightarrow d, \vdash d \Rightarrow e}{\vdash c \Rightarrow e}$	Transitivity of ISA

## NORMALIZATION RULES

And0=thing	$\vdash \text{and}() \equiv \text{THING}$	Intersection of empty collection is the universal set
All-thing	$\vdash \text{all}(p, \text{THING}) \equiv C - \text{THING}$	
Eq-thing	$\vdash \text{sameas}(pp, pp) \equiv C - \text{THING}$	Reflexivity of identity for roles
Eq-sym	$\vdash \text{sameas}(pp, qq) \equiv \text{sameas}(qq, pp)$	Symmetry of identity relationship
Fil-one=nil	$\frac{\text{not}(\{ii\} \subset \{jj\})}{\vdash \text{and}(\text{fills}(p, ii), \text{all}(p, \text{and}(\text{oneof}(jj), \dots))) \equiv \text{NOTHING}}$	The required set of fillers has to be a subset of the set of potential fillers.
Min-max=nil	$\frac{n \geq m}{\vdash \text{and}(\text{atleast}(n, p), \text{atmost}(m, p), \dots) \equiv \text{NOTHING}}$	Inconsistent lower and upper bounds
Cl-host=nil	$\vdash \text{and}(C\text{-THING}, H\text{-THING}, \dots) \equiv \text{NOTHING}$	Host and Classic domains do not mix
Disj=nil	$\frac{\vdash c \equiv d, n \neq k}{\vdash \text{and}(\text{disjprim}(c, g, n), \text{disjprim}(d, g, k), \dots) \equiv \text{NOTHING}}$	Distinct elements of the same partition group are disjoint
One0=nil	$\vdash \text{oneof}() \equiv \text{NOTHING}$	The empty set and NOTHING are identical
All-eq	$\vdash \text{sameas}(p \cdot qq, p \cdot rr) \equiv \text{all}(p, \text{sameas}(qq, rr))$	This rule crucially depends on roles being functions
All-and	$\vdash \text{and}(\text{all}(p, c), \text{all}(p, d), \dots) \equiv \text{and}(\text{all}(p, \text{and}(c, d)), \dots)$	<b>AND</b> is distributive over <b>ALL</b>
One-and	$\vdash \text{and}(\text{oneof}(ii), \text{oneof}(jj), \dots) \equiv \text{and}(\text{oneof}(ii \cap jj), \dots)$	<b>AND</b> of <b>ONE-OF</b> s is just set intersection
Fil-and	$\vdash \text{and}(\text{fills}(p, ii), \text{fills}(p, jj), \dots) \equiv \text{and}(\text{fills}(p, ii \cup jj), \dots)$	<b>AND</b> of <b>FILLS</b> is like union
MIN-AND	$\frac{n \geq m}{\vdash \text{and}(\text{atleast}(n, p), \text{atleast}(m, p), \dots) \equiv \text{and}(\text{atleast}(n, p), \dots)}$	Redundant, but explains implementation.
MAX-AND	$\frac{n \geq m}{\vdash \text{and}(\text{atmost}(n, p), \text{atmost}(m, p), \dots) \equiv \text{and}(\text{atmost}(m, p), \dots)}$	Not necessary, but explains implementation.
PRIM-ID	$\frac{\vdash c \equiv d}{\vdash \text{prim}(c, k) \equiv \text{prim}(d, k)}$	Primitiveness is in the index.
DPRIM-ID	$\frac{\vdash c \equiv d}{\vdash \text{disjprim}(c, g, k) \equiv \text{prim}(d, g, k)}$	Ditto.
MAX0	$\vdash \text{atmost}(0, p) \equiv \text{all}(p, \text{NOTHING})$	If you can't have any fillers, all fillers must belong to the empty set

## COMBINANT INFERENCE

ONE-SIZ	$\frac{size(\{ii\}) = n}{\vdash all(p, oneof(ii)) \implies atleast(n, p)}$	Knowing a superset of the set of fillers gives an upper bound on the number of fillers
FIL-SIZ	$\frac{size(\{ii\}) = n}{\vdash fills(p, ii) \implies atleast(n, p)}$	Similarly for knowledge about elements in the set
FIL=ONE	$\frac{size(\{ii\}) = n}{\vdash and(fills(p, ii), atleast(n, p), \dots) \implies all(p, oneof(ii))}$	With n fillers, where n is the upper bound, all fillers must be known.
ONE=FIL	$\frac{size(\{ii\}) = n}{\vdash and(all(p, oneof(ii)), atleast(n, p), \dots) \implies fills(p, ii)}$	If one needs n fillers and the superset containing the fillers has n elements, then we know exactly all the fillers.
EQ-SUBS	$\frac{\vdash and(sameas(vv \cdot qq, rr), sameas(vv, ww), \dots)}{\implies sameas(ww \cdot qq, rr)}$	Substitution of equals for equals is allowed. This captures the transitivity of <b>SAME-AS</b>
ALL-EQ	$\vdash and(sameas(pp, qq), all(pp, c)) \implies all(qq, c)$	If two role paths are identical, their restrictions must match.
FIL-EQ	$\frac{\vdash and(sameas(pp \cdot p, qq \cdot q), all(pp, fills(p, b)), \dots)}{\implies \vdash all(qq, fills(q, b))}$	If two role paths are identical then their fillers must be identical at the end.

### A3. Reasoning about Individuals in Classic 1.0

We rely on the GET-subtheory presented earlier to retrieve information about individuals stored in the database, which appears as an environment for the sequents. We remind the reader that the database (or inference rules presented later on) provide information about an individual's membership in concepts, fillers for roles, and whether some role is closed or not. Also note that in the rules below, we shall use membership in the term  $and(fills(p, \alpha), atmost(size(\alpha), p))$  as a condition identifying when all the fillers of role  $p$  are known. (Other equivalent conditions, involving  $all(p, oneof(jj))$ , can be obtained using concept subsumption rules.) This is a somewhat subtle point because one would be tempted to continue using  $allfillers$  to check whether all the fillers are known. However, in Classic, it is possible for the KBMS to *infer* that some role cannot have any more fillers, without this being explicitly stated in the database.

#### A3.1. Recognition of Individuals

The rules for determining whether an individual satisfies a description are again divided into groups: those that describe when an individual satisfies the conditions of a particular constructor, and those that allow new database facts to be inferred about individuals – so-called derivation rules. Moreover, we will need an additional sub-theory in order to be able to gather information about fillers of *role-paths* for roles that are functions.

## RECOGNIZING INSTANCES

	$\frac{db \stackrel{get}{\vdash} x \xrightarrow{\delta} c}{db \vdash x \rightarrow c}$	Obtain info from database
in-isa	$\frac{db \vdash x \rightarrow c, \vdash c \Rightarrow d}{db \vdash x \rightarrow d}$	Membership is inherited via Isa
in-THING	$db \vdash x \rightarrow THING$	Redundant: provided by GET and and()
in-AND	$\frac{db \vdash x \rightarrow c, db \vdash x \rightarrow and(cc)}{db \vdash x \rightarrow and(c, cc)}$	<b>AND</b> is like intersection
in-FILLS	$\frac{db \stackrel{get}{\vdash} x \xrightarrow{\phi} fillers(p, jj)}{db \vdash x \rightarrow fills(p, jj)}$	Fillers give straightforward information about <b>FILLS</b>
in-MIN	$\frac{db \vdash x \rightarrow fills(p, jj), \quad size(jj) = n}{db \vdash x \rightarrow atleast(n, p)}$	
in-MAX	$\frac{db \stackrel{get}{\vdash} x \xrightarrow{\phi} allfillers(p, jj), \quad size(jj) = n}{db \vdash x \rightarrow atmost(n, p)}$	This rule considers cardinality of closed roles. Atmost can also be derived using IS-A rule ONE-SIZE
in-ALL1	$\frac{db \vdash x \rightarrow and(fills(p, jj), atmost(n, p)), \quad n = size(jj)}{db \vdash x \rightarrow all(p, oneof(jj))}$	Knowing all the fillers gives a <b>ONE-OF</b> restriction on the role. Redundant with ISA rule FIL=ONE.
in-ALL	$\frac{db \vdash x \rightarrow all(p, oneof(jj)), \quad db \vdash jj \rightarrow c}{db \vdash x \rightarrow all(p, c)}$	If all possible fillers are known to be in c then <b>ALL</b> constraint is satisfied
in-TEST	$\frac{f(x) \text{ evaluates to true in database } db}{db \vdash x \rightarrow test(f)}$	
in-ONEOF	$\frac{\{jj\} \subseteq \{ii\}}{db \vdash jj \rightarrow oneof(ii)}$	Membership in <b>ONE-OF</b> is easy
in-SAME	$\frac{db \stackrel{path}{\vdash} x \rightarrow pathfills(pp, y), \quad db \stackrel{path}{\vdash} x \rightarrow pathfills(qq, y)}{db \vdash x \rightarrow sameas(pp, qq)}$	Assumes auxiliary rules for computing fillers of paths – see below

## AUXILIARY RULES FOR PATH FILLERS

$\frac{db \vdash x \longrightarrow \mathit{fills}(p, y)}{db \stackrel{\mathit{path}}{\vdash} x \longrightarrow \mathit{pathfills}(p, y)}$	Base case
$\frac{db \stackrel{\mathit{path}}{\vdash} x \longrightarrow \mathit{pathfills}(pp, y), db \stackrel{\mathit{path}}{\vdash} y \longrightarrow \mathit{pathfills}(qq, z)}{db \stackrel{\mathit{path}}{\vdash} x \longrightarrow \mathit{pathfills}(pp \cdot qq, z)}$	Transitivity of pathfills
$\frac{db \vdash x \longrightarrow \mathit{all}(pp, \mathit{fills}(q, y))}{db \stackrel{\mathit{path}}{\vdash} x \longrightarrow \mathit{pathfills}(pp \cdot q, y)}$	Through <b>ALL</b> restrictions, it is possible to find the filler at the end of a role path without knowing the intermediate fillers.
$db \vdash \mathit{all}(p, \mathit{all}(qq, \mathit{fills}(r, y))) \equiv \mathit{all}(p \cdot qq, \mathit{fills}(r, y))$	Definition of $\mathit{all}(pp, \mathit{fills}...)$

The KBMS can also infer new information about other individuals by “propagating” constraints on role restrictions, as discussed in Section 3.2.2.. The inference rule for this is:

### PROPAGATION RULE

$\frac{ALL\text{-prop} \quad db \vdash x \longrightarrow \mathit{fills}(p, y), db \vdash x \longrightarrow \mathit{all}(p, c)}{db \vdash y \longrightarrow c}$	Propagates info about a role filler based on <b>ALL</b> restrictions for the role.
--	--

### A3.2. Integrity Checking in Classic

The Classic system provides error messages when the knowledge base contains contradictory information about an individual  $b$  (denoted by the judgment  $\vdash \bowtie b$ ) — for example, when a role has more fillers than allowed by an **AT-MOST** restriction on it. The main inference rule concerns membership in the empty concept:

$$\frac{db \vdash x \longrightarrow \mathit{NOTHING}}{db \vdash \bowtie x}$$

This rule is sufficient for most cases because inconsistent facts will derive membership in disjoint concepts which are then conjoined by rule in-AND.

There are however additional cases where we get inconsistency, and some of them are tied to the open-world assumption. For this reason, we found it easiest to present another judgment, representing “known-not-to-be-an-instance-of” ( $\not\rightarrow$ ). Its axioms include:

## DERIVING NONMEMBERSHIP

notin- DISJ	$\frac{db \vdash x \longrightarrow c, \vdash \text{and}(c, d) \implies \text{NOTHING}}{db \vdash x \not\rightarrow d}$	Disjointness of terms implies non-membership
notin- NOTHING	$db \vdash x \not\rightarrow \text{NOTHING}$	
notin- AND	$\frac{db \vdash x \not\rightarrow c}{db \vdash x \not\rightarrow \text{and}(c, dd)}$	
notin- ALL	$\frac{db \vdash x \longrightarrow \text{fills}(p, y), db \vdash y \not\rightarrow c}{db \vdash x \not\rightarrow \text{all}(p, c)}$	If even one filler is known not to be in $c$ then <b>ALL</b> constraint is not satisfied
notin- TEST	$\frac{f(x) \text{ evaluates to false in database } db}{db \vdash x \not\rightarrow \text{test}(f)}$	
notin- ONEOF	$\frac{x \notin \{ii\}}{db \vdash x \not\rightarrow \text{oneof}(ii)}$	
notin- SAME	$\frac{\begin{array}{l} db \stackrel{\text{path}}{\vdash} x \longrightarrow \text{pathfills}(pp, y), \\ db \stackrel{\text{path}}{\vdash} x \longrightarrow \text{pathfills}(qq, z), y \neq z \end{array}}{db \vdash x \not\rightarrow \text{sameas}(pp, qq)}$	

Rules for non-membership for the other constructors can be stated but they are redundant since their effect can be obtained using rule notin-DISJ and the membership rules from the preceding section.

Finally, we can now add an additional rule for finding inconsistency:

$$\frac{db \vdash x \longrightarrow c, db \vdash x \not\rightarrow c}{db \vdash \bowtie x}$$

By the construction of our rules, we will now detect contradictory information about individuals through membership in the inconsistent concept or inconsistent membership.

### A3.3. On the Closing of Roles

The above set of rules interprets the presence of `closed(x, p)` in the database to signify that all the fillers of the role  $p$  for object  $x$  are known *in the database of facts*. Actually, the Classic system has a separate operator that “closes a role” after all inferences from previous facts have been made – i.e., after the database might have been augmented by new facts inferred using propagation rules. This kind of closing operation is auto-epistemic as we mentioned. It can be simulated by eliminating the `closed(., .)` term from the database, and requiring the user (or some intermediate system) to assert an upper bound on the number of fillers that is equal to the size of the currently inferred set. If this is done, then the current set of rules works without a hitch. If this is unacceptable, then the only general way we can see to deal with this is to introduce a rule that refers to failure to find any more fillers:

$$\frac{db \vdash x \longrightarrow \text{fills}(p, jj), \text{NOT}(db \vdash x \longrightarrow \text{fills}(p, k)), k \notin \{jj\}}{db \vdash x \longrightarrow \text{atmost}(\text{size}(jj), p)}$$

Such a rule is reminiscent of non-monotonic logic, which is not surprising given the nature of the role-closing operation. If the membership decision itself is not otherwise recursive, then this rule makes the reasoning non-recursively enumerable. But from a practical point of view this is not very important: once the user admits the possibility that the system may not return, then all bets are off. If on the other hand the

KBMS is trying to be effective by having a firm time bound on all computations, then the complement of this family is also of the same complexity, and this is the most likely case for practical implemented systems.