Constraint Programming Letters 1 (2007) 33–88 Submitted 1/2007; Published 11/2007

Data models as constraint systems: a key to the semantic web

Hassan Aït-Kaci HAK@ILOG.COM
ILOG, Inc.
1195 West Fremont Avenue
Sunnyvale, CA 94087, USA

Editor: Lucas Bordeaux, Barry O'Sullivan, and Pascal Van Hentenryck

Abstract

This article illustrates how constraint logic programming can be used to express data models in rule-based languages, including those based on graph pattern-matching or unification to drive rule application. This is motivated by the interest in using constraint-based technology in conjunction with rule-based technology to provide a formally correct and effective—indeed, efficient!—operational base for the semantic web.

Keywords: *semantic web, constraint programming, logic programming, unification, data models, object models, description logic, feature logic, inheritance*

1. Introduction

This article was written upon the invitation by CP 2006's organizers to expand on the contents of my communication as member of CP 2006's panel on *"The next 10 years of constraint programming"* (Aït-Kaci, 2006). Its essential message is that the *semantic web* is a particularly attractive area for applications of constraint-based formalisms. This is true since the latter offer a declarative paradigm for expressing virtually anything that has a formal, especially logical, semantics, including efficient operational semantics. Semantic-web researchers are currently in hot pursuit of a means to integrate "static knowledge" bases (*i.e.*, ontologies) with "dynamic knowledge" bases (*i.e.*, rules). Thus, it is herein argued that constraint logic programming ($\mathcal{CLP}$) is quite suitable a candidate for such an integration. The key is to use constraints to abstract data models upon which rule-based computation may be carried out. Thus, the next 10 years may be the most fructifying yet for constraint and WWW technologies should both communities seize the opportunity to cross-breed as the one offered by the construction of the semantic web. To be sure, this author does not claim lone discovery of this fact. Indeed, several promising directions in this vein are being actively and creatively mined as we speak—so to speak! This is true in particular for web-service discovery—see, *e.g.*, (Benbernou and Hacid, 2005; Preece et al., 2006).

1.1 Motivation

The author recently attended the 5th international semantic web conference (Cruz et al., 2006). It was his *"first"* such conference, his interest having been spurred as a member of the World Wide Web Consortium (W3C) Working Group (WG) on designing a Rule Interchange Format (RIF) as ILOG, Inc.'s representative. In both venues, several proposals have been put forth on the subject of *"integrating rules and ontologies."* The most prominent vein among these proposals centers on the integration of logic programming ($\mathcal{LP}$) style of rules (*e.g.*, Prolog), with the various declensions of one or several of the *"official"* W3C ontology languages; *e.g.*, OWL—Lite, $\mathcal{DL}$, Full—or whatever version of various *description logics* ($\mathcal{DL}$s) and their ancillary XML-based technologies (Motik et al., 2006; Grosof et al., 2003; Krötzsch et al., 2006).

However, it appears that only very few have yet exploited the powerful and flexible computational paradigm known as *constraint logic programming* ($\mathcal{CLP}$), which naturally—and formally!—enables such integrations, both semantically *and* operationally.[1] This is odd since one of the best formulations of this formalism, presented by Höhfeld and Smolka (1988), was originally proposed for the very purpose of integrating $\mathcal{LP}$ with $\mathcal{DL}$s! That the $\mathcal{CLP}$ scheme has not been thus far used as *the* key for achieving this integration is all the more surprising taking into account that the mainstream of work on formal ontologies for the semantic web trace back their origin to the formulation introduced by Schmidt-Schauß and Smolka (1991).

Hence what motivates this author is to explain precisely *why* and *how* the $\mathcal{CLP}$ scheme is adequate for the marriage of rules and ontologies. Concommitantly, the RIF WG requested a similar kind of explication for how constraints may be an appropriate formalism for capturing *"real life"* data models such as those of `Java`, `C#`, or even `C++`. The issues we address in this paper are thus all the more timely for this reason as well.

1.2 Relation to other work

How, then, does our proposal relate to other work? And hasn't constraint technology already been used for the semantic web? We presently review what we know of other efforts to mix rules and ontologies for the semantic web, and how constraint technology has been used.

A means to use description logic as a constraint language in a Horn rule language was in fact worked out before by Bucheit et al. (1993). That work is in fact the theoretical foundation of $\mathcal{AL}$-log (Donini et al., 1998) and CARIN (Levy and Rousset, 1998), and is itself a direct adaptation of the constraint system originally proposed by Schmidt-Schauß and Smolka (1991) to reason about typed attributive concepts. Indeed, it falls within (or very close to) the approach we present here. It is based on seeing $\mathcal{DL}$ statement constraints in the exact same sense as we say. But, although they use a solving process based on formula transformation, it differs from how order-sorted feature ($\mathcal{OSF}$) constraints are solved—see Section 3.1. The latter is based on congruence closure of feature paths (generalizing Herbrand unification) and *reduces* a constraint to solved form or $\perp$. The former is based on a Deductive Tableau method and *completes* a constraint by adding more constraints until it reaches a saturation state, which may then be decided consistent or not. This leads to problematic performance problems, especially for scalability when used on very large ontologies. Furthermore, using such eagerly saturative methods makes it clearly impossible to deal with semi-decidable constraint systems. On the other hand, lazily reductive methods like $\mathcal{OSF}$constraint solving can, by delaying potentially undecidable constraints until further information ensues. These points are further elaborated in Section 4.2. Finally, no formal connection with the $\mathcal{CLP}$ semantic scheme is made by Bucheit et al. (1993). Nevertheless, what they propose is a *bona fide* exemplar of seeing data description as constraints. We will discuss further this approach in Section 3.3.3, in relation with the material on the $\mathcal{OSF}$ and $\mathcal{DL}$ formalisms presented in Sections 3.1 and 3.2.

F-Logic (Kifer et al., 1995) is one popular formalism claimed to be adequate for the reasoning power needed for semantic-web applications (Kifer et al., 2005). It is a formal logic-programming

1. The reader is referred to (Jaffar and Maher, 1994) for an excellent survey of $\mathcal{CLP}$'s power and potential.

paradigm designed to accommodate a "frame" notation extending that of Herbrand terms—so-called *slotted* terms—allowing specifying subterms by keywords rather than position. This notation, used in lieu of arguments of predicates appearing in Horn rules, allows writing rules over attributed objects. There are running $\mathcal{LP}$ systems based on F-Logic: for example, FLORID (Frohn et al., 1997) and Flora-2 (Kifer, September 9, 2007). Although the syntax of F-Logic terms is close to that of $\mathcal{OSF}$ terms, which we present here, they do not have the same semantics at all. For one, F-Logic terms denote fully defined individuals while $\mathcal{OSF}$ terms (like $\mathcal{DL}$ concept expressions), denote sets as well as individuals (singleton sets). This is the difference between a partial description and a complete one: with the former a term is an approximation of individuals (including completes ones), and with the latter terms may only denote complete individuals. Another major difference is that, although F-Logic offers notation for slotted objects, classes, and inheritance, its semantics is not based on $\mathcal{CLP}$, and objects, classes, and inheritance are not processed as constraints. Rather, F-Logic merely offers syntactic sugar that is transformed into a semantically equivalent *tabulated*-logic programming form. The resulting program, when executed, realizes F-Logic's semantics operationally.

Although this approach is a perfectly admissible manner to proceed, it misses the point we advocate here. For one, by relying on the underlying all-purpose $\mathcal{LP}$ reasoning engine misses performance gains made possible by special-purpose solving methods. F-Logic needs to use a tabulated logic programming language such as XSB Prolog (Sagonas et al., 1993) rather than standard Prolog to avoid some termination pitfalls. Indeed, in order to handle recursive class definitions, one needs proof *"memoizing"* such as supported by tabulated-logic programming (Shen et al., 2001). Tabulated-logic programming is a family of Horn-clause resolution-based logic-programming languages—*i.e.*, Prolog—with a modified control strategy that uses proof-memoizing techniques inspired from Dynamic Programming. Control records the most general proofs it has so far undertaken or achieved for any predicate using *tables* (*i.e.*, relational *caches*).[2] Hence, this can avoid falling into fruitless infinite derivations when a proof is found to be a subproof of itself—*e.g.*, such as may be generated by a left-recursive rule. Thus, our essential concern is that F-Logic does not abide by the "data as constraint" slogan we are advocating here.

1.3 Organisation of contents

The remainder of our presentation is organized as follows. The style is a semi-formal tutorial. Its real aim is to stress subtle paradigm shifts that are of primordial importance in appreciating the potential of $\mathcal{CLP}$ as opposed to plain $\mathcal{LP}$ or $\mathcal{CP}$. Thus, Section 2 synopsizes the essence of $\mathcal{CLP}$. We present the basic scheme introduced by Jaffar and Lassez (1987) as reformulated by Höhfeld and Smolka (1988). Section 2.2 deals with how *constraint solving* (as opposed to *general-purpose logical reasoning*) is then put to practical use for meshing various data models in harmony with the logical rule semantics manipulating them. In Section 2.3, we show how the data models of Datalog and Prolog are expressed as constraints fitting the $\mathcal{CLP}$ scheme. In Section 3, we turn to typed attributed structures and express those as constraints. Section 3.1 gives a summary of the $\mathcal{OSF}$ formalism for describing data that takes the form of rooted labelled graphs. Section 3.2 gives a summary of basic Description Logic. Both formalisms are meant to be formal languages

2. One must not confuse *tabulated* LP (Shen et al., 2001) with *deductive tableau* LP (Manna and Waldinger, 1991). A deductive tableau is also a table, but of a different kind whose rows represent assertions and goals, and may be transformed by appropriate deduction rules—non-clausal resolution and induction, essentially.

for describing typed attributed data structures denoting sets. Section 4 compares the expressivity of both and how they are related. In Section 3.3, specific examples of data models are specified as constraints—including `Java`-style classes and objects, but also OWL-type ontologies. Section 4 analyzes the relative expressive and computational powers of the $\mathcal{OSF}$ and $\mathcal{DL}$ formalisms. Last, we conclude in Section 5 with some perspective opened by our proposal for the semantic web to view ontologies as constraints. We also adjoin a small appendix to recall basic terminology on Herbrand terms and substitutions in Section A, on monoidal algebra in Section B, and a technical note on strong extensionality in Section C.

2. Constraint logic programming

In 1987, at the height of research interest in logic programming, Jaffar and Lassez proposed a novel logic-programming *scheme* they called *constraint logic programming* (Jaffar and Lassez, 1987). The idea was to generalize the operational and denotational semantics of $\mathcal{LP}$ by dissociating the relational level—pertaining to resolving definite clauses made up of relational atoms—and the data level pertaining to the nature of the arguments of these relational atoms (*e.g.*, for Prolog, first-order Herbrand terms). Thus, for example, in Prolog seen as a $\mathcal{CLP}$ language, clauses such as:

```
append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
```

are construed as:

```
append(X1,X2,X3) :- true
                  | X1 = [],    X2 = L, X3 = L.
append(X1,X2,X3) :- append(X4,X5,X6)
                  | X1 = [H|T], X2 = L, X3 = [H|R],
                    X4 = T,     X5 = L, X6 = R.
```

The ‘`|`’ may be read as *“such that”* or as *“subject to.”* It is in fact the logical connective *“and”*—*i.e.*, as the one denoted by a *comma* (‘,’). The part of the rule’s RHS on the right of the ‘`|`’ is called its *constraint* part. It keeps together specific parts of the goal formula making the body of the rule—in this case, equations among (first-order) Herbrand terms. The rest of the rule besides the constraint is made up of relational atoms where all variables are distinct. Variables are shared between the relational rule part and the rule constraint.

At first sight, the above reformulation of the `append` predicate may look like a silly and more verbose rewriting of the same thing. And why the ‘`|`’ rather than the ‘`,`’ if they mean the same thing?

It is, indeed, a harmless rewriting of the same thing. But it is not so useless as I shall presently contend. Importantly, it isolates a subset of the factors of the logical *conjunction* that:

1. *commutes* with the other factors in the conjunction; and,

2. may be *solved* using a special-purpose constraint solver, presumably more efficiently than any general-purpose logic rule inference engine.

In addition, as we next explicate, it enables expressing a clean abstract model-theoretic as well as more *operational* proof-theoretic semantics for a large class of rule-based languages over disparate data models—not just Herbrand terms. In particular, it is a natural and effective means for integrating rule-based programming with data-description logics—currently a Holy Grail being actively sought to enable the *semantic web.* At least this is the impression one gets from such recent semantic-web conference papers such as, *e.g.*, (Motik et al., 2006).

2.1 The $\mathcal{CLP}$ scheme

In (Höhfeld and Smolka, 1988), a refinement of the scheme of (Jaffar and Lassez, 1987) is presented that is both more general and simpler in that it abstracts away the syntax of constraint formulae and relaxes some technical demands on the constraint language—in particular, the somewhat baffling *"solution-compactness"* condition required in (Jaffar and Lassez, 1987).[3]

The Höhfeld-Smolka $\mathcal{CLP}$ scheme requires a set $\mathcal{R}$ of *relational symbols* (or, predicate symbols) and a *constraint language* $\mathcal{L}$. It needs very few assumptions about the language $\mathcal{L}$, which must only be characterized by:

- $\mathcal{V}$, a countably infinite set of *variables* (denoted as capitalized $X, Y, \ldots$);
- Φ, a set of *formulae* (denoted $\phi, \phi', \ldots$) called *constraints*;
- a function **VAR**: $\Phi \mapsto \mathcal{V}$, which assigns to every constraint ϕ the set **VAR**(ϕ) of *variables constrained by* ϕ;
- a family of admissible *interpretations* $\mathfrak{A}$ over some domain $D^{\mathfrak{A}}$;
- the set **VAL**$(\mathfrak{A})$ of ($\mathfrak{A}$-)*valuations, i.e.*, total functions, $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$.

By "admissible" interpretation, we mean an algebraic structure and semantic homomorphisms that are appropriate for interpreting the objects in the constraint domains. For example, if the constraint domain is the set of first-order (Herbrand) terms on a ranked signature of uninterpreted function symbols, and the constraints are equations among these—*i.e.*, Prolog—then, any Herbrand interpretation would be an admissible interpretation for this specific constraint language.

Thus, $\mathcal{L}$ is not restricted to any specific syntax, *a priori.* Furthermore, nothing is presumed about any specific method for proving whether a constraint holds in a given interpretation $\mathfrak{A}$ under a given valuation α. Instead, we simply assume given, for each admissible interpretation $\mathfrak{A}$, a function $[\![\_]\!]^{\mathfrak{A}} : \Phi \mapsto \mathbf{2}^{\mathbf{VAL}(\mathfrak{A})}$ that assigns to a constraint $\phi \in \Phi$ the set $[\![\phi]\!]^{\mathfrak{A}}$ of valuations, which we call the *solutions* of ϕ under $\mathfrak{A}$.

Generally, and in our specific case, the constrained variables of a constraint ϕ will correspond to its free variables, and α is a solution of ϕ under the interpretation $\mathfrak{A}$ if and only if ϕ holds true in $\mathfrak{A}$ once its free variables are given values α. As usual, we shall denote this as "$\mathfrak{A}, \alpha \models \phi$."

Then, given $\mathcal{R}$, the set of relational symbols (denoted $r, r_1, \ldots$), and $\mathcal{L}$ as above, the language $\mathcal{R}(\mathcal{L})$ of *relational clauses* extends the constraint language $\mathcal{L}$ as follows. The syntax of $\mathcal{R}(\mathcal{L})$ is defined by:

- the same countably infinite set $\mathcal{V}$ of *variables*;
- the set $\mathcal{R}(\Phi)$ of formulae ϱ from $\mathcal{R}(\mathcal{L})$, which includes:

3. *"Compactness"* in logic is the property stating that if a formula is provable, then it is provable in finitely many steps.

  - all $\mathcal{L}$-constraints, *i.e.*, all formulae ϕ in Φ;
  - all relational atoms $r(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n \in \mathcal{V}$, mutually distinct;

  and is closed under the logical connectives $\&$ (conjunction) and $\rightarrow$ (implication); *i.e.*,

  - $\varrho_1 \ \& \ \varrho_2 \in \mathcal{R}(\Phi)$ if $\varrho_1, \varrho_2 \in \mathcal{R}(\Phi)$;
  - $\varrho_1 \rightarrow \varrho_2 \in \mathcal{R}(\Phi)$ if $\varrho_1, \varrho_2 \in \mathcal{R}(\Phi)$;

- the function **VAR** : $\mathcal{R}(\Phi) \mapsto \mathcal{V}$ extending the one on Φ in order to assign to every formula ϱ the set **VAR**(ϱ) of the *variables constrained by* ϱ:

  - $\mathbf{VAR}(r(X_1, \ldots, X_n)) \overset{\text{DEF}}{=} \{X_1, \ldots, X_n\}$;
  - $\mathbf{VAR}(\varrho_1 \ \& \ \varrho_2) \overset{\text{DEF}}{=} \mathbf{VAR}(\varrho_1) \cup \mathbf{VAR}(\varrho_2)$;
  - $\mathbf{VAR}(\varrho_1 \rightarrow \varrho_2) \overset{\text{DEF}}{=} \mathbf{VAR}(\varrho_1) \cup \mathbf{VAR}(\varrho_2)$;

- the family of admissible *interpretations* $\mathfrak{A}$ over some domain $D^{\mathfrak{A}}$ such that $\mathfrak{A}$ extends an admissible interpretation $\mathfrak{A}_0$ of $\mathcal{L}$, over the domain $D^{\mathfrak{A}} = D^{\mathfrak{A}_0}$ by adding relations $r^{\mathfrak{A}} \subseteq D^{\mathfrak{A}} \times \ldots \times D^{\mathfrak{A}}$ for each $r \in \mathcal{R}$;

- the same set $\mathbf{VAL}(\mathfrak{A})$ of *valuations* $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$.

It is important to note that each variable occurs only once in each atom, and in no other relational atom in a given clause. One may think of this as each relational atom having a unique variable name for each of its arguments. Of course, these variables may (and usually do!) occur in the constraint part; *e.g.*, in the form of argument bindings $X = e$. This requirement of "distinctness" for the variables appearing in relational atoms is simply for each variable to identify uniquely the argument of the atom it stands for, while ensuring that no inconsistency may ever arise with a constraint store. Only the constraint side of a clause may thus be inconsistent, as will be soon explained.

Again, for each interpretation $\mathfrak{A}$ admissible for $\mathcal{R}(\mathcal{L})$, the function $[\![\_]\!]^{\mathfrak{A}} : \mathcal{R}(\Phi) \mapsto \mathbf{2}^{\mathbf{VAL}(\mathfrak{A})}$ assigns to a formula $\varrho \in \mathcal{R}(\Phi)$ the set $[\![\phi]\!]^{\mathfrak{A}}$ of valuations, which we call the *solutions* of ϱ under $\mathfrak{A}$. It is defined to extend the interpretation of constraint formulae in $\Phi \subseteq \mathcal{R}(\Phi)$ inductively as follows:

- $[\![r(X_1, \ldots, X_n)]\!]^{\mathfrak{A}} \overset{\text{DEF}}{=} \{\alpha \mid \langle \alpha(X_1), \ldots, \alpha(X_n) \rangle \in r^{\mathfrak{A}}\}$;
- $[\![\phi_1 \ \& \ \phi_2]\!]^{\mathfrak{A}} \overset{\text{DEF}}{=} [\![\phi_1]\!]^{\mathfrak{A}} \cap [\![\phi_2]\!]^{\mathfrak{A}}$;
- $[\![\phi_1 \rightarrow \phi_2]\!]^{\mathfrak{A}} \overset{\text{DEF}}{=} (\mathbf{VAL}(\mathfrak{A}) - [\![\phi_1]\!]^{\mathfrak{A}}) \cup [\![\phi_2]\!]^{\mathfrak{A}}$.

Note that an $\mathcal{L}$-interpretation $\mathfrak{A}_0$ corresponds to an $\mathcal{R}(\mathcal{L})$-interpretation $\mathfrak{A}$, namely where $r^{\mathfrak{A}_0} = \emptyset$ for every $r \in \mathcal{R}$.

As in Prolog, we shall limit ourselves to *definite relational clauses* in $\mathcal{R}(\mathcal{L})$ that we shall write in the form:

$$r(\vec{X}) \ \leftarrow r_1(\vec{X}_1) \ \& \ \ldots \ \& \ r_m(\vec{X}_m) \ \ [\!] \ \ \phi, \qquad (1)$$

where $(0 \leq m)$ and:

- $r(\vec{X}), r_1(\vec{X}_1), \ldots, r_m(\vec{X}_m)$ are relational atoms in $\mathcal{R}(\mathcal{L})$; and,
- ϕ is a constraint formula in $\mathcal{L}$.

Again, the symbol $\,\|\,$ is just $\,\&\,$ in disguise. It is only used to make the various constituents more conspicuous, separating relational resolvent from the constraint formula ϕ.

Given a set $\mathcal{C}$ of definite $\mathcal{R}(\mathcal{L})$-clauses, a *model* $\mathfrak{M}$ of $\mathcal{C}$ is an $\mathcal{R}(\mathcal{L})$-interpretation such that every valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{M}}$ is a solution of every formula ϱ in $\mathcal{C}$, *i.e.*, $[\![\varrho]\!]^{\mathfrak{M}} = \mathbf{VAL}(\mathfrak{M})$. In fact, any $\mathcal{L}$-interpretation $\mathfrak{A}$ can be extended to a *minimal model* $\mathfrak{M}$ of $\mathcal{C}$. Here, minimality means that the added relational structure extending $\mathfrak{A}$ is minimal in the sense that if $\mathfrak{M}'$ is another model of $\mathcal{C}$, then $r^{\mathfrak{M}} \subseteq r^{\mathfrak{M}'}$ $(\subseteq D^{\mathfrak{A}} \times \ldots \times D^{\mathfrak{A}})$ for all $r \in \mathcal{R}$. For further details, see (Höhfeld and Smolka, 1988).

Also, a least fix-point semantics construction of minimal models of $\mathcal{CLP}$ programs is given in (Höhfeld and Smolka, 1988). The minimal model $\mathfrak{M}$ of $\mathcal{C}$ extending the $\mathcal{L}$-interpretation $\mathfrak{A}$ can be generated as the limit $\mathfrak{M} = \bigcup_{i\geq 0} \mathfrak{A}_i$ of a sequence of $\mathcal{R}(\mathcal{L})$-interpretations $\mathfrak{A}_i$ as follows. For all $r \in \mathcal{R}$ we define:

$$\left.\begin{array}{lcl} r^{\mathfrak{A}_0} & \stackrel{\text{DEF}}{=} & \emptyset; \\ r^{\mathfrak{A}_{i+1}} & \stackrel{\text{DEF}}{=} & \{\langle \alpha(x_1), \ldots, \alpha(x_n)\rangle \mid \alpha \in [\![\varrho]\!]^{\mathfrak{A}_i}\ ;\ r(x_1, \ldots, x_n) \leftarrow \varrho \in \mathcal{C}\}; \\ r^{\mathfrak{M}} & \stackrel{\text{DEF}}{=} & \bigcup_{i\geq 0} r_i^{\mathfrak{A}}. \end{array}\right\} \tag{2}$$

A *resolvent* is a formula of the form $\varrho \;\|\; \phi$, where ϱ is a possibly empty conjunction of relational atoms $r(X_1, \ldots, X_n)$—its *relational part*—and ϕ is a possibly empty conjunction of $\mathcal{L}$-constraints—its *constraint part*. Again, $\,\|\,$ is just $\,\&\,$ in disguise and is used only to emphasize which part is which. (As usual, an empty conjunction is assimilated to *true*, the formula that takes all arbitrary valuations as solution.)

Finally, the Höhfeld-Smolka scheme defines constrained *resolution* as a reduction rule on resolvents that gives a sound and complete interpreter for *programs* consisting of a set $\mathcal{C}$ of definite $\mathcal{R}(\mathcal{L})$-clauses. The reduction of a *resolvent* R of the form:

$$B_1 \;\&\; \ldots \;\&\; r(X_1, \ldots, X_n) \;\&\; \ldots B_k \;\|\; \phi \tag{3}$$

by the (renamed) program clause:

$$r(X_1, \ldots, X_n) \leftarrow A_1 \;\&\; \ldots \;\&\; A_m \;\|\; \phi' \tag{4}$$

is the new resolvent R' of the form:

$$B_1 \;\&\; \ldots \;\&\; A_1 \;\&\; \ldots \;\&\; A_m \;\&\; \ldots B_k \;\|\; \phi \;\&\; \phi'. \tag{5}$$

The soundness of this rule is clear: under every interpretation $\mathfrak{A}$ and every valuation such that R holds, then so does R', *i.e.*, $[\![R']\!]^{\mathfrak{A}} \subseteq [\![R]\!]^{\mathfrak{A}}$. It is also not difficult to prove its completeness: if $\mathfrak{M}$ is a minimal model of $\mathcal{C}$, and $\alpha \in [\![R]\!]^{\mathfrak{M}}$ is a solution of the formula R in $\mathfrak{M}$, then there exists a sequence of reductions of (the $\mathcal{R}(\mathcal{L})$-formula) R to an $\mathcal{L}$-constraint ϕ such that $\alpha \in [\![\phi]\!]^{\mathfrak{M}}$.

Before we give our formal view of constraint solving as a proof system, let us recapitulate a few important points:

- Although semantically discriminating some specific formulae as constraints, the $\mathcal{CLP}$ view agrees, and indeed uses, the interpretation of constraints as formulae, thus inheriting "for free" a crisp model-theory as shown above.

- Better yet, the most substantial benefit is obtained operationally. Indeed, this is so because we can identify among all formulae to be proven some specific formulae to be processed as constraints, for which presumably a specific-solving algorithm may be used rather than a general-purpose logic-programming machinery.

The above remarks are perhaps the most important idea regarding the $\mathcal{CLP}$ approach. Indeed, many miss this point: *"Constraints are logical formulae—so why not use only logic?"* Sure, constraints are logical formulae—and that is *good!* But the fact that such formulae, appearing as factors in a conjunction, *commute* with the other non-constraint factors enables freedom for the operational scheduling of resolvents to be reduced. This is the key situation being exploited in our approach. Yet another serendipitous benefit of this state of affairs is that it enables more declarative operational semantics than otherwise possible thanks to the technique of goal *residuation* (Aït-Kaci et al., 1987; Aït-Kaci and Nasr, 1989; Smolka, 1993; Aït-Kaci et al., 1994b). As well, as explained in (Aït-Kaci et al., 1997), an important effect of constraint solving is that it enables a simple means to *remember proven fact* (*i.e.,* proof memoizing)—something that model-theory is patently not concerned with. (See also Sections 1.2 and 3.3.4.)

Thanks to the separation of concerns explicated above between rules and constraints, we may use constraint solving operationally as *realizing* the logical semantics of *constraints as logical formulae* using special-purpose algorithms using a proof-theoretic notion of constraint *normalization.* We explain this next.

2.2 Constraint solving

In the Höhfeld-Smolka $\mathcal{CLP}$ scheme, the language of constraint Φ is not syntactically specified in any way, except that it makes use of the same set of variables as the relational rule part. Special valuations $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$ of variables taking values in an appropiate semantic domain of interpretation are deemed *solutions* in the sense that they *satisfy* all constraints as mandated by the $\mathcal{CLP}$ scheme. How to find these solutions operationally is orthogonal to the $\mathcal{CLP}$ model-theoretic semantics. A specific operational process computing constraint solutions is called *constraint solving.* It may be specified in any operational way as long as it may be formally proven to be correct with respect to the logical semantics of the constraints.

§**Decision problems –** There are two decision problems of interest regarding constraints in a constraint language Φ: (1) *consistency* and (2) *entailment.*

The constraint $\perp$, called the *inconsistent constraint,* is such that $\mathfrak{A}, \alpha \not\models \perp$ for every interpretation $\mathfrak{A}$ and $\mathfrak{A}$-valuation α. Two syntactic expressions e and e' are said to be *syntactically congruent*—noted $e \simeq e'$—if and only if they denote the same semantic object; *viz.,* $[\![e]\!] = [\![e']\!]$.

Definition 1 (Constraint consistency) *A constraint ϕ is said to be consistent if and only if $\phi \not\simeq \perp$.*

Thus, when data structures t and t' are viewed as constraints, we say that they are *unifiable* if and only if the constraint $t = t'$ is consistent. When t and t' are unifiable, *unification* of t and t' is the operation that computes a valuation α such that $\alpha(t)$ and $\alpha(t')$ are identical data structure. Clearly then, unifiability is a symmetric relation and unification is a commutative operation.

Definition 2 (Constraint entailment) *Given two constraints ϕ and ϕ', ϕ is said to entail ϕ' if and only if $\mathfrak{A}, \alpha \not\models \phi$ or $\mathfrak{A}, \alpha \models \phi'$ for every interpretation $\mathfrak{A}$ and $\mathfrak{A}$-valuation α.*

Given two data structures t and t', we say that t *subsumes* (or *is more general than*) t' if and only if t' entails t when viewed as constraints. When t subsumes t', *pattern-matching* is the operation computing a valuation α such that $\alpha(t)$ and t' are identical data structure. We then say that t *matches* t'. Clearly then, entailment is an asymmetric relation and pattern-matching is a non-commutative operation.

Typically, unification is used in rule-based computational systems such as $\mathcal{LP}$ and equational theorem-proving, while pattern-matching is used in rule-based systems using rewrite rules or production rules. The former allows refining input data to accommodate success, while the latter forbids modifying input data. Finally, note that pattern-matching can itself be reduced to a unification problem by treating all the variables of the entailing data structure as constants. This is akin to stating that constraint ϕ entails constraint ϕ' if and only if $\phi \ \& \ \phi' \simeq \phi$.

Therefore, when data structures are viewed as constraints describing data, the only decision procedure that is needed for constraint solving is consistency checking.

§**Constraint normalization –** Because constraints are logical formulae, constraint solving may be done by syntax-transformation rules in the manner advocated by Plotkin (1981). Such a syntax-transformation process is called *constraint normalization.* It is convenient to specify it as a set of semantics-preserving syntax-driven conditional rewrite rules called *constraint normalization rules.* We shall write such rules in "fraction" form such as:

$$(\mathcal{A}_n) \quad \underline{\textbf{RULE NAME}} : \qquad \left[\texttt{Condition}\right] \quad \frac{\texttt{Prior Form}}{\texttt{Posterior Form}}$$

where $\mathcal{A}_n$ is a label identifying the rule: $\mathcal{A}$ is the rule's constraint system's name, and n is a number, or a symbol, uniquely identifiying the rule within its system. Such a rule specifies how the prior formula may be transformed into the posterior formula, modulo syntactic congruences. `Condition` is an optional side metacondition on the formulae involved in the rules. When a side condition is specified, the rule is applied only if this condition holds. A missing condition is implicitly $\mathfrak{true}$. A normalization a rule is said to be *correct* if and only if the denotation of the prior is the same as that of the posterior whenever the side condition holds.

§**Normal form –** A constraint formula that cannot be further transformed by any normalization rule is said to be in *normal form.* Thus, given a syntax of constraint formulae, and a set of correct constraint-normalization rules, constraint normalization operates by successively applying any applicable rule to a constraint formula, only by syntax transformation.

§**Solved form –** Solved forms are particular normal forms that can be immediately seen to be consistent or not. Indeed, normal forms constitute a canonical representation for constraints. Of course, for constraint normalization to be effective for the purpose of constraint solving, a rule must somehow produce a posterior form whose satisfiability is simpler to decide than that of its prior form. Indeed, the point is to converge eventually to a constraint form that can be trivially decided consistent, or not, based on specific syntactic criteria.

§**Residuated form –** Constraints that are in normal form but not in solved form are called *residuated* constraints. Such a constraint is one that cannot be normalized any further, but that may not be decided either consistent or inconsistent in its current form. Thanks to the commutativity of conjunction, residuated forms may be construed as *suspended* computation. Indeed, because constraint normalization preserves the logical semantics of constraints, the process *commutes* with the relational resolution as expressed by the $\mathcal{CLP}$ resolution operation that yields a new constrained resolvent (5) from an old constrained resolvent (3) and a constrained clause (4). This interplay establishes "for free" an implicit *coroutining* between the resolution and constraint-solving processes as these processes communicate through their shared logical variables.

We next specify some common logic-programming rule dialect classes using the $\mathcal{CLP}$ scheme by explicating the kind of constraint formulae they are manipulating, with what normalization rules, and towards what solved forms.

2.3 Examples

To illustrate the foregoing scheme, we now recast the two well-known logic programming languages Datalog and Pure Prolog in terms of $\mathcal{CLP}$ by explicating their constraint systems.

2.3.1 DATALOG

Datalog is a simplified logic-programming dialect sufficient for expressing relational data, views, and queries, as well as recursion. It is a formal tool used by academics for expressing computation in Deductive Databases (Ullman, 2003).

A Datalog program consists of two parts: an *intensional* database (IDB) and an *extensional* database (EDB). The IDB is an unordered collection rules of the form:

$$r_0(d_1^0, \ldots, d_{n_0}^0) \;\leftarrow\; \bigwedge_{i\geq 1} a_i(d_1^i, \ldots, d_{n_i}^i).$$

where the r_i's are relational symbols, the a_i's are possibly negated relational symbols (*i.e.*, either r or $\neg r$), and the $d_{n_i}^i$'s are either logical variables or constants. The EDB is an unordered collection of relational tuples of the form:

$$r(c_1, \ldots, c_n).$$

where r is a relational symbol that does not appear as the head of an IDB rule, and the c_i's are constants. When no negation is allowed in the rules, the dialect is called *Positive Datalog.* When negation is restricted so that no rule head's $r(\ldots)$ may lead to an atom $\neg r(\ldots)$ through any recursive dependency, the dialect is called *Stratified Datalog.*

It is not difficult to show that the least fix-point model of Positive Datalog concides with that defined by Equations (2), where $\mathcal{D}$ is the constraint system that solves equations between variables and values appearing as arguments of tuples in the EDB. This constraint language consists of conjunctions of equations of the form $s \doteq t$ where s and t are either variables or constants.

The solved forms are conjunctions of equations either of the form $X \doteq a$, where X is a variable and a a constant, or $X \doteq Y$ where Y appears nowhere else. Constraint normalization rules are very simple: given a conjunction ϕ of such equations, we apply non-deterministically any of the rules of Figure 1 until none is applicable. The expression $\phi[X/Y]$ denotes the constraint ϕ where all occurrences of Y are replaced by X. These rules are *confluent* modulo variable renaming. Confluent rules are such that order of application does not matter—the rules have the "Church-Rosser" property. Recall that constraint normalization rules are always implicitly applied modulo syntactic congruence—*viz.*, here: associativity and commutativity of the $\&$ operator. Clearly, they also always terminate, ending up either in $\bot$, the inconsistent constraint, or in a conjunction of equations in solved form.

$(\mathcal{D}_1)$ **<u>ERASE</u>**:
[**if** t is a constant or a variable]

$$\frac{\phi \;\&\; t \doteq t}{\phi}$$

$(\mathcal{D}_2)$ **<u>FLIP</u>**:
[**if** a is a constant **and** X is a variable]

$$\frac{\phi \;\&\; a \doteq X}{\phi \;\&\; X \doteq a}$$

$(\mathcal{D}_3)$ **<u>SUBSTITUTE</u>**:
[**if** X and Y are variables **and** Y occurs in ϕ]

$$\frac{\phi \;\&\; X \doteq Y}{\phi[X/Y] \;\&\; X \doteq Y}$$

$(\mathcal{D}_4)$ **<u>FAIL</u>**:
[**if** a and b are constants **and** $a \neq b$]

$$\frac{\phi \;\&\; a \doteq b}{\bot}$$

Figure 1: **The constraint system $\mathcal{D}$**

This constraint-normalization process merely amounts to verifying constant arguments and binding variable arguments. Hence, a solved form is nothing other than a *binding environment* corresponding to a tuple belonging to the model of the computed relation—*i.e.*, what we called a variable valuation (α) in Equations (2). With this setup, Datalog $\in \mathcal{CLP}(\mathcal{D})$, where $\mathcal{D}$ is the constraint system of Figure 1. We informally use the notation $\mathcal{CLP}(\mathcal{A})$ to characterize a $\mathcal{CLP}$ language over a constraint system $\mathcal{A}$.

2.3.2 PURE PROLOG

In this section, we describe a non-deterministic unification algorithm presented as a set of constraint-normalization rules. Each normalization rule is *correct*; *i.e.*, it is a syntactic transformation of a set of equations that preserves all and only solutions of the original constraint. (See Appendix Section A for basic notions for first-order Herbrand terms and substitutions.) This is in contrast with Robinson's unification algorithm, which is (still!) often presented as an atomic operation on terms (Robinson, 1965). These normalization rules were first formulated by Jacques Herbrand in 1930 in his PhD thesis—reprinted in (Herbrand, 1971), Page 148—that is, 35 years before Robinson's algorithm was published! This was already explicitly pointed out in 1976 by Gérard Huet in his French *thèse d'état* (Huet, 1976). These rules were later rediscovered by Martelli and Montanari (1982) —20 years after Robinson's paper! They were seeking to simplify [!] Robinsons's algorithm, apparently unaware of Huet's remark. As we shall see later in this document—see Sections 3.1.4 and 3.3.1—this algorithm is a special case of a more general one based on $\mathcal{OSF}$ constraint-solving by normalization. For related readings giving a a generalized abstract view of unification and constraint-solving in a category-theoretic setting, see also (Schmidt-Schauß and Siekmann, 1988) and (Goguen, 1989).

§**Herbrand term unification –** An *equation* is a pair of terms, written $s \doteq t$. A substitution σ is a *solution* (or a *unifier*) of a set of equations $\{s_i \doteq t_i\}_{i=1}^n$ iff $s_i\sigma = t_i\sigma$ for all $i = 1, \ldots, n$. Two sets of equations are *equivalent* iff they both admit *all* and *only* the same solutions. Following (Martelli and Montanari, 1982), we define two transformations on sets of equations—*term decomposition* and *variable elimination*. They both preserve solutions of sets of equations.

> **TERM DECOMPOSITION**: If a set E of equations contains an equation $f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)$, where $f \in \Sigma_n, (n \geq 0)$, then the set $E' = E - \{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\} \cup \{s_i \doteq t_i\}_{i=1}^n$ is equivalent to E. If $n = 0$, the equation is simply deleted.
>
> **VARIABLE ELIMINATION**: If a set E of equations contains an equation $X \doteq t$ where $t \neq X$, then the set $E' = (E - \{X \doteq t\})\sigma \cup \{X \doteq t\}$ where $\sigma = \{t/X\}$, is equivalent to E.

A set of equations E is partitioned into two subsets: its *solved* part and its *unsolved* part. The solved part is its maximal subset of equations of the form $X \doteq t$ such that X occurs nowhere in the full set of equations except as the left-hand side of this equation alone. The unsolved part is the complement of the solved part. A set of equations is said to be *fully solved* iff its unsolved part is empty.

In Figure 2 is a unification algorithm. It is a non-deterministic normalization procedure for a constraint $\phi = \varepsilon_1 \ \& \ \ldots \ \& \ \varepsilon_n$ corresponding to a set $E = \{\varepsilon_1, \ldots, \varepsilon_n\}$ of equations. The *"Cycle"* rule performs the so-called *"occurs-check"* test. Omitting this rule altogether yields rational term unification; *i.e.*, cyclic equations may be obtained as solved forms. Most implemented systems omit occurs-check either for reason of efficiency (*e.g.*, most Prolog compilers) or simply because their data model's semantics has *bona fide* interpretations for cyclic terms—*e.g.*, (Colmerauer, 1990; Aït-Kaci and Podelski, 1993). For a thorough understanding of the logic of finite and infinite rational tree constraints, one must read Maher (1988a,b). For linguistics applications based on a formalism mixing categorial grammars and feature terms, see Damas et al. (1994).

If this non-deterministic equation-normalization process terminates with success, the set of equations that emerges as the outcome is fully solved. Its solved part defines a substitution called the *most general unifier* (MGU) of all the terms participating as sides of equations in E. If it terminates with failure, the set of equations E is unsatisfiable and no unifier for it exists. Thus, Prolog $\in \mathcal{CLP}(\mathcal{H})$, where $\mathcal{H}$ is the constraint system of Figure 2.

Of course, the benefit of using $\mathcal{CLP}$ to reformulate Prolog and Datalog is only an academic exercise confirming that it is at least capable of that much expressive power. Going beyond conventional logic-programming languages' expressivity, the exact same manner of proceeding can be (and has been) used for logic-programming reasoning over more interesting data models. Examples are $\mathcal{H}^\lambda$ integrating Herbrand terms with interpreted functions—*i.e.*, the λ-Calculus—as done by Aït-Kaci and Nasr (1989), or using guarded rules as done by Smolka (1993), or using rewrite rules over typed objects as done by Aït-Kaci and Podelski (1994).

As mentioned before, we will reformulate Herbrand unification in the more general framework of $\mathcal{OSF}$ constraints, as $\mathcal{OSF}$ constraint normalization. The $\mathcal{OSF}$ approach is more general than Jacques Herbrand's algorithm in the sense that it works not only for Herbrand terms, but also for order-sorted labelled graph structures using an $\mathcal{OSF}$ constraint syntax that amounts to conjunctions of finer-grained atomic constraints. Operationally, this allows more commutation with inference operations such as, *e.g.*, logical resolution, and therefore the more declarative non-deterministic concurrent entertwining of both processes. Indeed, when a constraint system is only

$(\mathcal{H}_1)$ **ERASE**:

$\left[\ \textbf{if}\quad t \in \Sigma_0 \cup \mathcal{V}\ \right]$

$$\frac{\phi\ \&\ t \doteq t}{\phi}$$

$(\mathcal{H}_2)$ **FLIP**:

$\left[\begin{array}{rl} \textbf{if} & t \text{ is not a variable} \\ \textbf{and} & X \text{ is a variable} \end{array}\right]$

$$\frac{\phi\ \&\ t \doteq X}{\phi\ \&\ X \doteq t}$$

$(\mathcal{H}_3)$ **SUBSTITUTE**:

$\left[\ \textbf{if}\quad X \text{ occurs in } \phi\ \right]$

$$\frac{\phi\ \&\ X \doteq t}{\phi[X/t]\ \&\ X \doteq t}$$

$(\mathcal{H}_4)$ **DECOMPOSE**:

$\left[\ \textbf{if}\quad f \in \Sigma_n, \quad (n \geq 0)\ \right]$

$$\frac{\phi\ \&\ f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)}{\phi\ \&\ s_1 \doteq t_1\ \&\ \ldots\ \&\ s_n \doteq t_n}$$

$(\mathcal{H}_5)$ **FAIL**:

$\left[\begin{array}{rll} \textbf{if} & f \in \Sigma_m, & (m \geq 0) \\ \textbf{and} & g \in \Sigma_n, & (n \geq 0) \\ \textbf{and} & m \neq n & \end{array}\right]$

$$\frac{\phi\ \&\ f(s_1, \ldots, s_m) \doteq g(t_1, \ldots, t_n)}{\perp}$$

$(\mathcal{H}_6)$ **CYCLE**:

$\left[\begin{array}{rl} \textbf{if} & X \text{ is a variable} \\ \textbf{and} & t \text{ is not a variable} \\ \textbf{and} & X \text{ occurs in } t \end{array}\right]$

$$\frac{\phi\ \&\ X \doteq t}{\perp}$$

Figure 2: **The constraint system $\mathcal{H}$**

semi-decidable—*e.g.*, higher-order unification (Huet, 1972)—complete rule resolution over such constraints is possible by dove-tailing resolution steps and constraint-solving steps—*e.g.*, λProlog (Nadathur and Miller, 1998).

In the next section, we develop a finer grain notion of term—whether tree, DAG, or graph, node- and/or edge-labelled, with or without arity or schema constraints—to formalize more adequately modern data models such as, *e.g.*, objects and their class types, inheritance, *etc.*, ... Such terms are defined as specific *"crystallized"* syntaxes that *"dissolve"* into a semantically equivalent conjunction of elementary constraints. The chemical metaphor of a molecular structure dissolving into a free solution of ions is quite appropriate here. The term syntax structure is the *"molecule,"* and the *"ions"* are the elementary constraints floating freely in the *"aqueous solution"*—*i.e.*, the conjunction. Thus, the *"ions"*—*i.e.*, the elementary constraints— are allowed to *"react"*—*i.e.*, be normalized—as they *"move about"* thanks to $\&$ being associative and commutative. The "empty" *"aqueous solution"* is the constraint $\mathfrak{true}$. The constraint-*solving* process thus starts with a constraint-*dissolving* process. This chemical metaphor is not new and was originally proposed in (Banâtre and Le Métayer, 1986), and later used to define the *Chemical Abstract Machine*, the calculus of concurrency of (Berry and Boudol, 1990). Although, the chemical metaphor is not made explicit in concurrency models based on constraints—*e.g.*, (Saraswat, 1989)—it works for constraint-based models of concurrency as well as for *higher-order* concurrency models. Concurrent languages such as Gamma (Le Métayer, 1994) and Oz (Smolka, 1994) are based on this elegant metaphor.

3. Typed attributed stuctures as constraints

Many modern computation systems are based on a notion of object and class. An object is a record structure—*i.e.*, a composite structure consisting of a conjunction of *fields* holding *values*. A class is a type of objects—*i.e.*, a composite structure consisting of a conjunction of fields holding *types*. A class describes a template for all objects of its type. Object to class adequacy is ensured by type verification. Such type verification may be done partly statically, or dynamically. It may consist of *type checking*—*i.e.*, confirming that all object fields carry only values as prescribed by the type of this field in the object's class—or *type inference*—*i.e.*, deducing appropriate most general types wherever type information is missing or incomplete—or both. Static type checking may be seen as *abstract interpretation*—*i.e.*, a decidable approximation of the dynamic model of computation. Typically, appropriately called *dependent types*—*i.e.*, any type depending on dynamic values—are checked dynamically—*e.g.*, array bounds in `Java`. When types are viewed as constraints, dynamic type checking based on constraint-solving in a logical rule language may also be used as a performance booster as it focuses the inference process only on relevant values. In addition, type constraints are incrementally memoized as they are verified, therefore acting as *proof caches*. As a result, nothing about a type should ever be proved twice.

This relation of object/class type adequacy can be captured precisely and formally as a constraint system when the classes and objects *themselves* are seen no longer as *labelled graph structures* but as *logical constraints*. This is the purpose of the order-sorted feature constraint system we summarize next, after we review some basic vocabulary.

§**Attributive conceptual taxonomies –** In the literature, the following words are often used interchangeably for the same category of symbols: *attribute*, *projection*, *role*, *field*, *slot*, *property*, *feature*. For us as well: any such symbol will denote a function—even *role*, which denotes a binary relation (*i.e.*, a set-valued function). So, without loss of generality, we shall call such symbols *features*.

The following words are also often used interchangeably to mean roughly the same thing: *type*, *class*, *sort*, *kind*, *domain*, *extension*. However, such is not the case in this presentation! Although they all denote sets of values, there are important distinctions; *viz.*, we use:

- *"type"*—for conventional programming *data* types; *viz.*., types such as those used in most popular programming languages such as `Java`, *C#*, or `C`/`C++`, *etc.*., . . .
- *"class"*—for types of objects,
- *"sort"*—for mathematical set-denoting symbols,
- *"kind"*—for types of types (as used in Type Theory),
- *"domain"*—for finite-domain or interval constraints,
- *"extension"*—for the set of values populating a type.

In the AI literature, some also use the term *"concept"* to denote a set—*i.e.*, a monadic relation. We will too when we deal with Description Logic expressions as constraints, to emphasize the connection.

3.1 Order-sorted feature constraints

We recall briefly here the essentials of a constraint formalism for order-sorted featured ($\mathcal{OSF}$) objects and classes.

In (Aït-Kaci and Nasr, 1986), ψ-terms were proposed as flexible record structures for logic programming. Indeed, we shall see that ψ-terms are a generalization of first-order terms. However, ψ-terms are of wider interest. Since first-order terms are the pervasive data structures used by symbolic programming languages, whether based on predicate or equational logic, the more flexible ψ-terms offer an interesting alternative as a formal data model for expressing computation over typed attributed objects using pattern-directed rules.

The easiest way to describe a ψ-term is with an example. Here is a ψ-term that may be used to denote a generic `person` object structure:

$$\begin{aligned} P : \texttt{person}(\texttt{name} \Rightarrow\ & \texttt{id}(\texttt{first} \Rightarrow \texttt{string}, \\ & \qquad\quad \texttt{last} \Rightarrow S : \texttt{string}), \\ \texttt{age} \Rightarrow\ & 30, \\ \texttt{spouse} \Rightarrow\ & \texttt{person}(\texttt{name} \Rightarrow \texttt{id}(\texttt{last} \Rightarrow S), \\ & \qquad\qquad \texttt{spouse} \Rightarrow P)). \end{aligned} \tag{6}$$

Namely, a 30 year-old person who has a name in which the first and last parts are strings, and whose spouse is a person sharing his or her last name, that latter person's spouse being the first person in question.

This expression looks like a record structure. Like a typical record, it has field names; *i.e.*, the symbols on the left of $\Rightarrow$. We call these *feature* symbols. In contrast with conventional records, however, ψ-terms can carry more information. Namely, the fields are attached to *sort* symbols (*e.g.*, `person`, `id`, `string`, 30, *etc.*). These sorts may indifferently denote individual values (*e.g.*, 30) or sets of values (*e.g.*, `person`, `string`). In fact, values are assimilated to singleton-denoting sorts. Sorts are partially ordered so as to reflect set inclusion; *e.g.*, `employee` $<$ `person` means that all `employee`s are `person`s. Finally, sharing of structure can be expressed with *variables* (*e.g.*, P and S). This sharing may be circular (*e.g.*, P).

In what follows, we see how these terms may be interpreted as logical constraints called $\mathcal{OSF}$ constraints. More precisely, ψ-terms correspond to $\mathcal{OSF}$ constraints in solved form. Next, we define a simple constraint formalism for expressing, and reasoning with, sorted attributed structures. The reader may wish to consult Appendix Section B for needed formal notions.

3.1.1 $\mathcal{OSF}$ ALGEBRAS

An $\mathcal{OSF}$ *Signature* is given by $\langle \mathcal{S}, \leq, \wedge, \mathcal{F} \rangle$ such that:

- $\mathcal{S}$ is a set of *sorts* containing the sorts $\top$ and $\bot$;
- $\leq$ is a decidable partial order on $\mathcal{S}$ such that $\bot$ is the least and $\top$ is the greatest element;
- $\langle \mathcal{S}, \leq, \wedge \rangle$ is a lower semilattice ($s \wedge s'$ is called the greatest common subsort of s and s');
- $\mathcal{F}$ is a set of *feature symbols*.

Referring to the ψ-term example (6), the set of sorts $\mathcal{S}$ contains set-denoting symbols such as `person`, `id`, and `string`. The set of features $\mathcal{F}$ contains function-denoting symbols—symbols on the left of $\Rightarrow$—such as `name`, `name`, `first`, `last`, `spouse`, *etc.*, ... The ordering on the sorts $\mathcal{S}$ denotes set inclusion and the infimum operation $\wedge$ denotes set intersection. Therefore, $\top$ denotes the all-inclusive sort (the set of all things), and $\bot$ denotes the all-exclusive sort (the set of no things). This is formalized next.

Given an $\mathcal{OSF}$ signature $\langle \mathcal{S}, \leq, \wedge, \mathcal{F} \rangle$, an $\mathcal{OSF}$ *algebra* is a structure:

$$\mathfrak{A} = \langle D^{\mathfrak{A}}, (s^{\mathfrak{A}})_{s \in \mathcal{S}}, (f^{\mathfrak{A}})_{f \in \mathcal{F}} \rangle$$

such that:

- $D^{\mathfrak{A}}$ is a non-empty set, called the *domain* of $\mathfrak{A}$;
- for each sort symbol s in $\mathcal{S}$, $s^{\mathfrak{A}}$ is a subset of the domain; in particular, $\top^{\mathfrak{A}} = D^{\mathfrak{A}}$ and $\bot^{\mathfrak{A}} = \emptyset$;
- $(s \wedge s')^{\mathfrak{A}} = s^{\mathfrak{A}} \cap s'^{\mathfrak{A}}$ for two sorts s and s' in $\mathcal{S}$;
- for each feature f in $\mathcal{F}$, $f^{\mathfrak{A}}$ is a total unary function from the domain into the domain; *i.e.*, $f^{\mathfrak{A}} : D^{\mathfrak{A}} \mapsto D^{\mathfrak{A}}$.

The essence of meaning-preserving mappings between $\mathcal{OSF}$ algebras is that they should respect feature application and sort inclusion. Thus, an $\mathcal{OSF}$ *homomorphism* $\gamma : \mathfrak{A} \mapsto \mathfrak{B}$ between two $\mathcal{OSF}$ algebras $\mathfrak{A}$ and $\mathfrak{B}$ is a function $\gamma : D^{\mathfrak{A}} \mapsto D^{\mathfrak{B}}$ such that:

- $\gamma(f^{\mathfrak{A}}(d)) = f^{\mathfrak{B}}(\gamma(d))$ for all $d \in D^{\mathfrak{A}}$;
- $\gamma(s^{\mathfrak{A}}) \subseteq s^{\mathfrak{B}}$.

The notion of interest for inheritance is that of $\mathcal{OSF}$ *endomorphism*. That is, when an $\mathcal{OSF}$ homomorphism γ is internal to an $\mathcal{OSF}$ algebra (*i.e.*, $\mathfrak{A} = \mathfrak{B}$), it is called an $\mathcal{OSF}$ endomorphism of $\mathfrak{A}$. This means:

- $\forall f \in \mathcal{F}, \forall d \in D^{\mathfrak{A}},\;\; \gamma(f^{\mathfrak{A}}(d)) = f^{\mathfrak{A}}(\gamma(d))$

- $\forall s \in \mathcal{S},\ \ \gamma(s^{\mathfrak{A}})\ \subseteq s^{\mathfrak{A}}$

As pictured in Figure 3, this definition captures formally and precisely *inheritance of attributes* as used, *e.g.*, in object-oriented classes, semantic networks, and formal ontological logics defining *concept hierarchies*. Namely, a concept C_1 (the subconcept) inherits from a concept C_2 (its

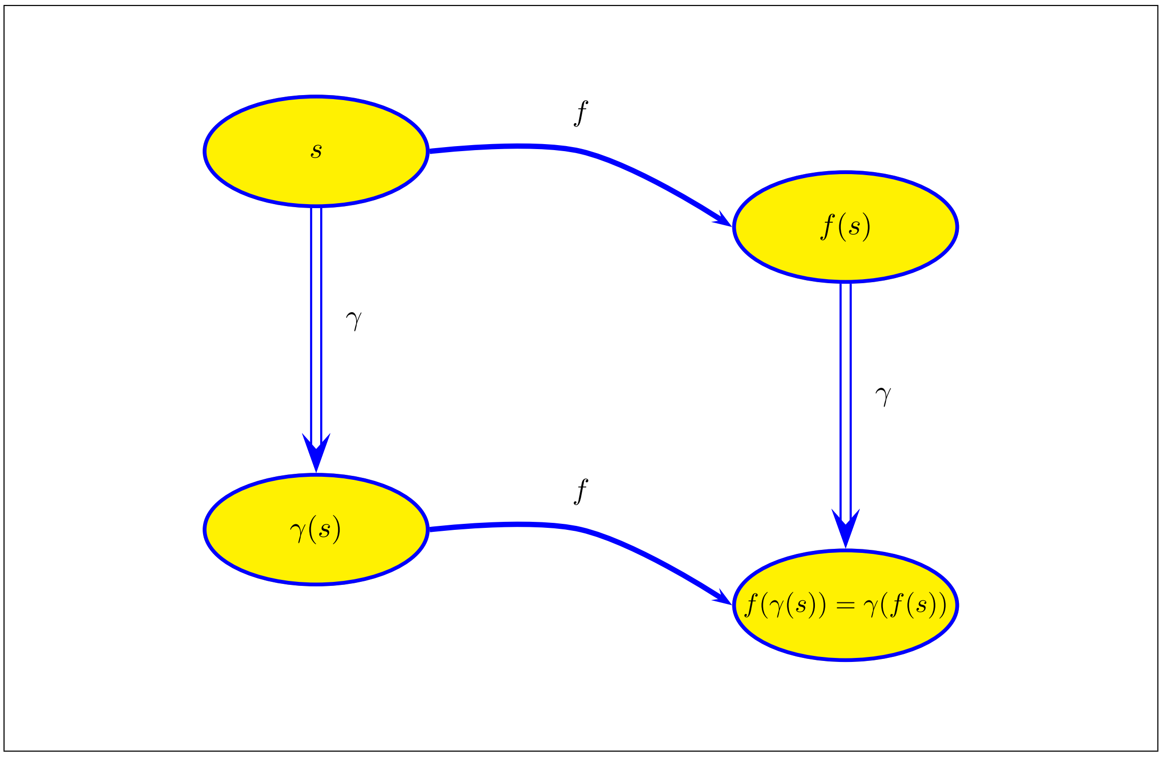

Figure 3: **Property inheritance as $\mathcal{OSF}$ endomorphism**

superconcept) *if and only if* there exists an $\mathcal{OSF}$ endormorphism taking the set denoted by the superconcept C_2 to the set denoted by the subconcept C_1.

3.1.2 $\mathcal{OSF}$ TERMS

An *$\mathcal{OSF}$ term* t is an expression of the form: $X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n)$ where X is a variable in $\mathcal{V}$, s is a sort in $\mathcal{S}$, $f_1, \ldots, f_n$ are features in $\mathcal{F}$, $n \geq 0$, $t_1, \ldots, t_n$ are $\mathcal{OSF}$ terms, and where $\mathcal{V}$ is a countably infinite set of variables.

Given a term $t = X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n)$, the variable X is called its *root* variable and sometimes referred to as $\textbf{ROOT}(t)$. The set of all variables occurring in t is defined as $\textbf{VAR}(t) = \{\textbf{ROOT}(t)\} \cup \bigcup_{i=1}^{n} \textbf{VAR}(t_i)$.

Given a term t as above, an $\mathcal{OSF}$ interpretation $\mathfrak{A}$, and an $\mathfrak{A}$-valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$, the *denotation* of t is given by:

$$[\![t]\!]^{\mathfrak{A},\alpha} \ \stackrel{\text{DEF}}{=\!=} \ \{\alpha(X)\} \cap s^{\mathfrak{A}} \cap \bigcap_{1 \leq i \leq n} (f_i^{\mathfrak{A}})^{-1}([\![t_i]\!]^{\mathfrak{A},\alpha}). \tag{7}$$

Hence, for a fixed $\mathfrak{A}$-valuation α, $[\![t]\!]^{\mathfrak{A},\alpha}$ is either the empty set or the singleton set $\{\alpha(\textbf{ROOT}(t))\}$. In fact, it is *not* the empty set if and only if the value $\alpha(\textbf{ROOT}(t))$ lies in the denotation of the sort s, as well as each and every inverse image by the denotation of feature f_i of the denotation of the corresponding subterm $[\![t_i]\!]^{\mathfrak{A},\alpha}$ *under the same* $\mathfrak{A}$*-valuation* α. Thus, the denotation of an $\mathcal{OSF}$ term t for all possible valuations of the variables is given by the set:

$$[\![t]\!]^{\mathfrak{A}} \overset{\text{DEF}}{=} \bigcup_{\alpha:\mathcal{V}\mapsto D^{\mathfrak{A}}} [\![t]\!]^{\mathfrak{A},\alpha}. \tag{8}$$

Definition 3 ($\mathcal{OSF}$ Term Subsumption) *Let t and t' be two $\mathcal{OSF}$ terms. Then, $t \leq t'$ ("t is subsumed by t'") if and only if, for all $\mathcal{OSF}$ algebras $\mathfrak{A}$, $[\![t]\!]^{\mathfrak{A}} \subseteq [\![t']\!]^{\mathfrak{A}}$.*

An $\mathcal{OSF}$ term $t = X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n)$ is said to be *"in normal form"* whenever all the following properties hold:

- s is a non-bottom sort in $\mathcal{S}$;
- $f_1, \ldots, f_n$ are pairwise distinct features in $\mathcal{F}$, $n \geq 0$,
- $t_1, \ldots, t_n$ are all $\mathcal{OSF}$ terms in *normal* form,
- no variable occurs in t with more than one non-$\top$ sort. That is, if V occurs in t both as $V : s$ and $V : s'$, then $s = \top$ or $s' = \top$.

An $\mathcal{OSF}$ term in normal form is called a *"ψ-term."* We call Ψ the set of all ψ-terms.

3.1.3 $\mathcal{OSF}$ CONSTRAINTS

A logical reading of an $\mathcal{OSF}$ term is immediate as its information content can be characterized by a simple formula. For this purpose, we need a simple clausal language as follows.

An atomic *$\mathcal{OSF}$ constraint* is one of (1) $X : s$, (2) $X \doteq X'$, or (3) $X.f \doteq X'$, where X and X' are variables in $\mathcal{V}$, s is a sort in $\mathcal{S}$, and f is a feature in $\mathcal{F}$. A (conjunctive) *$\mathcal{OSF}$ constraint* is a conjunction (*i.e.*, a set) of atomic $\mathcal{OSF}$ constraints $\phi_1 \,\&\, \ldots \,\&\, \phi_n$. Given an $\mathcal{OSF}$ algebra $\mathfrak{A}$, an $\mathcal{OSF}$ constraint ϕ is *satisfiable* in $\mathfrak{A}$, $\mathfrak{A}, \alpha \models \phi$, if there exists a valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$ such that:

$$\begin{array}{lll@{\quad}l} \mathfrak{A},\alpha & \models X : s & \textbf{iff} & \alpha(X) \in s^{\mathfrak{A}}; \\ \mathfrak{A},\alpha & \models X \doteq Y & \textbf{iff} & \alpha(X) = \alpha(Y); \\ \mathfrak{A},\alpha & \models X.f \doteq Y & \textbf{iff} & f^{\mathfrak{A}}(\alpha(X)) = \alpha(Y) \\ \mathfrak{A},\alpha & \models \phi \,\&\, \phi' & \textbf{iff} & \mathfrak{A},\alpha \models \phi \textbf{ and } \mathfrak{A},\alpha \models \phi'. \end{array} \tag{9}$$

We can always associate with an $\mathcal{OSF}$ term $t = X : s(f_1 \Rightarrow t_1, \ldots, f_n \Rightarrow t_n)$ a corresponding $\mathcal{OSF}$ constraint $\varphi(t)$ as follows:

$$\begin{array}{lllllll} \varphi(t) & \overset{\text{DEF}}{=} & X : s & \& \; X.f_1 \doteq X_1 & \& & \ldots & \& \; X.f_n \doteq X_n \\ & & & \& \; \varphi(t_1) & \& & \ldots & \& \; \varphi(t_n) \end{array} \tag{10}$$

where $X_1, \ldots, X_n$ are the roots of $t_1, \ldots, t_n$, respectively. We say that $\varphi(t)$ is obtained from *dissolving* the $\mathcal{OSF}$ term t. It has been shown that the set-theoretic denotation of an $\mathcal{OSF}$ term and the logical semantics of its dissolved form coincide exactly (Aït-Kaci and Podelski, 1993):

$$[\![t]\!]^{\mathfrak{A}} \overset{\text{DEF}}{=} \{\alpha(X) \mid \alpha \in \textbf{VAL}(\mathfrak{A}),\ \mathfrak{A},\alpha \models C_t^{\exists}(X)\}$$

where $C_t[X]$ is shorthand for the formula $X \doteq \textbf{ROOT}(t) \,\&\, \varphi(t)$, and $C_t^{\exists}[X]$ abbreviates the formula $\exists\textbf{VAR}(t)\; C_t[X]$.

3.1.4 $\mathcal{OSF}$ UNIFICATION

Definition 4 (Solved $\mathcal{OSF}$ Constraints) *An $\mathcal{OSF}$ constraint ϕ is said to be in solved form if for every variable X, ϕ contains:*

- *at most one sort constraint $X : s$, with $\bot < s$; and,*
- *at most one feature constraint $X.f \doteq X'$ for each f;*
- *if $X \doteq X' \in \phi$, then X does not appear anywhere else in ϕ.*

Again, given an $\mathcal{OSF}$ constraint ϕ, non-deterministically applying any applicable rule among the rules shown in Figure 4 until none apply will always terminate in the inconsistent constraint or a solved $\mathcal{OSF}$ constraint. Each of these rules can easily be shown to be correct. They can also just easily be shown to be confluent modulo variable renaming. The rules of Figure 4 are solution-

$(\mathcal{O}_1)$ **SORT INTERSECTION**:
$$\frac{\phi \;\&\; X : s \;\&\; X : s'}{\phi \;\&\; X : s \wedge s'}$$

$(\mathcal{O}_2)$ **INCONSISTENT SORT**:
$$\frac{\phi \;\&\; X : \bot}{X : \bot}$$

$(\mathcal{O}_3)$ **FEATURE FUNCTIONALITY**:
$$\frac{\phi \;\&\; X.f \doteq X' \;\&\; X.f \doteq X''}{\phi \;\&\; X.f \doteq X' \;\&\; X' \doteq X''}$$

$(\mathcal{O}_4)$ **VARIABLE ELIMINATION**:
$[\;$ **if** $X \neq X'$ **and** $X \in \textbf{VAR}(\phi)\;]$
$$\frac{\phi \;\&\; X \doteq X'}{\phi[X/X'] \;\&\; X \doteq X'}$$

$(\mathcal{O}_5)$ **VARIABLE CLEANUP**:
$$\frac{\phi \;\&\; X \doteq X}{\phi}$$

Figure 4: **Basic $\mathcal{OSF}$-constraint normalization rules**

preserving, finite terminating, and confluent (modulo variable renaming). Furthermore, they always result in a normal form that is either the inconsistent constraint or an $\mathcal{OSF}$ constraint in solved form (Aït-Kaci and Podelski, 1993). These rules are all we need to perform the unification of two $\mathcal{OSF}$ terms. Namely, two terms t_1 and t_2 are $\mathcal{OSF}$ unifiable if and only if the normal form of $\textbf{ROOT}(t_1) \doteq \textbf{ROOT}(t_2) \;\&\; t_1 \;\&\; t_2$ is not $\bot$.

An $\mathcal{OSF}$ constraint ϕ in solved form is always satisfiable in a canonical interpretation—*viz.*., the $\mathcal{OSF}$ graph algebra Ψ (Aït-Kaci and Podelski, 1993). As a consequence, the $\mathcal{OSF}$-constraint normalization rules yield a decision procedure for the satisfiability of $\mathcal{OSF}$ constraints.

3.1.5 DISJUNCTION AND NEGATION

We now extend basic $\mathcal{OSF}$ terms to express disjunctive and negative information. The syntax of $\mathcal{OSF}$ terms is generalized as shown in Figure 5. We use the standard BNF grammar notation where '$[X]$' means "*optional* X", 'X^*' means "*a sequence of zero or more* X's", and 'X^+' means "*a sequence of one or more* X's." Next, we explain what these new constructs mean and how they are handled as constraints.

OSFTERM	::=	[VARIABLE :] TERM
TERM	::=	CONJUNCTIVETERM
	|	DISJUNCTIVETERM
	|	NEGATIVETERM
CONJUNCTIVETERM	::=	SORT [(ATTRIBUTE$^+$)]
ATTRIBUTE	::=	FEATURE $\Rightarrow$ OSFTERM
DISJUNCTIVETERM	::=	{ OSFTERM [; OSFTERM]* }
NEGATIVETERM	::=	$\neg$ OSFTERM

Figure 5: **Extended $\mathcal{OSF}$ term syntax**

§**Disjunction –** In Section 3.1.1, the $\mathcal{OSF}$ sort signature $\mathcal{S}$ is required to be a (lower) semilattice with $\top$ and $\bot$. This means that a unique **GLB** exists for any pair of sorts. Yet, it is common to find sort signatures for which this is not the case. For example, the sort signature shown in Figure 6 violates this condition; therefore, it is not a semilattice.

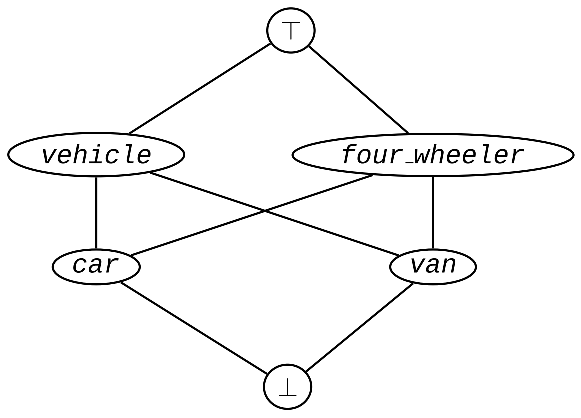

Figure 6: **Example of a non-semilattice sort signature**

However, since the ordering on sorts denotes set inclusion, sort conjunction denotes set intersection and is the **GLB** for the sort ordering. Therefore, by semantic duality, sort *disjunction* denotes set union and is *the least upper bound* (**LUB**) of two sorts. Hence, a *disjunctive* $\mathcal{OSF}$ term is an expression of the form $\{t_1; \ldots; t_n\}$ where $n \geq 0$, and t_i is either a conjunctive $\mathcal{OSF}$ term as defined in Section 3.1.2 or again a disjunctive $\mathcal{OSF}$ term.

The denotation of a disjunctive term is simply the union of the denotations of its constituents. Namely, given an $\mathcal{OSF}$ interpretation $\mathfrak{A}$, and an $\mathfrak{A}$-valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$:

$$[\![\{t_1; \ldots; t_n\}]\!]^{\mathfrak{A},\alpha} \overset{\text{DEF}}{=} \bigcup_{1 \leq i \leq n} [\![t_i]\!]^{\mathfrak{A},\alpha}. \tag{11}$$

Thus, it follows from the interpretation of a disjunctive $\mathcal{OSF}$ term $\{t_1; \ldots; t_n\}$ that, when $n = 0$, $\{\} \simeq \bot$; and, when $n = 1$, $\{t\} \simeq t$.

Similarly, a disjunctive $\mathcal{OSF}$ *constraint* is a construct of the form $\phi_1 \;\|\; \ldots \;\|\; \phi_n$, where the ϕ_i's are either atomic $\mathcal{OSF}$ constraints, conjunctive $\mathcal{OSF}$ constraints as defined in Section 3.1.3, or again disjunctive $\mathcal{OSF}$ constraints. Given an $\mathcal{OSF}$ algebra $\mathfrak{A}$, a disjunctive $\mathcal{OSF}$ constraint $\phi \;\|\; \phi'$ is *satisfiable* in $\mathfrak{A}$ iff either ϕ or ϕ' is satisfiable in $\mathfrak{A}$. Namely,

$$\mathfrak{A}, \alpha \models \phi \;\|\; \phi' \quad \textbf{iff} \quad \mathfrak{A}, \alpha \models \phi \;\; \textbf{or} \;\; \mathfrak{A}, \alpha \models \phi'. \tag{12}$$

The $\mathcal{OSF}$-constraint normalization rules handling disjunction are given in Figure 7. They simply

$(\mathcal{O}_6)$ **<u>NON-UNIQUE GLB</u>:**

$$\left[\begin{array}{l}\textbf{if } \; s_i \in \max_{\leq}\{t \in \mathcal{S} \mid t \leq s \;\; \textbf{and} \;\; t \leq s'\} \\ \forall i, \; i = 1, \ldots, n\end{array}\right] \qquad \frac{\phi \;\&\; X : s \;\&\; X : s'}{\phi \;\&\; \big(X : s_1 \;\|\; \ldots \;\|\; X : s_n\big)}$$

$(\mathcal{O}_7)$ **<u>DISTRIBUTIVITY</u>:**

$$\frac{\phi \;\&\; \big(\phi' \;\|\; \phi''\big)}{\big(\phi \;\&\; \phi'\big) \;\|\; \big(\phi \;\&\; \phi''\big)}$$

$(\mathcal{O}_8)$ **<u>DISJUNCTION</u>:**

$$\frac{\phi \;\|\; \phi'}{\phi}$$

Figure 7: **Disjunctive $\mathcal{OSF}$-constraint normalization**

consist in non-deterministic branching in the direction of either of the disjuncts. Recall that all our normalization rules work up to associativity, commutativity, and idempotence of both the $\&$ and $\|$ operators. The $\mathcal{OSF}$ term-dissolving function φ is extended to disjunctive $\mathcal{OSF}$ *terms* by transforming them into disjunctive $\mathcal{OSF}$ *constraints* as follows:

$$\varphi(\{t_1; \ldots; t_n\}) \overset{\text{DEF}}{=} \varphi(t_1) \;\|\; \ldots \;\|\; \varphi(t_n).$$

Note that we can as well extend the syntax of $\mathcal{OSF}$ terms by allowing disjunctive sorts where sort symbols are expected. A disjunctive sort is of the form $\{s_1; \ldots; s_n\}$, where the s_i's are either sort symbols in $\mathcal{S}$ or again disjunctive sorts. In this case:

$$\varphi\big(X:\{s_1;\ldots;s_m\}(f_i \Rightarrow t_i)_{i=1}^n\big) \overset{\text{DEF}}{=} \varphi\big(X:s_1(f_i \Rightarrow t_i)_{i=1}^n\big) \parallel \ldots \parallel \varphi\big(X:s_m(f_i \Rightarrow t_i)_{i=1}^n\big).$$

§**Negation –** We proceed similarly for negation. Namely, the denotation of the negative $\mathcal{OSF}$ term $\neg t$, given an $\mathcal{OSF}$ interpretation $\mathfrak{A}$ and $\mathfrak{A}$-valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{A}}$, is defined by:

$$[\![\neg t]\!]^{\mathfrak{A},\alpha} \overset{\text{DEF}}{=} D^{\mathfrak{A}} \backslash [\![t]\!]^{\mathfrak{A},\alpha}. \tag{13}$$

Accordingly, the $\mathcal{OSF}$ term-dissolving function φ is extended by the equations shown in Fig. 8, where X_i' is a new variable and $X_i = \textbf{ROOT}(t_i)$ is the root variable of t_i, for $i = 1, \ldots, n$ and $n \geq 0$, and:

$$\varsigma(X:s) \overset{\text{DEF}}{=} \begin{cases} \varsigma(X:s') & \textbf{if } \; s = \overline{\overline{s'}}, \\ \varsigma(X:s_1) \;\&\; \ldots \;\&\; \varsigma(X:s_n) & \textbf{if } \; s = \overline{\{s_1;\ldots;s_n\}}, \\ X:s & \textbf{otherwise}. \end{cases}$$

$$\begin{array}{rcl} \varphi(\neg(\neg t)) & \overset{\text{DEF}}{=} & \varphi(t) \\ \varphi(\neg\{t_1;\ldots;t_n\}) & \overset{\text{DEF}}{=} & \varphi(\neg t_1) \;\&\; \ldots \;\&\; \varphi(\neg t_n) \\ \varphi(\neg X:s(f_i \Rightarrow t_i)_{i=1}^n) & \overset{\text{DEF}}{=} & \varsigma(X:\overline{s}) \;\parallel\; X.f_1 \doteq X_1 \;\&\; \varphi(\neg t_1) \\ & & \parallel\; X.f_1 \doteq X_1' \;\&\; X_1' \not\doteq X_1 \;\&\; \varphi(t_1) \\ & \ldots & \parallel\; X.f_n \doteq X_n \;\&\; \varphi(\neg t_n) \\ & & \parallel\; X.f_n \doteq X_n' \;\&\; X_n' \not\doteq X_n \;\&\; \varphi(t_n) \end{array}$$

Figure 8: **Negative $\mathcal{OSF}$ term dissolution**

Thus, dissolving a negative $\mathcal{OSF}$ constraint transforms it into a possibly disjunctive $\mathcal{OSF}$ constraint where the symbol '$\neg$' no longer occurs, and atomic constraints are as before, but also disequality constraints $X \not\doteq Y$ and complemented sort constraints of the form $X : \overline{s}$, for $X, Y \in \mathcal{V}$ and $s \in \mathcal{S}$. The notation $\overline{s}$, for $s \in \mathcal{S}$, denotes the *complement* of sort s; *viz.*, $\overline{s}^{\mathfrak{A}} \overset{\text{DEF}}{=} D^{\mathfrak{A}} \backslash s^{\mathfrak{A}}$.

Satisfiability of the new atomic $\mathcal{OSF}$ *disequality* constraint $X \not\doteq X'$, for $X \in \mathcal{V}$, is defined as:

$$\mathfrak{A}, \alpha \models X \not\doteq X' \quad \textbf{iff} \quad \alpha(X) \neq \alpha(X'). \tag{14}$$

Because dissolution of a negative $\mathcal{OSF}$ term eliminates the negation symbol '$\neg$' altogether by introducing complemented sorts and disequalities among variables, we need two additional rules for normalizing negative $\mathcal{OSF}$ constraints. They are given in Figure 9.

3.1.6 ADDITIONAL AXIOMS

The set of $\mathcal{OSF}$-constraints normalization rules presented thus far may be strengthened with useful additional axioms that enable important functionality commonly found in object/class-based systems—*viz.*., *partial features*, *element sorts*, and *aggregates*. We next describe additional rules that achieve such functionality while preserving confluence and finite termination when combined with the previous $\mathcal{OSF}$ constraint-normalization rules.

$(\mathcal{O}_9)$ **DISEQUALITY**:
$$\frac{\phi \ \&\ X \neq X}{\bot}$$

$(\mathcal{O}_{10})$ **COMPLEMENT**:
$\left[\ \textbf{if}\ s' \in \max_{\leq}\{t \in \mathcal{S} \mid s \not\leq t\ \textbf{and}\ t \not\leq s\}\ \right]$
$$\frac{\phi \ \&\ X : \overline{s}}{\phi \ \&\ X : s'}$$

Figure 9: **Negative $\mathcal{OSF}$-constraint normalization**

§**Partial features –** Given a feature f, its *domain* $\mathbf{DOM}(f)$ is the set of maximal sorts $\{s_1, \ldots, s_n\}$ in $\mathcal{S}$ such that f is defined—*i.e.*, $\mathbf{DOM} : \mathcal{F} \mapsto \mathbf{2}^{\mathcal{S}}$. A feature f such that is $\mathbf{DOM}(f) = \{\top\}$ is said to be *total*. A feature f is nowhere defined whenever $\mathbf{DOM}(f) = \{\bot\}$. It is partial when it is not total although defined on some non bottom sort. Given a feature $f \in \mathcal{F}$, for each sort $s \in \mathbf{DOM}(f)$, the *range* of f in s is the sort $\mathbf{RAN}_s(f) \in \mathcal{S}$ of values taken by feature f on sort s. The $\mathcal{OSF}$-constraint normalization rule for enforcing such partial features is shown as *"Partial Feature"* in Figure 10. Computational linguists, who have borrowed heavily from the $\mathcal{OSF}$ formalism to express HPSG grammars for natural-language processing, call the axiom enforced by this rule "feature appropriateness" (Carpenter, 1991).

$(\mathcal{O}_{11})$ **PARTIAL FEATURE**:
$\left[\ \textbf{if}\ s \in \mathbf{DOM}(f)\ \ \textbf{and}\ \ \mathbf{RAN}_s(f) = s'\ \right]$
$$\frac{\phi \ \&\ X.f \doteq X'}{\phi \ \&\ X.f \doteq X' \ \&\ X : s \ \&\ X' : s'}$$

$(\mathcal{O}_{12})$ **WEAK EXTENSIONALITY**:
$\left[\begin{array}{l}\textbf{if}\ s \in \mathcal{E}\ \ \textbf{and}\ \ \forall f \in \mathbf{ARITY}(s) : \\ \{X.f \doteq Y, X'.f \doteq Y\} \subseteq \phi\end{array}\right]$
$$\frac{\phi \ \&\ X : s \ \&\ X' : s}{\phi \ \&\ X : s \ \&\ X \doteq X'}$$

$(\mathcal{O}_{13})$ **VALUE AGGREGATION**:
$\left[\begin{array}{l}\textbf{if}\ s \text{ and } s' \text{ are both subsorts of} \\ \text{commutative monoid } \langle\star, \mathbf{1}_\star\rangle\end{array}\right]$
$$\frac{\phi \ \&\ X = e : s \ \&\ X = e' : s'}{\phi \ \&\ X = e \star e' : s \wedge s'}$$

Figure 10: **Additional $\mathcal{OSF}$-constraint normalization rules**

§**Element sorts –** A sort denotes a set. When this set is a singleton, the sort is assimilated to the value contained in the denoted singleton. The normalization rules to do so are as follows.

Let $\mathcal{E}$ (for "*element*," or "*extensional*," sorts) be the set of sorts in $\mathcal{S}$ that denote singletons. Define the *arity* $\mathbf{ARITY}(e)$ of such an element sort e giving its *feature arity* as a set of features—

i.e., $\textbf{ARITY} : \mathcal{E} \mapsto \mathbf{2}^{\mathcal{F}}$. The set $\textbf{ARITY}(e)$ is the set of features that completely determine the unique element of sort e. In other words, whenever all features of $\textbf{ARITY}(e)$ denote singletons, then so does e. All such values ought to be uniquely identified. Note in passing that all atomic constants in $\mathcal{E}$ always have empty arity. For example, for any number n, $\textbf{ARITY}(n) = \emptyset$. The $\mathcal{OSF}$-constraint normalization rule that enforces this uniqueness axiom on element sorts is called *"Weak Extensionality"* as shown in Figure 10.

With this rule, for example, if $\mathcal{S} = \{\top, \bot, \mathtt{nil}, \mathtt{cons}, \mathtt{list}, \mathtt{nat}, 0, 1, 2, \ldots\}$ such that $\mathtt{nil} < \mathtt{list}$, $\mathtt{cons} < \mathtt{list}$, $n < \mathtt{nat}$ for $n \in \mathbb{N}$ (where $<$ is the subsort ordering). Let $\mathcal{E} = \{\mathtt{nil}, \mathtt{cons}, n\}$, $(n \in \mathbb{N})$, such that $\textbf{ARITY}(\mathtt{nil}) = \emptyset$, $\textbf{ARITY}(\mathtt{cons}) = \{\mathtt{head}, \mathtt{tail}\}$, and $\textbf{ARITY}(n) = \emptyset$ for $n \in \mathbb{N}$. Then, the $\mathcal{OSF}$ term:

$$X : \mathtt{cons}(\mathtt{head} \Rightarrow 1, \mathtt{tail} \Rightarrow \mathtt{nil}) \ \& \ Y : \mathtt{cons}(\mathtt{head} \Rightarrow 1, \mathtt{tail} \Rightarrow \mathtt{nil})$$

is normalized into:

$$X : \mathtt{cons}(\mathtt{head} \Rightarrow 1, \mathtt{tail} \Rightarrow \mathtt{nil}) \ \& \ X = Y$$

This rule is called *"weak"* because it can only enforce uniquess of *acyclic* elements. Rules enforcing the necessary stronger condition for cyclic terms can also be given (see Appendix Section C).

§**Relational features and aggregation –** The $\mathcal{OSF}$ formalism deals with functional features. However, relational features may also come handy. A relational feature is a binary relation or, equivalently, a set-valued function. In other words, a multi-valued functional attribute may be aggregated into sets. Such a set-valued feature is called a *"role"* or *"property"* in $\mathcal{DL}$ lingo (*e.g.*, in OWL)—see Section 3.2. Indeed, combining rules *"Sort Intersection"* with *"Feature Functionality"* (see Figure 4) enforces that a variable's sort, and hence value, may only be computed by intersection of consistent sorts. On the other hand, a relational feature denotes a set-valued function, and normalization must thus provide a means to aggregate mutually distinct values of some sort.

This semantics is easily accommodated with the following value aggregation rule, which generalizes the *"Sort Intersection"* rule. Incidentally, computing sort intersection is doable in constant time by encoding sorts as binary vectors as shown in (Aït-Kaci et al., 1989). This is a tremendous source of efficiency when compared to an encoding of a class hierarchy's partial order using symbolic $\mathcal{FOL}$ rules, as done in F-Logic for example (Kifer et al., 1995). The notation for the atomic constraint "$X : s$" is generalized to carry an optional value $e \in \mathcal{E}$ (*i.e.*, e is an extensional sort): "$X = e : s$" means "X has value e of sort s"—where $X \in \mathcal{V}$, $e \in \mathcal{E}$, $s \in \mathcal{S}$. The shorthand "$X = e$" means "$X = e : \top$." When the sort $s \in \mathcal{S}$ is a commutative monoid $\langle \star, \mathbf{1}_\star \rangle$, the shorthand "$X : s$" means "$X = \mathbf{1}_\star : s$."

The semantics conditions (9) are simply extended with:

$$\mathfrak{A}, \alpha \ \models \ X = e : s \ \textbf{ iff } \ e^{\mathfrak{A}} \in s^{\mathfrak{A}} \ \textbf{ and } \ \alpha(X) = e^{\mathfrak{A}}. \tag{15}$$

Now, recall that any monoid $M = \langle \star, \mathbf{1}_\star \rangle$ is quasi-ordered with the $\star$-*prefix* relation $\prec_\star$. This quasi-ordering (or preorder) is the natural approximation ordering for elements of the monoid. Thus, element values of a sort that denotes a commutative monoid may be composed using this monoid's operation. In particular, such a monoid operation may be that of a *set* constructor—*i.e.*, an *associative commutative idempotent* constructor.

Note that the *"Value Aggregation"* rule in Figure 10 is more general than need be for just accommodating sets. Indeed, it can accommodate other collection structures such as lists (free monoid), multisets (commutative non-idempotent), or even other computed (as opposed to constructed) commutative aggregation operations such as min, max, sum, product, *etc.*, … Thus, one may use this rule by using $\mathbf{AGGREGATE}(f, s, m, \star, \mathbf{1}_\star)$ to declare that feature f takes values in sort range m denoting a specific commutative monoid $\langle \star, \mathbf{1}_\star \rangle$ when f is applied on sort s (*i.e.*, $s \in \mathbf{DOM}(f)$ and $\mathbf{RAN}_s(f) = m$). In other words,

$$X : s \;\&\; X.f \doteq Y \;\&\; Y = \mathbf{1}_\star : m. \tag{16}$$

Then, Rule *"Partial Feature"* used in conjunction with Rule *"Value Aggregation"* rule of Fig. 10 will work correctly.

Note also that we require a *commutative* monoid to ensure confluence of this rule with the other $\mathcal{OSF}$-constraint normalization rules in a non-deterministic normalization setting. In other words, the order in which the rules are applied does not matter on the outcome of the aggregation. Hence, the $*$ operation on the two values e and e' may then be defined as the appropriate aggregation. Thus may elements be aggregated by constraint normalization into any suitable form we wish (*e.g.*, list, set, multiset, sum, product, min, max, and, or, *etc.*, …). The notion of a monoid is all we need to express very powerful aggregative data structures such as the *monoid comprehensions* calculus (Fegaras and Maier, 2000; Grust, 2003). Indeed, the λ-calculus can be simply and effectively extended with the power of aggregative monoidal structures (*i.e.*, lists, sets, multisets) and accumulators (*i.e.*, sum, product, min, max, *etc.*.) using a simple notion of *monoid homomorphism*, which provides an elegant formalism way to express declaratively iterative computation over aggregative constructs.

Decidability results concerning the differences between attributive concepts using functional features *vs.* relation roles are reviewed in (Schmidt-Schauß and Smolka, 1991). Aggregation has also been considered in the same setting in (Baader and Sattler, 1997) with similar decidability results. This last work offers intriguing potential connections with the paradigm of declarative aggregation as described in (Fegaras and Maier, 2000) or (Grust, 2003) where a versatile computable algebraic theory of monoid comprehensions is defined in terms of monoid homomorphisms allowing the perspicous declarative descriptions of aggregates. The monoid comprehension calculus is a conservative extension of the λ-calculus and the object-relational model, and enjoys algebraic properties that greatly facilitate query optimization.

§**Ontology unfolding –** Description Logics support the notion of *terminology*, or *TBox*, which is a means to define concepts in terms of other concepts (Baader and Nutt, 2003). In other words, a TBox specifies equations defining non-primitive concepts in terms of base concepts and themselves, thus allowing cyclic concept definitions. These may be viewed as recursive type equations and may be solved semantically and proof-theoretically depending on the nature of the $\mathcal{DL}$ one uses (Aït-Kaci, 1984, 1986; Bucheit et al., 1993; Baader and Nutt, 2003).

The $\mathcal{OSF}$ formalism offers a terminological facility also in the form of sort equations (Aït-Kaci et al., 1997). This is what we call a *conceptual ontology* since it defines concepts. It may be viewed as a schema abbreviating some sorts in terms of others. We restrict ourselves to sort equations of the form $s \equiv t$, where s is a sort and t is an $\mathcal{OSF}$ term, as for $\mathcal{DL}$'s TBox definitions (Baader and Nutt, 2003). More exactly, $\mathcal{DL}$ does not use $\mathcal{OSF}$ terms but $\mathcal{DL}$ concept expressions, and it does not deal with path equality constraints. We call such a "TBox" an *$\mathcal{OSF}$ theory*.

Clearly, expressivity of the $\mathcal{OSF}$ constraint calculus is greatly enhanced when sorts may be recursively defined, especially when variables may appear in sort definitions (Aït-Kaci et al., 1997; Zajac, 1992; Krieger and Schäfer, 1994). A conceptual ontology is in fact very close to a class schema definition in object-oriented programming. Although in object-oriented programming, typically, classes and object do not enjoy the expressivity offered by either ψ-terms or $\mathcal{DL}$ concept expressions. Objects are made according to blueprints specified as (recursive) *class* definitions. A class acts as a template, restricting the aspect of the objects that are its instances. Thus, a convenience for expressing conceptual ontologies in the form of *sort definitions* is provided by the $\mathcal{OSF}$ formalism, expanding in this way the capability of the basic and additional $\mathcal{OSF}$ axioms of Figures 4 and 10 to express more complex integrity constraints on objects.

This enables an incompletely specified object to remain always consistent with its class as information accrues about this object. A sort definition associates a ψ-term structure to a sort. Intuitively, one may then see a sort as an *abbreviation* of a more complex structure. Hence, a sort definition specifies a template that an object of this sort must abide by, whenever it uses any part of the structure appearing in the ψ-term defining the sort.

For example, consider the ψ-term:

$$\begin{aligned}&\texttt{person}(\texttt{name} \Rightarrow \top(\texttt{last} \Rightarrow \texttt{string}),\\ &\qquad\texttt{spouse} \Rightarrow \top(\texttt{spouse} \Rightarrow \top,\\ &\qquad\qquad\qquad\texttt{name} \Rightarrow \top(\texttt{last} \Rightarrow \texttt{"Smith"}))).\end{aligned}$$

Without sort definitions, there is no reason to expect that this structure should be incomplete, or inconsistent, as intended. Let us now define the sort `person` as an abbreviation of the structure:

$$\begin{aligned}&P : \texttt{person}(\texttt{name} \Rightarrow \texttt{id}(\texttt{first} \Rightarrow \texttt{string},\\ &\qquad\qquad\qquad\qquad\texttt{last} \Rightarrow S : \texttt{string}),\\ &\qquad\texttt{spouse} \Rightarrow \texttt{person}(\texttt{name} \Rightarrow \texttt{id}(\texttt{last} \Rightarrow S),\\ &\qquad\qquad\qquad\qquad\texttt{spouse} \Rightarrow P)).\end{aligned}$$

This definition of the sort `person` expresses the expectation whereby, whenever a `person` object has features `name` and `spouse`, these should lead to objects of sort `id` and `person`, respectively. Moreover, if the features `first` and `last` are present in the object indicated by `name`, then they should be of sort `string`. Also, if a `person` object had sufficient structure as to involve feature paths `name.last` and `spouse.name.last`, then these two paths should lead to the same object. And so on.

For example, with this sort definition, the `person` object with last name `"Smith"` above should be made to comply with the definition template by being *normalized* into the term:

$$\begin{aligned}&X : \texttt{person}(\texttt{name} \Rightarrow \texttt{id}(\texttt{last} \Rightarrow N : \texttt{"Smith"}),\\ &\qquad\texttt{spouse} \Rightarrow \texttt{person}(\texttt{spouse} \Rightarrow X,\\ &\qquad\qquad\qquad\qquad\texttt{name} \Rightarrow \texttt{id}(\texttt{last} \Rightarrow N))).\end{aligned}$$

In this example, it is assumed, of course, that $\texttt{"Smith"} < \texttt{string}$.

Note that sort definitions are not *feature declarations*. Namely, sort definitions do not enforce the existence, or lack thereof, of the specified features that appear in a sort's definition for every object of that sort. This kind of consistency checking is performed by sort signatures schema constraints enforced by rules such as the *"Partial Feature" $\mathcal{OSF}$*-constraint normalization rule in Figure 10 using the declared domains and ranges of features. Rather, a sort's definition specifies sort and equality constraints on feature paths from the sort being defined. For instance, we could use $\mathtt{person}(\mathtt{hobby} \Rightarrow \mathtt{movie\_going})$ without worrying about violating the template for $\mathtt{person}$ since the feature $\mathtt{hobby}$ is not constrained by the sort definition of $\mathtt{person}$. However, it could be further constrained by declaring feature $\mathtt{hobby}$'s domains and ranges.

This lazy inheritance of structural constraints from the class template into an object's structure is invaluable for efficiency reasons. Indeed, if all the (possibly voluminous) template structure of a sort were to be systematically expanded into an object of this sort that uses only a tiny portion of it, space and time would be wasted. More importantly, lazy inheritance is a way to ensure termination of consistency checking. For example, the sort definition of $\mathtt{person}$ above is recursive, as it involves the sort $\mathtt{person}$ in its body. Completely expanding these sorts into their templates would go on for ever.

An incidental benefit of sort-unfolding in the context of a sort semilattice is what we call *proof memoizing*. Namely, once the definition of a sort for a variable X has been unfolded, and the attached constraints proven for X, this proof is automatically and efficiently recorded by the expanded sort. The accumulation of proofs corresponds exactly to the greatest lower bound operation. Besides the evident advantage of not having to repeat computations, this memoizing phenomenon accommodates expressions that would loop otherwise.

Let us take a small example to illustrate this point. Lists can be specified by declaring $\mathtt{nil}$ and $\mathtt{cons}$ to be subsorts of the sort $\mathtt{list}$ and by defining for the sort $\mathtt{cons}$ the template ψ-term $\mathtt{cons}(\mathtt{head} \Rightarrow \top, \mathtt{tail} \Rightarrow \mathtt{list})$. Now, consider the expression $X : [1|X]$, the circular list containing the one element 1—*i.e.*, desugared as $X : \mathtt{cons}(\mathtt{head} \Rightarrow 1, \mathtt{tail} \Rightarrow X)$. Verifying that X is a list, since it is the $\mathtt{tail}$ of a $\mathtt{cons}$, terminates immediately on the grounds that X has already been memoized to be a $\mathtt{cons}$, and $\mathtt{cons} < \mathtt{list}$. In contrast, the semantically equivalent Prolog program with two clauses: $\mathtt{list}([])$ and $\mathtt{list}([H|T])$ `:-` $\mathtt{list}(T)$ would make the goal $\mathtt{list}(X : [1|X])$ loop. (See Sections 1.2 and 3.3.4.)

A formal and practical solution for the problem of checking the consistency of a ψ-term object modulo a sort hierarchy of structural class templates is described in (Aït-Kaci et al., 1997). The problem (called *"$\mathcal{OSF}$ theory unification"*) is formalizable in First-Order Logic ($\mathcal{FOL}$): objects as $\mathcal{OSF}$ constraint formulae, classes as axioms defining an $\mathcal{OSF}$ theory, class inheritance as testing the satisfiability of an $\mathcal{OSF}$ constraint in a model of the $\mathcal{OSF}$ theory. As a result, models for $\mathcal{OSF}$ theories may be shown to exist. It is shown in (Aït-Kaci et al., 1997) that the $\mathcal{OSF}$ theory unification problem is undecidable. However, checking the consistency of an $\mathcal{OSF}$ term modulo an $\mathcal{OSF}$ theory is semi-decidable. This is achieved by constraint normalization rules for $\mathcal{OSF}$ theory unification given in (Aït-Kaci et al., 1997), which is complete for detecting incompatibility of an object with respect to an $\mathcal{OSF}$ theory; *i.e.*, checking non-satisfiability of a constraint in a model of the axioms. This system specifies the third Turing-complete calculus used in LIFE (Aït-Kaci and Podelski, 1993), besides its logical (Horn rules over ψ-terms) and the functional one (rewrite rules over ψ-terms).

Remarkably, the $\mathcal{OSF}$-theory constraint normalization rule system given in (Aït-Kaci et al., 1997) enjoys an interesting property: it consists of a set of ten meaning-preserving syntax-transformation rules that is partitioned into two complementary rule subsets: a system of nine confluent and terminating *weak* rules, and one additional *strong* rule, whose addition to the other rules preserves confluence, but may lead to non-termination. There are two nice consequences of this property:

1. it yields *a complete normalization strategy* consisting of repeatedly normalizing a term first with the terminating rules, and then apply, if at all necessary, the tenth rule; and,
2. it provides a formally correct *compilation scheme* of $\mathcal{OSF}$ theories (*i.e.*, multiple-inheritance constrained class hierarchies) by partial evaluation since all sort definitions of a theory can be normalized with respect to the theory itself using only the weak rules.

3.2 Description logic

Description Logic ($\mathcal{DL}$) is a formal language for describing simple sets of objects—called *concepts*—that are subsets of elements of a domain of interpretation, and properties thereof—called *roles*—that are binary relations on this universe.

3.2.1 $\mathcal{DL}$ SYNTAX

$\mathcal{DL}$'s syntax is defined by a grammar of expressions for *concept descriptions* making up complex concepts by combining simpler ones with operators denoting elementary set operations. As is the case for $\mathcal{OSF}$ logic, there are many variations of $\mathcal{DL}$ languages—$\mathcal{DL}$ *dialects*—depending on how expressive one needs to be; that is, what specific constructs are supported. This entails as many computational and decidability properties enjoyed by (or plaguing) the various expressivity classes of such logical dialects. Which particular $\mathcal{DL}$ dialect one should be concerned with matters only regarding the kinds of inferences one expects to be able to carry out in it, and how inherently expensive in time and space these are. The specific $\mathcal{DL}$ dialects we mention here and there in this paper are simply for illustration. See (Heinsohn et al., 1994) for a thorough survey and comparative analysis of such dialects. The interested reader is also referred to (Lunz, 2006) and (Lambrix, 2006) for a plethora of up-to-date information on $\mathcal{DL}$ literature and (re)sources.

Figure 11 gives grammar rules for a few popular $\mathcal{DL}$ constructs that may be used to build concept and role expressions. In the grammar of Figure 11, the non-terminal symbols 'CONCEPT' and 'ROLE' derive respectively *concept* and *role* expressions. The terminal symbol '$\mathtt{Name}$' is used to stand for names of primitive concepts and roles, as well as constant individual elements of some domain of interpretation. Let $\mathcal{C}$ [resp., $\mathcal{R}$] be the set of concept [resp. role] expressions C [resp. R] generated by this grammar.

In the following sections, we quickly overview a simple set-theoretic denotational semantics for $\mathcal{DL}$ constructs and a syntax-directed constraint-based deductive system for reasoning with $\mathcal{DL}$ knowledge.

3.2.2 $\mathcal{DL}$ SEMANTICS

Let $\mathfrak{I}$ be $\mathcal{DL}$-interpretation structure with domain $D^{\mathfrak{I}}$, a (possibly countably infinite) set. Names of constants denote atomic concepts (*i.e.*, subsets of $D^{\mathfrak{I}}$), atomic roles (*i.e.*, subsets of $D^{\mathfrak{I}} \times D^{\mathfrak{I}}$), or individual elements in $D^{\mathfrak{I}}$. Thus, let $\mathfrak{C}_{\mathtt{Name}}$ [resp., $\mathfrak{R}_{\mathtt{Name}}$; or, resp., $\mathfrak{I}_{\mathtt{Name}}$] be the subset of $D^{\mathfrak{I}}$ [resp., the subset of $D^{\mathfrak{I}} \times D^{\mathfrak{I}}$; or, the individual element in $D^{\mathfrak{I}}$] that the symbol $\mathtt{Name}$ denotes.

Given a set S, the notation $|S|$ denotes the cardinality of S. Given sets A, B, and C, and two binary relations $\alpha \subseteq A \times B$ and $\beta \subseteq B \times C$, their *composition* is the binary relation $\alpha \circ \beta \subseteq A \times C$ defined as: $\alpha \circ \beta \overset{\text{DEF}}{=} \{\langle x, y\rangle \in A \times C \mid \exists z \in B,\ \langle x, z\rangle \in \alpha$ **and** $\langle z, y\rangle \in \beta\}$.

CONCEPT	$::=$	$\top$	*top concept*
	$\mid$	$\bot$	*bottom concept*
	$\mid$	`Name`	*atomic concept*
	$\mid$	$\{$`Name`$, \ldots ,$`Name`$\}$	*concept extension*
	$\mid$	CONCEPT $\sqcap$ CONCEPT	*conjunctive concept*
	$\mid$	CONCEPT $\sqcup$ CONCEPT	*disjunctive concept*
	$\mid$	$\neg$CONCEPT	*negative concept*
	$\mid$	$\forall$ROLE.CONCEPT	*universal-role concept*
	$\mid$	$\exists$ROLE.CONCEPT	*existential-role concept*
	$\mid$	$\leq n.$ROLE	*role max-cardinality concept*
	$\mid$	$\geq n.$ROLE	*role min-cardinality concept*
ROLE	$::=$	`Name`	*atomic role*
	$\mid$	ROLE $\sqcap$ ROLE	*conjunctive role*
	$\mid$	ROLE $\bullet$ ROLE	*composite role*

Figure 11: **Syntax rules for common $\mathcal{DL}$ concept and role constructs**

Given a role R and a, b in $D^{\mathfrak{I}}$, whenever $\langle a, b\rangle \in [\![R]\!]^{\mathfrak{I}}_{\mathfrak{r}}$, we say that a is a *subject* of b for R (or an R-*subject* of b), and we say that b is an *object* of a for R (or an R-*object* of a). For x and y in $D^{\mathfrak{I}}$, we write $R[x]$ to denote the set of all R-objects of x, and $R^{-1}[y]$ the set of all R-subjects of y. That is,

$$\begin{aligned} \forall x \in D^{\mathfrak{I}} \quad R[x] &\overset{\text{DEF}}{=\!=} \{y \in D^{\mathfrak{I}} \mid \langle x, y\rangle \in [\![R]\!]^{\mathfrak{I}}_{\mathfrak{r}}\}, \\ \forall y \in D^{\mathfrak{I}} \quad R^{-1}[y] &\overset{\text{DEF}}{=\!=} \{x \in D^{\mathfrak{I}} \mid \langle x, y\rangle \in [\![R]\!]^{\mathfrak{I}}_{\mathfrak{r}}\}. \end{aligned} \tag{17}$$

Note that for any x and y in $D^{\mathfrak{I}}$, and roles R_1 and R_2, $(R_1 \bullet R_2)[x] = R_2[R_1[x]]$ and $(R_1 \bullet R_2)^{-1}[y] = R_1^{-1}[R_2^{-1}[y]]$, where, $\forall S \subseteq D^{\mathfrak{I}}$, $R[S] \overset{\text{DEF}}{=\!=} \bigcup_{x \in S} R[x]$ and $R^{-1}[S] \overset{\text{DEF}}{=\!=} \bigcup_{y \in S} R^{-1}[y]$. Note also that, by definition, $(R_1 \sqcap R_2)[x] = R_1[x] \cap R_2[x]$. Essentially, the set-theoretic meaning $[\![C]\!]^{\mathfrak{I}}_{\mathfrak{c}}$ of a $\mathcal{DL}$'s concept-description expression C is a subset of the universe of discourse, whose elements possibly verify simple conditions on the extent and cardinality of the object sets of binary relations for which they are subjects. In the sequel, we will simply write $[\![\_]\!]^{\mathfrak{I}}$, or even simpler $[\![\_]\!]$ rather than either $[\![\_]\!]^{\mathfrak{I}}_{\mathfrak{c}}$ or $[\![\_]\!]^{\mathfrak{I}}_{\mathfrak{r}}$ for the semantic mappings whenever it is obvious from the context which sub/super/script is meant.

The semantics of $\mathcal{DL}$ concept and role expressions is given by the semantic mappings $[\![\_]\!]^{\mathfrak{I}}_{\mathfrak{c}} : \mathcal{C} \mapsto \mathbf{2}^{D^{\mathfrak{I}}}$ and $[\![\_]\!]^{\mathfrak{I}}_{\mathfrak{r}} : \mathcal{R} \mapsto \mathbf{2}^{D^{\mathfrak{I}} \times D^{\mathfrak{I}}}$ defined inductively as shown in Figure 12.

Depending on what concept and role constructs it has, a particular $\mathcal{DL}$ will have different expressivity and decidability results. In other words, expressivity of a given $\mathcal{DL}$ depends on whether its grammar has just a subset of, or all, the rules in Figure 11, or additional ones—*e.g.*, for expressing so-called *role-value maps*; *i.e.*, equality constraints among role composition. For example, the system obtained by keeping all the rules in Figure 11 except for the one for composite roles is called $\mathcal{ALCNR}$ and has nice properties such as decidability of knowledge-base satisfiability (Bucheit et al., 1993; Donini et al., 1996). The convention for naming such logics is to use a mnemonic letter

$$\begin{array}{rcl}
[\![\top]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & D^{\mathfrak{I}} \\
[\![\bot]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \emptyset \\
[\![\mathtt{Name}]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \mathfrak{C}_{\mathtt{Name}} \\
[\![\{\mathtt{Name}_1, \ldots, \mathtt{Name}_n\}]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \{\mathfrak{I}_{\mathtt{Name}_1}, \ldots, \mathfrak{I}_{\mathtt{Name}_n}\} \\
[\![C_1 \sqcap C_2]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & [\![C_1]\!]^{\mathfrak{I}}_{\mathfrak{c}} \cap [\![C_2]\!]^{\mathfrak{I}}_{\mathfrak{c}} \\
[\![C_1 \sqcup C_2]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & [\![C_1]\!]^{\mathfrak{I}}_{\mathfrak{c}} \cup [\![C_2]\!]^{\mathfrak{I}}_{\mathfrak{c}} \\
[\![\neg C_1]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & D^{\mathfrak{I}} \backslash [\![C_1]\!]^{\mathfrak{I}}_{\mathfrak{c}} \\
[\![\forall R.C]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \{x \in D^{\mathfrak{I}} \mid R[x] \subseteq [\![C]\!]^{\mathfrak{I}}_{\mathfrak{c}}\} \\
[\![\exists R.C]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \{x \in D^{\mathfrak{I}} \mid R[x] \cap [\![C]\!]^{\mathfrak{I}}_{\mathfrak{c}} \neq \emptyset\} \\
[\![\leq n.R]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \{x \in D^{\mathfrak{I}} \mid |R[x]| \leq n\} \\
[\![\geq n.R]\!]^{\mathfrak{I}}_{\mathfrak{c}} & \stackrel{\text{DEF}}{=\!=} & \{x \in D^{\mathfrak{I}} \mid |R[x]| \geq n\} \\
[\![\mathtt{Name}]\!]^{\mathfrak{I}}_{\mathfrak{r}} & \stackrel{\text{DEF}}{=\!=} & \mathfrak{R}_{\mathtt{Name}} \\
[\![R_1 \sqcap R_2]\!]^{\mathfrak{I}}_{\mathfrak{r}} & \stackrel{\text{DEF}}{=\!=} & [\![R_1]\!]^{\mathfrak{I}}_{\mathfrak{r}} \cap [\![R_2]\!]^{\mathfrak{I}}_{\mathfrak{r}} \\
[\![R_1 \bullet R_2]\!]^{\mathfrak{I}}_{\mathfrak{r}} & \stackrel{\text{DEF}}{=\!=} & [\![R_1]\!]^{\mathfrak{I}}_{\mathfrak{r}} \circ [\![R_2]\!]^{\mathfrak{I}}_{\mathfrak{r}}
\end{array}$$

Figure 12: **Semantics of common $\mathcal{DL}$ concept and role constructs**

encoding trick for their names: they all start with $\mathcal{AL}$ (for Attributive Logic) and we add a $\mathcal{U}$ if it can express for universal roles, an $\mathcal{E}$ for existential roles, a $\mathcal{C}$ concept complementation, an $\mathcal{N}$ for role number restrictions, and an $\mathcal{R}$ for role conjunction. Note that names are not unique per the logic they denote. For example, $\mathcal{ALC}$ and $\mathcal{ALEU}$ are different names for the same $\mathcal{DL}$. We prefer using the shorter names—*e.g.*, $\mathcal{ALC}$ rather than the equivalent $\mathcal{ALEU}$ (Baader and Nutt, 2003).

Note that from the set-theoretic semantics of $\mathcal{DL}$ in Figure 12, one can derive several syntactic congruence for the $\mathcal{DL}$ syntactic operators. Defining $S \simeq S'$ to mean $[\![S]\!] = [\![S']\!]$, it is easy to show that all the syntactic equivalences shown in Figure 13 hold. Therefore, we shall always implicitly consider all $\mathcal{DL}$ syntactic constructs modulo these syntactic congruences. Figure 14 shows examples of $\mathcal{DL}$ concept expressions and their meaning. Other concept-forming constructs may be defined in terms of more primitive ones. For example, it may sometimes come handy to use the following syntactic shorthand role-cardinality notations:

shorthand	$\leadsto$	**meaning**
$< n.R$	$\leadsto$	$\neg(\geq n.R)$,
$> n.R$	$\leadsto$	$\neg(\leq n.R)$,
$= n.R$	$\leadsto$	$(\leq n.R) \sqcap (\geq n.R)$,
$\neq n.R$	$\leadsto$	$(< n.R) \sqcup (> n.R)$.

$$
\begin{array}{rcl|rcl}
\neg\bot & \simeq & \top & C_1 \sqcup C_2 & \simeq & C_2 \sqcup C_1 \\
\neg\top & \simeq & \bot & C_1 \sqcap C_2 & \simeq & C_2 \sqcap C_1 \\
\{x\} \sqcup \{y\} & \simeq & \{x, y\} & \neg(C_1 \sqcup C_2) & \simeq & \neg C_1 \sqcap \neg C_2 \\
C \sqcup \bot & \simeq & C & \neg(C_1 \sqcap C_2) & \simeq & \neg C_1 \sqcup \neg C_2 \\
C \sqcup \top & \simeq & \top & (C_1 \sqcup C_2) \sqcup C_3 & \simeq & C_1 \sqcup (C_2 \sqcup C_3) \\
C \sqcap \top & \simeq & C & (C_1 \sqcap C_2) \sqcap C_3 & \simeq & C_1 \sqcap (C_2 \sqcap C_3) \\
C \sqcap \bot & \simeq & \bot & C_1 \sqcup (C_2 \sqcap C_3) & \simeq & (C_1 \sqcup C_2) \sqcap (C_1 \sqcup C_3) \\
C \sqcup C & \simeq & C & C_1 \sqcap (C_2 \sqcup C_3) & \simeq & (C_1 \sqcap C_2) \sqcup (C_1 \sqcap C_3) \\
C \sqcap C & \simeq & C & & &
\end{array}
$$

Figure 13: **Some $\mathcal{DL}$ syntactic congruences**

Example	**Meaning**
$\mathtt{Human} \sqcap \mathtt{Male}$	*the set of all things that are humans and males*
$\mathtt{Doctor} \sqcup \mathtt{Lawyer}$	*the set of all things that are doctors or lawyers*
$\neg\mathtt{Male}$	*the set of all things that are not males*
$\{\mathtt{john}, \mathtt{mary}\}$	*the set* $\{\mathfrak{I}_{\mathtt{john}}, \mathfrak{I}\mathtt{mary}\}$
$\forall\mathtt{hasChild}.\mathtt{Doctor}$	*the set of all things, all of whose children are doctors*
$\exists\mathtt{hasChild}.\mathtt{Laywer}$	*the set of all things, one of whose children is a lawyer*
$\leq 1.\mathtt{hasChild}$	*the set of all things that have at most one child*
$\geq 2.\mathtt{hasChild}$	*the set of all things that have at least two children*

Figure 14: **Examples of $\mathcal{DL}$ concept expressions**

3.2.3 $\mathcal{DL}$ KNOWLEDGE BASES

§**Terminological *vs.* assertional knowledge –** As mentioned before, $\mathcal{DL}$ knowledge (*i.e.*, facts and properties of the described universe), is represented by storing $\mathcal{DL}$ formulae in a *knowledge base*, which consists of two complementary parts: a *terminological* knowldege base (or *TBox*) and an *assertional* knowldege base (or *ABox*). Informally, the TBox defines the concept and role vocabulary and their hierarchical inheritance ordering, while the ABox is the extensional data populating the various sets and relations. This separation is akin to that of the intensional database , or IDB, from the extensional database, or EDB, in Datalog. In $\mathcal{DL}$, the TBox is the IDB and the ABox is the EDB.

Figure 15 shows examples of terminological axioms and their meaning.

$\mathcal{DL}$ **Syntax**	**Example**	**Meaning**
$C_1 \sqsubseteq C_2$	`Human` $\sqsubseteq$ `Biped` $\sqcap$ `Animal`	*humans are biped animals*
$C_1 \equiv C_2$	`Man` $\equiv$ `Human` $\sqcap$ `Male`	*men, and only men, are humans and males*
$C_1 \sqsubseteq \neg C_2$	`Male` $\sqsubseteq \neg$`Female`	*males are not females*
$\{x_1\} \equiv \{x_2\}$	$\{$`Tim Berners-Lee`$\} \equiv \{$`TBL`$\}$	`Tim Berners-Lee` *and* `TBL` *are the same individual*
$\{x_1\} \equiv \neg\{x_2\}$	$\{$`TBL`$\} \equiv \neg\{$`JesusChrist`$\}$	`TBL` *and* `JesusChrist` *are not the same individual*
$P_1 \sqsubseteq P_2$	`hasDaughter` $\sqsubseteq$ `hasChild`	*anything that has a daughter has also a child*
$P_1 \equiv P_2$	`cost` $\equiv$ `price`	*something has a cost iff it has also a price*
$P_1 \equiv P_2^-$	`hasChild` $\equiv$ `hasParent`$^-$	*something has a child iff that child has it as a parent*
$P^+ \sqsubseteq P$	`ancestor`$^+ \sqsubseteq$ `ancestor`	*one's ancestor's ancestor in also one's ancestor*
$\top \sqsubseteq \geq 1.P$	$\top \sqsubseteq \geq 1.$`hasName`	*everything must have at least one name*

Figure 15: **Examples of terminological axioms**

3.2.4 $\mathcal{DL}$ REASONING

Reasoning in $\mathcal{DL}$ is usually carried out based on Deductive Tableau methods (Schmidt-Schauß and Smolka, 1991; Bucheit et al., 1993; Donini et al., 1996; Horrocks and Patel-Schneider, 1998; Horrocks et al., 1999). For example, Figure 16 shows a system of constraint-propagation rules for the $\mathcal{ALCNR}$ $\mathcal{DL}$ dialect (Bucheit et al., 1993). We call this constraint system $\mathcal{C}$ for *Concept* constraint. These rules transform a set S of $\mathcal{DL}$-constraints each of the form either (1) $x : C$, (2) xRy, or (3) $x \not\doteq y$, where x, y are variables is some set $\mathcal{V}$, C is a $\mathcal{DL}$-concept expression and R is a $\mathcal{DL}$-role expression of the forms shown in Figure 11. Given such a set S, a variable x, and a role expression R, we use the notation $R_S[x] \overset{\text{DEF}}{=} \{y \mid xRy \in S\}$. For conjunctive roles, $(R \sqcap R')_S[x] \overset{\text{DEF}}{=} R_S[x] \cap R'_S[x]$, and for composite roles, $(R \bullet R')_S[x] \overset{\text{DEF}}{=} \{y \mid y \in R'_S[z] \text{ for some } z \in R_S[x]\}$.

The semantics of such constraints is as follows. Given a $\mathcal{DL}$-interpretation $\mathfrak{I}$ and an $\mathfrak{I}$-valuation $\alpha : \mathcal{V} \mapsto D^{\mathfrak{I}}$:

$$\begin{array}{lcll} \mathfrak{I},\alpha & \models & x : C & \textbf{iff} \quad \alpha(x) \in C^{\mathfrak{I}}; \\ \mathfrak{I},\alpha & \models & x \not\doteq y & \textbf{iff} \quad \alpha(x) \neq \alpha(y); \\ \mathfrak{I},\alpha & \models & xRy & \textbf{iff} \quad \langle \alpha(x), \alpha(y) \rangle \in R^{\mathfrak{I}}; \\ \mathfrak{I},\alpha & \models & S & \textbf{iff} \quad \mathfrak{I},\alpha \models \phi \text{ for all } \phi \in S. \end{array} \tag{18}$$

$(\mathcal{C}_\sqcap)$ **<u>CONJUNCTIVE CONCEPT</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (C_1 \sqcap C_2) \;\in\; S \\ \textbf{and} & \{x : C_1, x : C_2\} \not\subseteq S \end{array}\right] \qquad \frac{S}{S \cup \{x : C_1, x : C_2\}}$$

$(\mathcal{C}_\sqcup)$ **<u>DISJUNCTIVE CONCEPT</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (C_1 \sqcup C_2) \;\in\; S \\ \textbf{and} & x : C_i \;\notin\; S \;\; (i = 1, 2) \end{array}\right] \qquad \frac{S}{S \cup \{x : C_1\}}$$

$(\mathcal{C}_\forall)$ **<u>UNIVERSAL ROLE</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (\forall R.C) \;\in\; S \\ \textbf{and} & y \;\in\; R_S[x] \\ \textbf{and} & y : C \;\notin\; S \end{array}\right] \qquad \frac{S}{S \cup \{y : C\}}$$

$(\mathcal{C}_\exists)$ **<u>EXISTENTIAL ROLE</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (\exists R.C) \;\in\; S \;\; \textbf{s.t.} \;\; R \overset{\textbf{DEF}}{=\!=} \left(\bigsqcap_{i=1}^{m} R_i\right) \\ \textbf{and} & z : C \;\in\; S \Rightarrow z \;\notin\; R_S[x] \\ \textbf{and} & y \text{ is new} \end{array}\right] \qquad \frac{S}{S \cup \{xR_iy\}_{i=1}^{m} \cup \{y : C\}}$$

$(\mathcal{C}_\geq)$ **<u>MIN CARDINALITY</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (\geq n.R) \;\in\; S \;\; \textbf{s.t.} \;\; R \overset{\textbf{DEF}}{=\!=} \left(\bigsqcap_{i=1}^{m} R_i\right) \\ \textbf{and} & |R_S[x]| \neq n \\ \textbf{and} & y_i \text{ is new } (0 \leq i \leq n) \end{array}\right] \qquad \frac{S}{\begin{array}{l} S \;\cup\; \{xR_iy_j\}_{i,j=1,1}^{m,n} \\ \quad \cup\; \{y_i \neq y_j\}_{1 \leq i < j \leq n} \end{array}}$$

$(\mathcal{C}_\leq)$ **<u>MAX CARDINALITY</u>:**

$$\left[\begin{array}{ll} \textbf{if} & x : (\leq n.R) \;\in\; S \\ \textbf{and} & |R_S[x]| > n \\ \textbf{and} & y, z \;\in\; R_S[x] \\ \textbf{and} & y \neq z \;\notin\; S \end{array}\right] \qquad \frac{S}{S \cup S[y/z]}$$

Figure 16: **Some $\mathcal{DL}$-constraint propagation rules ($\mathcal{ALCNR}$)**

3.3 Examples

We now illustrate how the $\mathcal{OSF}$ and $\mathcal{DL}$ formalisms may be used with the $\mathcal{CLP}$ scheme as data description languages on which computational rules may be specified.

3.3.1 HERBRAND TERM UNIFICATION AS $\mathcal{OSF}$ CONSTRAINT SOLVING

We can recast unification of Herbrand and rational terms as an instance of $\mathcal{OSF}$ constraint solving. In other words, what was covered in Section 2.3.2 using variable/term substitutions can be generalized to the problem of normalizing an $\mathcal{OSF}$ constraint corresponding to the labelled-graph representation of terms. This has the nice consequence of simplifying the formal presentation as well since it does not need nor use cumbersome term substitutions and their compositions. Most importantly, the main benefit is a finer grain setting than offered by the Herbrand term unification algorithm of Figure 2. This allows more commutation flexibility between constraint solving and rule resolving. Keeping in mind the chemical metaphor, one may think that the more elementary the ions in the aqueous solution the easier they are to move and react.

Clearly, a first-order rational term in $\mathcal{T}_{\Sigma,\mathcal{V}}$ can be viewed as a particular ψ-term. For this, it suffices to take $\mathcal{S} = \Sigma \cup \{\top, \bot\}$ and $\mathcal{F} = \mathbb{N}^*$. Namely, function symbols in $\Sigma = \bigcup_{n>0} \Sigma_n$ denote singleton sorts (*i.e.*, they are mutually incomparable except that $\forall f \in \Sigma, \bot < f < \top$), and numbers as features. Thus, the term $f(t_1, \ldots, t_n)$ is the ψ-term $f(1 \Rightarrow t_1, \ldots, n \Rightarrow t_n)$. The features here are simply *argument positions* and are interpreted in the $\mathcal{OSF}$ formalism as *projection functions*. Additional axioms are needed to enforce arity constraints. Namely:

$$\textbf{ARITY}(\top) = \emptyset \tag{19}$$

$$\textbf{ARITY}(\bot) = \{i \in \mathbb{N}^* \mid i \leq \max\{n > 0 \mid \Sigma_n \neq \emptyset\}\} \tag{20}$$

$$\forall f \in \Sigma_n : \quad \textbf{ARITY}(f) = \{1, \ldots, n\} \tag{21}$$

$$\forall i \in \mathcal{F} : \quad \textbf{DOM}(i) = \bigcup_{i \leq n} \Sigma_n \tag{22}$$

$$\forall i \in \mathcal{F}, \forall f \in \Sigma : \quad \textbf{RAN}_f(i) = \begin{cases} \top & \textbf{if } f \in \textbf{DOM}(i), \\ \bot & \textbf{otherwise}. \end{cases} \tag{23}$$

Condition (19) states that $\top$ has empty arity. This corresponds to the fact that logical variables may appear only as term leaves. Condition (20) states that $\bot$ has the maximal arity of all symbols. Condition (21) declares the arity for each function symbol. Condition (22) declares the domains for each argument position—namely, the set of symbols that have at least that many arguments. Condition (23) enforces the domains and ranges declared in the signature for function symbols according to their arity constraints.

For sorted algebras, the sort signature $\mathcal{S}$ may also contain non-minimal sorts above the singleton-denoting function symbols of Σ. Thus, multi- or order-sorted versions of free term algebra $\mathcal{T}_{\Sigma,\mathcal{V}}$ are readily expressible in the $\mathcal{OSF}$ formalism by making Condition (23) involve non-singleton sorts other than $\top$ as the range of projection features. With these signature constraints, the *"Partial Feature"* rule of Figure 10 combined with the basic $\mathcal{OSF}$ rules of Figure 4 will make unification of (rational) Herbrand terms behave as expected. For *non-rational* terms, one must also perform an *"occurs-check"* test in the *"Variable Elimination"* rule.

3.3.2 RULES OVER OBJECTS

Rules over typed attributed objects with class inheritance such as typically used in popular object-oriented programming languages such as *Java*, *C#*, or *C/C++*, fit perfectly the $\mathcal{OSF}$ formalism with sort definitions such as used in (Aït-Kaci and Nasr, 1986) where a basic integration of $\mathcal{LP}$ with feature-term inheritance is described. LogIn was, to our knowledge, the first proposal motivated by using order-sorted feature term inheritance as constraints in logic programming. The motivation behind its design was that when *"types"* (or *"sorts"*) form a lower semi-lattice (*i.e.*, a partial order with *greatest lower bounds*—**GLB**'s), unification of labelled graphs modulo this order is the key to achieving a better focus on relevant goals by pruning out relations by set-denoting constraints. Indeed, LogIn is easily characterized as a $\mathcal{CLP}$ language based on the $\mathcal{OSF}$ constraint system.

Clearly, this paradigm is directly amenable for specifying rules over *Java*-style object-oriented data models. Take for example the simple multiple-inheritance class-interface hierarchy of Figure 17. Each class interface name corresponds to a sort. The declarations of Figure 17 define the

```
interface AdultPerson {
   Name id;
   Date dob;
   int age;
   String ssn;
}
interface Employee extends AdultPerson {
   Title position;
   String institution;
   Employee supervisor;
   int salary;
}
interface MarriedPerson extends AdultPerson {
   MarriedPerson spouse;
}
interface MarriedEmployee extends Employee, MarriedPerson {
}
interface RichEmployee extends Employee {
}
```

Figure 17: **Example of a *Java* multiple-inheritance class-interface hierarchy**

sort partial order given in Figure 18.

It is not difficult to see that $\mathcal{OSF}$ term unification or entailment can thus accommodate rules using unification or pattern-matching over such objects.

This approach, exemplified by LogIn (Aït-Kaci and Nasr, 1986), has had several descendants, most notably LIFE (Aït-Kaci, 1993; Aït-Kaci et al., 1994a), but it also inspired many natural-language processing ($\mathcal{NLP}$) formalisms based on pure $\mathcal{OSF}$ term constraints such as Nora (Fischer, 1993), or augmented with relational dependencies such as STUF (Dörre and Seiffert, 1991), itself fully expressed in the spirit of the Höhfeld-Smolka $\mathcal{CLP}$ scheme. One may indeed see the LogIn family of languages (which includes LIFE) as $\mathcal{CLP}(\mathcal{O})$ languages. Strictly speaking, the

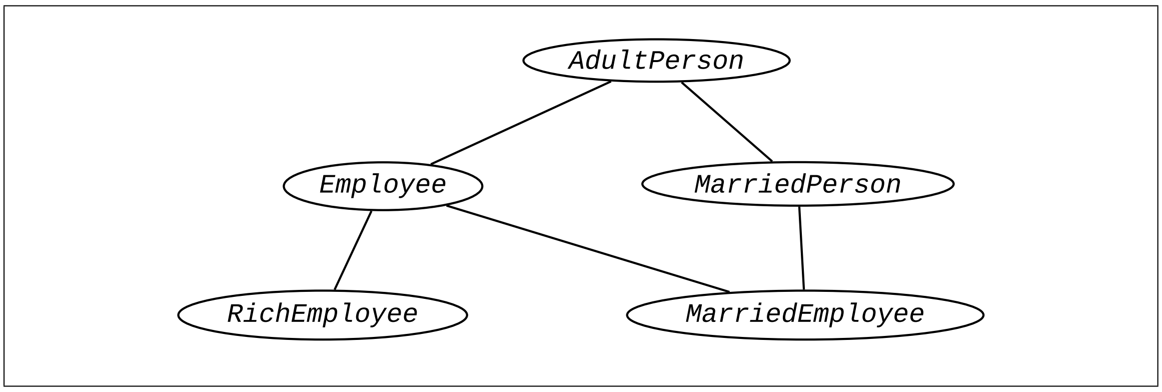

Figure 18: **Partial sort order for the *Java* hierarchy of Figure 17**

LogIn language as described in (Aït-Kaci and Nasr, 1986) only includes the rules of Figures 4 and 7, as well as the rules for sort definitions described in (Aït-Kaci et al., 1997), although without variables in sort definitions—which makes $\mathcal{OSF}$ unification modulo sort definitions decidable. LIFE (Aït-Kaci, 1993) extends LogIn with interpreted functions over $\mathcal{OSF}$ terms, as well as with sort definitions with variables and relational conditions—See Section 3.3.4.

Clearly, non-logical rules systems such as those based on condition/action production rules typically used in experts systems using object pattern-matching (as opposed to unification) can also be expressed using $\mathcal{OSF}$constraint entailment (as opposed to constraint conjunction).

3.3.3 HORN RULES OVER DESCRIPTION LOGIC CONCEPTS

Definite clauses over constraint systems implementing the semantics of any dialect in the family $\mathcal{DL}$ languages have been proposed. Examples of such systems are CARIN (Levy and Rousset, 1998), and $\mathcal{AL}$-log (Donini et al., 1998). Both are logic-programming languages that exploit the description logic $\mathcal{ALCNR}$ (Bucheit et al., 1993; Donini et al., 1996). CARIN is less restrictive than $\mathcal{AL}$-log. In fact, both works fit fully the approach we preconize here (indeed, CARIN $\in \mathcal{CLP}(\mathcal{C})$). However, although they present an interesting marriage of $\mathcal{DL}$ with definite clauses, neither make explicit the link with the $\mathcal{CLP}$ scheme nor, therefore, do they stress the free model-theoretic semantics thereby inherited.

More seriously, CARIN's designers seem to have an incorrect understanding of the $\mathcal{OSF}$ formalism and how it relates to $\mathcal{DL}$s. In particular, in (Levy and Rousset, 1996), one can read:

> *"ψ-terms [...] differ from description logics in several significant ways. [...] they are more limited [...] since they can only allow functional roles [...] For example, number restrictions and existential statements that are standard in description logics are not expressible in ψ-terms."*

However, these (unsubstantiated) statements are patently incorrect on all counts as clearly shown in Section 4.2.

3.3.4 ORDERED SORTS AS PROOF MEMOIZING

The basic $\mathcal{OSF}$-unification rules of Figure 4 may be viewed as proof rules for proving monadic predicates (*i.e.*, $X : s$ is consistent iff $s(X)$ is true), functional dependencies (*i.e.*, $X.f \doteq Y$ is consistent iff $Y = f(X)$), and equality (*i.e.*, $X \doteq Y$ is consistent iff $X = Y$). It may thus be argued that any logic reasoner may do the job of providing a satisfactory operational semantics to such constraints. This is not so, however, since $\mathcal{OSF}$-unification proves sort constraints by reducing them monotonically w.r.t. the sort ordering. This means that when $X : s$ has been proven, the proof of $s(X)$ is recorded as *the sort s itself!*. Indeed, if further down a proof, it is again needed to prove $X : s$, the process simply remembers it simply by looking at X's binding. This, however, would not be the case in a basic relational setting such as Prolog where having proven goal $s(X)$ is not remembered in any manner.

Note that a subtle but wonderful consequence of this proof memoizing power of $\mathcal{OSF}$-style unification arises when relational rules are also used in the body of sort definition. Indeed, clearly the integration Rules/Ontology may be used as well symmetrically as Ontology/Rules. In other words, while ontological constraints may be used for inference and computation by rules, rules may in turn be used to enhance the expressive power of ontologies by allowing relational conditions to sort definitions as done in LIFE (Aït-Kaci, 1993).

Let us briefly illutrate on an example how this works, and why this is invaluable. Let us consider again the sort hierarchy of Figure 18. LIFE also uses interpreted functions (Aït-Kaci et al., 1994a; Aït-Kaci and Podelski, 1994). For example, we assume here that the function $\texttt{ageInYears}$ in Figure 19 is a function that takes an $\texttt{AdultPerson}$ object and returns its age in number of years from its $\texttt{dob}$ feature and the current date—*i.e.*, it is of type $\texttt{AdultPerson} \mapsto \texttt{int}$ and could be defined as, *e.g.*, $\texttt{ageInYears}(A : \texttt{AdultPerson}) \rightarrow \texttt{currentYear} - A.\texttt{dob}.\texttt{year}$. This sort hierarchy defines recursive sorts and attaches relational conditions to each. The inheritance scheme means that a sort's features, equations thereof, as well as relational constraints are passed down to subsorts. If we label every sort definition's conditions with the maximal sort they restrict by using a notation such as $\texttt{sort}\#(\texttt{condition})$, then it is a simple matter to implement relational-proof memoizing as follows. In LIFE, equations involving interpreted functions, such as $+$, $*$, $\leq$, $\geq$, that miss some information are suspended until such information materializes—*i.e.*, incomplete function calls *residuate*. In particular, the feature-projection function '.' applied to a non-feature expression residuates on its second argument (*e.g.*, Y in the constraint $X.Y = Z$, where X, X, and Z are variables). That is, it waits until its second argument evaluates to a constant that denotes a feature name. For example, the constraint $X.Y = Z$ will wait for the feature argument until Y gets bound to, say, $\texttt{foo}$, at which point the constraint is "awakened" and X gets then bound to $\top(\texttt{foo} \Rightarrow Z)$. Obviously, this works also when projecting on any functional expression that eventually evaluates to a feature name.

Let's say that we prove the following sequence of $\mathcal{OSF}$ constraints:

1. $X : \texttt{adultPerson}(\texttt{age} \Rightarrow 25)$,
2. $X : \texttt{employee}$,
3. $X : \texttt{marriedPerson}(\texttt{spouse} \Rightarrow Y)$.

First, we prove $X : \texttt{adultPerson}$ and establish the condition $\texttt{adultPerson}\#(X.age \geq 18)$. Note that, if this succeeds, X gets bound to a term with root sort $\texttt{adultPerson}$ where all the features leading to a variable in the constraint are materialized. As we proceed to proving $X : \texttt{employee}$, since we have already proven that the $\texttt{age}$ was correct, there is no need to do so again. This is achieved by proving only the conditions imposed on $\texttt{employee}$ that are tagged by a subsort of $\texttt{employee}$; namely, $\texttt{employee}\#(\texttt{higherRank}(E.\texttt{position}, P))$ and

```
:: P : adultPerson      ( id ⇒ name
                        , dob ⇒ date
                        , age ⇒ A : int
                        , ssn ⇒ string
                        )
   | A = ageInYears(P), A ≥ 18.
-------------------------------------------------------------

employee <: adultPerson.
:: employee             ( position ⇒ T : title
                        , institution ⇒ string
                        , supervisor ⇒ E : employee
                        , salary ⇒ S : int
                        )
   | higherRank(E.position,T) , E.salary ≥ S.
-------------------------------------------------------------

marriedPerson <: adultPerson.
:: M : marriedPerson    ( spouse ⇒ P : marriedPerson   )
   | P.spouse = M.
-------------------------------------------------------------

marriedEmployee <: employee.
marriedEmployee <: marriedPerson.

-------------------------------------------------------------

richEmployee <: employee.
:: R : richEmployee     ( institution ⇒ I
                        , salary ⇒ S
                        )
   | stockValue(I) = V , hasShares(R,I,N) , S + N ∗ V ≥ 200000.
```

Figure 19: **A relationally constrained LIFE sort hierarchy for the classes in Figure 17**

$\texttt{employee}\#(E.\texttt{salary} \geq S)$. Next, we proceed to proving $X : \texttt{marriedPerson}(\texttt{spouse} \Rightarrow Y)$. However, we already proved $X : \texttt{employee}$. Thus, by sort intersection, we must prove $X : \texttt{marriedEmployee}(\texttt{spouse} \Rightarrow Y)$. Hence, only $\texttt{marriedPerson}\#(Y.\texttt{spouse} = X)$ needs to be established. All other inherited conditions having a sort label greater than $\texttt{married-Employee}$—such as $\texttt{employee}$—can therefore be safely ignored.

Clearly, this method allows *never to prove more than once any relational condition* restricting a sort definition. This property seems to have been overlooked by many researchers who consider that the $\mathcal{OSF}$ formalism is, like the $\mathcal{DL}$ formalism, just a peculiar notational variant of a subset of $\mathcal{FOL}$.

4. Relation between $\mathcal{OSF}$ and $\mathcal{DL}$ formalisms

Description Logic ($\mathcal{DL}$) and Order-Sorted Feature ($\mathcal{OSF}$) logic are two mathematical formalisms that possess proof-theories based on a constraint formalism. Both are direct descendants of Ron Brachman's original ideas (Brachman, 1977). This inheritance goes through my own early work formalizing Brachman's ideas (Aït-Kaci, 1984), which in turn inspired the work of Gert Smolka, who pioneered the use of constraints *both* for the $\mathcal{DL}$ (Schmidt-Schauß and Smolka, 1991) *and* $\mathcal{OSF}$ (Smolka, 1988) formalisms. While the $\mathcal{DL}$ approach has become the mainstream of research on the semantic web, the lesser known $\mathcal{OSF}$ formalisms have evolved out of Unification Theory (Schmidt-Schauß and Siekmann, 1988), and been used in constraint logic programming and computational linguistics (Dörre and Rounds, 1990; Emele and Zajac, 1990; Dörre and Seiffert, 1991; Emele, June 1991; Zajac, 1991; Emele and Zajac, 1992; Smolka, 1992; Zajac, 1992; Carpenter, 1992; Aït-Kaci, 1993; Fischer, 1993; Aït-Kaci and Podelski, 1993; Aït-Kaci et al., 1994a; Smolka and Treinen, 1994; Treinen, 1997; Müller et al., 2000, 2001).

Both formalisms were introduced for describing attributed typed objects. Thus, $\mathcal{OSF}$ and $\mathcal{DL}$ have several common, as well as distinguishing, aspects. Thanks to both formalisms using the common language of $\mathcal{FOL}$ for expressing semantics, they may thus be easily compared—see, for example, (Nebel and Smolka, 1990, 1991). We here brush on some essential points of comparison and contrast.

4.1 Common aspects

$\mathcal{DL}$ reasoning is generally carried out using (variations on) Deductive Tableau methods (Manna and Waldinger, 1991). This is also the case of the constraint propagation rules of Figure 16, which simply mimick a Deductive Tableau decision procedure (Donini et al., 1996). $\mathcal{OSF}$ reasoning is done by the $\mathcal{OSF}$-constraint normalization rules of Figures 4 and 10, which implement a logic of sorted-feature equality.

§**Object descriptions –** Both the $\mathcal{DL}$ and $\mathcal{OSF}$ formalisms describe typed attributed objects. In each, objects are data structures described by combining set-denoting concepts and relation-denoting roles.

§**Logic-based semantics –** Both $\mathcal{DL}$ and $\mathcal{OSF}$ logic are syntatic formalisms expressing meaning using conventional logic styles. In other words, both formalisms take their meaning in a common universal language—*viz.*, (elementary) Set Theory. This is good since it eases understanding each formalism in relation to the other thanks to their denotations in the common language.

§**Proof-theoretic semantics –** Both $\mathcal{DL}$ and $\mathcal{OSF}$ logics have their corresponding proof theory. Indeed, since both formalisms are syntactic variants of fragments of $\mathcal{FOL}$, proving theorems in each can always rely on $\mathcal{FOL}$ mechanized theorem proving.

§**Constraint-based formalisms –** Even further, both $\mathcal{DL}$ and $\mathcal{OSF}$ logic are operationalized using a constraint-based decision procedure. As we have expounded, this makes both paradigms amenable to being manipulated by rule-based systems such as based on $\mathcal{CLP}$, rewrite rules, or production rules.

§**Concept definitions –** Both $\mathcal{DL}$ and $\mathcal{OSF}$ provide a means for defining concepts in terms of other concepts. This enables a rich framework for expressing recursive data structures.

4.2 Distinguishing aspects

There are also aspects in each that distinguish the $\mathcal{DL}$ and $\mathcal{OSF}$ formalisms apart. However, several of these distinguishing features are in fact cosmetic—*i.e.*, are simply equivalent notation for the same meaning. Remaining non-cosmetic differences are related to the nature of the deductive processes enabled out by each formalism.

§**Functional features *vs.* relational roles –** The $\mathcal{OSF}$ formalism uses *functions* to denote attributes while the $\mathcal{DL}$ formalism uses *binary relations* for the same purpose. Many have argued that this difference is fundamental and restricts the expressivity of $\mathcal{OSF}$ *vs.* $\mathcal{DL}$. This, however, is only a cosmetic difference as we have already explained. First of all, a function $f : A \mapsto B$ *is* a binary relation since $f \in A \times B$. It a *functional* relation because it obeys the axiom of functionality; namely,

$$\langle a, b\rangle \in f \;\&\; \langle a, b'\rangle \in f \;\Rightarrow\; b = b'. \tag{24}$$

In other words, a function is a binary relation that associates at most one range element to any domain element. This axiom is fundamental as it is is used in basic $\mathcal{OSF}$ unification *"Feature Functionality"* shown in Figure 4. Indeed, the correctness of this rule relies on the semantics of features as functions, not as relations.

However, a relation $R \in A \times B$ is equivalent to either of a pair of set-denoting functions—*viz.*., either the function $R[\_] : A \mapsto \mathbf{2}^B$, returning the R-*object* (or R-*image*) set $R[x] \subseteq B$ of an element $x \in A$; or, dually, the function $R^{-1}[\_] : B \mapsto \mathbf{2}^A$, returning the R-*subject* (or R-*antecedent*) set $R^{-1}[y] \subseteq A$ of an element $y \in B$—see Equations (17). Indeed, the following statements (1)–(3) are equivalent:

$$\forall \langle x, y\rangle \in A \times B, \quad (1)\;\; \langle x, y\rangle \in R \quad \textbf{iff} \quad (2)\;\; y \in R[x] \quad \textbf{iff} \quad (3)\;\; x \in R^{-1}[y].$$

Therefore, it is a simple matter for the $\mathcal{OSF}$ formalism to express relational attributes (or roles) with features taking values as sets. This is trivially done as a special case of the *"Value Aggregation"* $\mathcal{OSF}$ unification rule shown in Figure 10, using a set data constructor—*i.e.*, a commutative idempotent monoid.

§**Sets *vs.* individuals –** Because the $\mathcal{OSF}$ formalism has only set-denoting sorts, it is often misconstrued as unable to deal with individual elements of these sets. However, as explained in Section 3.1.6, this is again an innocuous cosmetic difference since elements are simply assimilated to singleton-denoting sorts.

§**No number restrictions *vs.* Number restrictions –** Strictly speaking, the $\mathcal{OSF}$ formalism has no special constructs for number restrictions as they exist in $\mathcal{DL}$. Now, this does not mean that it lacks the power to enforce such constraints. Before we show how this may be done, however, it important to realize that it may not always be a good idea to use the $\mathcal{DL}$ approach to do so.

Indeed, as can be seen in Figure 16, the *"Min Cardinality"* rule $(\mathcal{C}_{\leq})$ will introduce $n(n-1)/2$ new disequality constraints for each such constraint of cardinality n. Clearly, this is a source of gross inefficiency a n increases. Similarly, the *"Existential Role"* rule $(\mathcal{C}_{\exists})$ will systematically introduce a new variable for a role, *even when this role is never accessed!* It does so because, it materializes the full extent of role value sets. In other words, $\mathcal{C}$ constraint-propagation rules flesh out complete skeletons for attributed data structures whether or not the actual attribute values are needed.

By contrast, it is simple and efficient to accommodate cardinality constraints in the $\mathcal{OSF}$ calculus with value aggregation using a set constructor (*i.e.*, an idempotent commutative monoid $M = \langle \star, \mathbf{1}_{\star} \rangle$), and a function $\mathbf{CARD} : M \mapsto \mathbb{N}$ that returns the number of elements in a set. Then, imposing a role cardinality constraint for a role r in a feature term $t = X : s(r \Rightarrow S = \{e_1, \ldots, e_n\} : m)$, where sort m denotes M's domain, is achieved by the constraint $\varphi(t)$ & $\mathbf{CARD}(S) \leq n$—or $\varphi(t)$ & $\mathbf{CARD}(S) \geq n$. If the set contains variables, these constraints will residuate as needed pending the complete evaluation of the function **CARD**. However, as soon as enough non-variable elements have materialized in the set that enable the decision, the constraint will be duly enforced. Clearly, this "lazy" approach saves the time and space wasted by $\mathcal{DL}$-propagation rules, while fully enforcing the needed cardinalities.

Incidentally, note also that this principle allows not only min and max cardinality, but any constraints on a set, whether cardinality or otherwise. Importantly, this foregoing method works not only for sets, but can be used with arbitrary aggregations using other monoids.

§**Greatest fix point *vs.* least fix point –** It is well known that unfolding recursive definitions of all kinds (be it function, relation, or sort) is precisely formalized as computing a fix point in some information-theoretic lattice. Recall that a function between two ordered sets $f : A, \leq \mapsto A', \leq'$ is monotone iff and only if $\forall x, y \in A,\ x \leq y \Rightarrow f(x) \leq' f(y)$. Recall also that, given a complete lattice $\mathfrak{L} \overset{\text{DEF}}{=} \langle D^{\mathfrak{L}}, \sqsubseteq^{\mathfrak{L}}, \sqcap^{\mathfrak{L}}, \sqcup^{\mathfrak{L}}, \top^{\mathfrak{L}}, \bot^{\mathfrak{L}} \rangle$ and a monotone function $\mathcal{F} : D^{\mathfrak{L}} \mapsto D^{\mathfrak{L}}$, Tarski's fix-point theorem states that the set $\mathbf{FP}(\mathcal{F}) \overset{\text{DEF}}{=} \{x \in D^{\mathfrak{L}} \mid \mathcal{F}(x) = x\}$ of fix points of $\mathcal{F}$ is itself a complete sublattice of $\mathfrak{L}$ (Birkhoff, 1979). Moreover, its bottom element is called $\mathcal{F}$'s *least fix point* (LFP), written $\mathcal{F}^{\uparrow}$, defined by Equation (25):

$$\mathcal{F}^{\uparrow} \overset{\text{DEF}}{=} \bigsqcup_{n \in \mathbb{N}}^{\mathfrak{L}} \mathcal{F}^n(\bot^{\mathfrak{L}}) \tag{25}$$

and its top element is called $\mathcal{F}$'s *greatest fix point* (GFP), written $\mathcal{F}^{\downarrow}$, defined by Equation (26):

$$\mathcal{F}^{\downarrow} \overset{\text{DEF}}{=} \bigsqcap_{n \in \mathbb{N}}^{\mathfrak{L}} \mathcal{F}^n(\top^{\mathfrak{L}}) \tag{26}$$

where:

$$\mathcal{F}^n(x) = \begin{cases} x & \textbf{if } n = 0, \\ \mathcal{F}(\mathcal{F}^{n-1}(x)) & \textbf{otherwise}. \end{cases}$$

Informally, $\mathcal{F}^{\uparrow}$ is the *upward* iterative limit of $\mathcal{F}$ starting from the least element in $D^{\mathfrak{L}}$, while $\mathcal{F}^{\downarrow}$ is its *downward* iterative limit starting from the greatest element in $D^{\mathfrak{L}}$. One can easily show that $\mathcal{F}(\mathcal{F}^{\uparrow}) = \mathcal{F}^{\uparrow}$ [resp., $\mathcal{F}(\mathcal{F}^{\downarrow}) = \mathcal{F}^{\downarrow}$], and that no element of $D^{\mathfrak{L}}$ lesser than $\mathcal{F}^{\uparrow}$ [resp., greater than $\mathcal{F}^{\downarrow}$] is a fix point of $\mathcal{F}$.

One may wonder when one, or the other, kind of fix point captures the semantics intended for a set of recursive definitions. Intuitively, LFP semantics is appropriate when inference proceeds by deriving *necessary consequences* from facts that hold true, and GFP semantics is appropriate when inference proceeds by deriving *sufficient conditions* for facts to hold true. One might also say that LFP is *deductive* since it moves from premiss to consequent, and that GFP is *abductive* since it moves from consequent to premiss. Therefore, LFP computation can model only well-founded (*i.e.*, terminating) recursion, while GFP computation can also model non well-founded (*i.e.*, not necessarily terminating) recursion. Hence, typically, LFP computation is naturally described as a *bottom-up* process, while GFP computation is naturally described as a *top-down* process.

An example of LFP semantics is given by the semantics of $\mathcal{CLP}$ relations defined in Equations 2, while an example of GFP semantics is given by the non-deterministic $\mathcal{CLP}$ resolution process described in Equations 3–5. Indeed, the former proceeds bottom-up, while the latter goes top-down. Note that, in this case, the two fix points coincide. Such is not necessarily the case in general.

Another example of GFP semantics is given by the unification algorithm of Figure 2. Indeed, unification transforms a set of equations into an equivalent one using sufficient conditions by processing the terms top-down from roots to leaves. The problem posed is to find sufficient conditions for a term equation to hold on the constituents (*i.e.*, the subterms) of both sides of the equation. For first-order terms, this process converges to either failure or producing a most general sufficient condition in the form of a variable substitution, or equation set in solved form (the MGU). Similarly, the $\mathcal{OSF}$-constraint normalization rules of Figures 4, 7, 9, and 10 also form an example of converging GFP computation for the same reasons. Yet another example of GFP computation where the process may diverge is the lazy recursive sort definition unfolding described in (Aït-Kaci et al., 1997).

On the other hand, constraint-propagation rules based on Deductive Tableau methods such as used in (Schmidt-Schauß and Smolka, 1991) or shown in Figure 16 are LFP computations. Indeed, they proceed bottom-up by building larger and larger constraint sets by completing them with additional (and often redundant) constraints. In short, $\mathcal{OSF}$-constraint normalization follows a reductive semantics (it eliminates constraints) while $\mathcal{DL}$-constraint propagation follows an inflationary semantics (it introduces constraints). As a result, $\mathcal{DL}$'s tableau-style reasoning method is expansive—therefore, expensive in time and space. One can easily see this simply by realizing that each rule in Figure 16 builds a larger set S as it keeps adding more constraints and more variables to S. Only the *"Max Cardinality"* rule $(\mathcal{C}_{\leq})$ may reduce the size of S to enforce upper limits on a concept's extent's size by merging two variables. Finally, it requires that the constraint-solving process be decidable.

By contrast, the $\mathcal{OSF}$ labelled-graph unification-style reasoning method is more efficient both in time and space. Moreover, it can accommodate semi-decidable—*i.e.*, undecidable, though recursively enumerable—constraint-solving. Indeed, no rule in Figures 4, 7, 9, and 10 ever introduces a new variable. Moreover, all the rules in Figure 4 as well as the rule 10, except for the *"Partial Feature"* rule, all eliminate constraints. Even this latter rule introduces no more constraints than the number of features in the whole constraint. The rules in Figures 7 and 9 may replace some constraints with more constraints, but the introduced constraints are all more restrictive than those eliminated.

§**Coinduction *vs.* induction –** Remarkably, the interesting duality between least and greatest fixpoint computations is in fact equivalent to another fundamental one; namely, *induction vs. coinduction* in computation and logic, as nicely explained in (Sangiorgi, 2004). Indeed, while induction allows to derive a whole entity from its constituents, coinduction allows to derive the constituents from the whole. Thus, least fix-point computation is induction, while greatest fix-point computation is coinduction. Indeed, coinduction is invaluable for reasoning about non well-founded computations such as those carried out on potentially infinite data structures (Aczel, 1988), or (possibly infinite) process bisimulation (Baeten and Weijland, 1990).

This is a fundamental difference between $\mathcal{DL}$ and $\mathcal{OSF}$ formalisms: $\mathcal{DL}$ reasoning proceeds by actually building a model's domain verifying a TBox, while $\mathcal{OSF}$ reasoning proceeds by eliminating impossible values from the domains. Interestingly, this was already surmised in (Schmidt-Schauß and Smolka, 1991) where the authors state:

> *"[. . .] approaches using feature terms as constraints [. . .] use a lazy classification and can thus tolerate undecidable subproblems by postponing the decision until further information is available. [. . . these] approaches are restricted to feature terms; however, an extension to KL-ONE-like concept terms appears possible."*

Indeed, the extended $\mathcal{OSF}$ formalism we have overviewed in this article is a means to achieve precisely this.

5. Conclusion

We have shown how constraint logic programming offers an invaluable abstraction mechanism for "integrating" correctly, seamlessly, and—to boot!—operationally, rule-based programming (*e.g.*, definite-clause logic programming) with data description logics. Seen as a formal constraint system, the data model is thus abstracted from the rule model. In order to demonstrate how the $\mathcal{CLP}$ scheme hinges on the fact that the rule dimension and the data model dimension are *orthogonal*, we have illustrated this paradigm by formulating four $\mathcal{LP}$ languages, namely Datalog, Prolog, LIFE, and CARIN, as members of the $\mathcal{CLP}$ language family $\mathcal{CLP}(\mathcal{A})$, where $\mathcal{A} = \mathcal{D}, \mathcal{H}, \mathcal{O}, \mathcal{C}$. Indeed, one may conjugate any formal rule-based system (*i.e.*, not only Horn-based) with any data model as long as the latter may be expressed using constraints. This independence property leads to a clear separation of concerns and great benefits either regarding correctness (due to clean formal semantics of *constraint entailment as pattern matching* and *constraint conjunction as unification*), or implementation (due to efficient dedicated constraint-solving algorithms). We have also reviewed two well-known data description formalisms based on constraints, Order-Sorted Feature Logic and Description Logic, explicating how they work and how they are formally related.

Clearly, constraints are the right medium for expressing symbolic or numeric data, or mixtures of both. For example, as shown by the pioneering work of the late Paris Kanellakis on *Constraint Databases*, one may use relational rules acting on non-symbolic, or hybrid, data as in Geographical Information System (GIS) where cartographic data may be described by *geometric constraints* in the form, *e.g.*, of linear inequalities delineating map areas as convex polygons (Kanellakis and Goldin, 1994; Brodsky, 1996), for which Mathematical Programming techniques used in Operations Research may be used. In addition, the algebraic properties of constraint are appropriate for making different constraint systems cooperate by helping one another when each in isolation may not have enough information to proceed to a solution (Aït-Kaci et al., 2007, 2006).

Easing rule interoperability is yet another substantial benefit of the "data as constraint" approach. Indeed, constraints are the right level of abstraction for rule interchange because they allow *approximation*. Approximation is important for exchange as one may still wish to exchange rules at some level of *abstraction*. If data is assimilated to constraints then abstraction is possible simply by relaxing some constraints describing the data over which the rules are defined. This is precisely the method used in (Aït-Kaci et al., 2007, 2006) for the verification of production rules by abstract interpretation over constraints.

Finally, the most obvious benefit of seeing data description as constraints is that it *simplifies* things at both the theoretical and practical level. It is a rare happening in information science that such be the case for it not to be of some welcome convenience. The most immediate is the perspicuous expression of rule-based computation and inference over various data models. For this reason, it may be of importance for facilitating some of the advertized objectives of the semantic-web effort.

We hope that this article will spur the reader's interest in pursuing some of the ideas we have discussed.

Acknowledgments

The author wishes to express his gratitude both to Andreas Podelski and Gert Smolka for many enlightening discussions over the years on several issues touched upon in this work. In adddition, Gert is specially thanked for providing copies of no longer available papers during the preparation of this article. I also thank Jean-Louis Ardoint and Bruno Berstel for kindly proofreading earlier drafts, catching umpteen typos, glitches, and various other inanities. Thanks also to Lucas Bordeaux, Barry O'Sullivan, and Pascal Van Hentenryck, the organizers of CP 2006 and editors of this special issue of CPL for inviting this communication. Finally, and perhaps most importantly, I am indebted to Ron Brachman for getting me interested in his fascinating ideas and nice diagrams at a time when he called them "SI-Nets" (Brachman, 1977).

Appendix

Appendix A. Herbrand terms and substitutions

Let $\{\Sigma_n\}_{n\geq 0}$ be an indexed family of mutually disjoint sets of (function) symbols of arity n. Let $\Sigma = \bigcup_{n\geq 0} \Sigma_n$ be the set of all function symbols.If we assume $\Sigma_0 \neq \emptyset$ ground Herbrand terms will be *finite* trees—*i.e.*, wherein all path are finite and lead from the root to the leaves. These structures are called *inductive* as they embody computation from the leaves to the root. On the other hand, *rational terms* (or regular graphs) do not have these restrictions: paths in a rational term may be of infinite length, although the number of its subterms is itself finite. Prolog III's rational terms are an example Colmerauer (1990). Other examples are *coinductive* structures used in so-called non-strict programming languages—*e.g.*, lazy lists and trees.

Let $\mathcal{T}_\Sigma$ be the set of *ground terms* defined as the smallest set such that:

- if $a \in \Sigma_0$ then $a \in \mathcal{T}_\Sigma$;
- if $f \in \Sigma_n$ and $t_i \in \mathcal{T}_\Sigma, (1 \leq i \leq n)$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma$.

Let $\mathcal{V}$ be a countably infinite set of *variables*. By convention, variables will be capitalized not to confuse them with constants in Σ_0.

The set of *first-order (Herbrand) terms* is written $\mathcal{T}_{\Sigma,\mathcal{V}}$ and is defined as the smallest set such that:

- if $X \in \mathcal{V}$ then $X \in \mathcal{T}_{\Sigma,\mathcal{V}}$;
- if $a \in \Sigma_0$ then $a \in \mathcal{T}_{\Sigma,\mathcal{V}}$;
- if $f \in \Sigma_n$ and $t_i \in \mathcal{T}_{\Sigma,\mathcal{V}}, (1 \leq i \leq n)$, then $f(t_1, \ldots, t_n) \in \mathcal{T}_{\Sigma,\mathcal{V}}$.

We shall write simply $\mathcal{T}$ instead of $\mathcal{T}_{\Sigma,\mathcal{V}}$ omitting the symbol signature and set of variables when implicit.

For example, given the signature Σ such that $p \in \Sigma_3$, $h \in \Sigma_2$, $f \in \Sigma_1$, and $a \in \Sigma_0$, and given that W, X, Y, and Z are variables in $\mathcal{V}$, the terms $p(Z, h(Z, W), f(W))$ and $p(f(X), h(Y, f(a)), Y)$ are in $\mathcal{T}$.

A *substitution* is a finitely non-identical assignment of terms to variables; *i.e.*, a function σ from $\mathcal{V}$ to $\mathcal{T}$ such that the set $\{X \in \mathcal{V} \mid X \neq \sigma(X)\}$ is finite. This set is called the *domain* of σ and denoted by $\textbf{DOM}(\sigma)$. Such a substitution is also written as a set such as $\sigma = \{t_i/X_i\}_{i=1}^n$ where $\textbf{DOM}(\sigma) = \{X_i\}_{i=1}^n$ and $\sigma(X_i) = t_i$ for $i = 1$ to n.

A substitution σ is uniquely extended to a function $\overline{\sigma}$ from $\mathcal{T}$ to $\mathcal{T}$ as follows:

- $\overline{\sigma}(X) = \sigma(X)$, if $X \in \mathcal{V}$;
- $\overline{\sigma}(a) = a$, if $a \in \Sigma_0$;
- $\overline{\sigma}(f(t_1, \ldots, t_n)) = f(\overline{\sigma}(t_1), \ldots, \overline{\sigma}(t_n))$, if $f \in \Sigma_n$, $t_i \in \mathcal{T}, (1 \leq i \leq n)$.

Since they coincide on $\mathcal{V}$, and for notation convenience, we deliberately confuse a substitution σ and its extension $\overline{\sigma}$. Also, rather than writing $\sigma(t)$, we shall write $t\sigma$. Given two substitutions $\sigma = \{t_i/X_i\}_{i=1}^n$ and $\theta = \{s_j/Y_j\}_{j=1}^m$, their composition $\sigma\theta$ is the substitution which yields the same result on all terms as first applying σ then applying θ on the result. One computes such a composition as the set:

$$\sigma\theta = \ \big(\, \{t\theta/X \mid t/X \in \sigma\} \ - \ \{X/X \mid X \in \textbf{DOM}(\sigma)\} \,\big) \ \cup \ \big(\, \theta \ - \ \{s/Y \mid Y \in \textbf{DOM}(\sigma)\} \,\big).$$

For example, if $\sigma = \{f(Y)/X, U/V\}$ and $\theta = \{b/X, f(a)/Y, V/U\}$, then composing σ and θ yields $\sigma\theta = \{f(f(a))/X, f(a)/Y, V/U\}$; composing θ and σ gives $\theta\sigma = \{b/X, f(a)/Y, U/V\}$.

Substitution composition defines a preorder (*i.e.*, a reflexive and transitive) relation on substitutions. A substitution σ is said to be *more general* than a substitution θ iff there exists a substitution ϱ such that $\theta = \sigma\varrho$. For example, $\{f(Y)/X\}$ is more general than $\{f(f(a))/X, f(a)/Y\}$.

Appendix B. Monoidal algebra

We recall some simple, but often overlooked, facts linking monoidal operations to corresponding order relations. These basic facts are important in that they allow viewing monoidal computation as an approximation process based on the associated order.

A *monoidal algebra* is a structure $\langle D, \star\rangle$ consisting of a *domain* D of elements—*i.e.*, a set—with an internal operation $\star : D \times D \mapsto D$. In any monoidal algebra, the operation $\star$ has an associated *prefix* relation defined as:

$$\forall x, y \in D, \;\; x \prec_\star y \;\; \textbf{iff} \;\; \exists z \in D, \; x \star z = y. \tag{27}$$

§**Semigroup –** A *semigroup* $\langle D, \star\rangle$ is a monoidal algebra with domain D whose operation $\star$ is *associative*; that is,

$$\forall x, y, z \in D, \;\; x \star (y \star z) = (x \star y) \star z. \tag{28}$$

Note that in a semigroup $\langle D, \star\rangle$, the prefix relation $\prec_\star$ is always transitive (by virtue of associativity of $\star$). However, it is not necessarily reflexive.

§**Monoid –** A *monoid* $\langle D, \star, \mathbf{1}_\star\rangle$ is a semigroup $\langle D, \star\rangle$ with a special element $\mathbf{1}_\star \in D$, called the ($\star$-)*identity*—or *unit*—element, such that:

$$\forall x \in D, \;\; x \star \mathbf{1}_\star = \mathbf{1}_\star \star x = x. \tag{29}$$

Note that in a monoid $\langle D, \star, \mathbf{1}_\star\rangle$, the prefix relation $\prec_\star$ is also reflexive (by virtue of the unit element). Therefore, it is a preorder, and is sometimes called the monoid's *prefix approximation*. For example, first-order terms substitutions form such a monoid. The prefix relation for substitution composition is precisely the usual "more general than" quasi-ordering. It is a preorder only because it is not anti-symmetric—*i.e.*, in this case $\prec_\star \cap \succ_\star$ is equality up to variable renaming.

§**Commutative structure –** A *commutative* structure is any of the foregoing structures whose operation $\star$ also obeys the *commutativity* axiom:

$$\forall x, y \in D, \;\; x \star y = y \star x. \tag{30}$$

§**Semilattice –** A *semilattice* $\langle D, \star\rangle$ is a commutative semigroup such that $\star$ is *idempotent*; *i.e.*:

$$\forall x \in D, \;\; x \star x = x. \tag{31}$$

Another natural relation may be defined in terms of the $\star$ operation in an idempotent monoidal structure. It is the relation $\leq_\star$ on D defined as:

$$\forall x, y \in D, \; x \leq_\star y \;\; \textbf{iff} \;\; x \star y = y. \tag{32}$$

The relation $\leq_\star$ is called the *semilattice ordering* and indeed defines a partial order on D. Namely, $\leq_\star$ is reflexive (by idempotence of $\star$), anti-symmetric (by commutativity of $\star$) and transitive (by associativity of $\star$).

In a semilattice, the prefix relation $\prec_\star$ is also an ordering and furthermore it coincides with the semilattice ordering; that is, $\forall x, y, x \prec_\star y$ iff $x \leq_\star y$.

Proof Assume that $x \leq_\star y$. By definition, this means that $x \star y = y$. Thus, it is clear that $\exists z,\ x \star z = y$ (taking $z = y$). Therefore, $x \prec_\star y$. Now assume that $x \prec_\star y$. Then, by definition, $x \star z_{xy} = y$ for some $z_{xy} \in D$. Hence,

$$\begin{aligned} x \star y &= x \star (x \star z_{xy}) && \text{(replacing } y \text{ by its value)} \\ &= (x \star x) \star z_{xy} && \text{(associativity)} \\ &= x \star z_{xy} && \text{(idempotence)} \\ &= y \end{aligned}$$

and so, $x \leq_\star y$. ■

Note that, $\star$ is automatically a *supremum* operation for its semilattice ordering; namely, for all $x, y, z \in D$:

$$\textbf{if}\ \ y \leq_\star x\ \ \textbf{and}\ \ z \leq_\star x\ \ \textbf{then}\ \ y \star z \leq_\star x. \tag{33}$$

Proof Assume that $y \leq_\star x$ and $z \leq_\star x$; then,

$$\begin{array}{rcll} y \star x & = & x & \text{[by (32)]} & (a) \\ z \star x & = & x & \text{[by (32)]} & (b) \\ (y \star x) \star (z \star x) & = & x \star x & \text{[by } (a) \text{ and } (b)\text{]} \\ (y \star x) \star (z \star x) & = & x & \text{[by (31)]} \\ (y \star z) \star (x \star x) & = & x & \text{[by (28) and (30)]} \\ (y \star z) \star x & = & x & \text{[by (31)]} \\ y \star z & \leq_\star & x & \text{[by (32)].} \end{array}$$

■

Finally, note that if a semilattice $\langle D, \star \rangle$ is also a monoid $\langle D, \star, \mathbf{1}_\star \rangle$, Equation (32) entails that $\mathbf{1}_\star$ is the (necessarily unique) *least* element of D for $\leq_\star$. Then, it is sometimes written as $\perp$ (and called *bottom*). Thus, a semilattice with bottom can also be described as an idempotent commutative monoid.

The following table gives examples of common monoidal algebras.

Domain	$\star$	$\mathbf{1}_\star$	$\prec_\star$	**Algebra**
Σ^*	$\cdot$	ϵ	$\prec$	*free monoid*
$\mathbb{R}$	$+$	0	$\leq$	*commutative monoid*
$\mathbb{N}$	$*$	1	*divides*	*commutative monoid*
$\mathbf{2}^S$	$\cup$	$\emptyset$	$\subseteq$	*semilattice*
$\mathbf{2}^S$	$\cap$	S	$\supseteq$	*semilattice*

In this table, Σ is a finite alphabet, and Σ^* is the set of all finite strings of symbols in Σ, including the empty string ϵ. The operation '$\cdot$' is string concatenation. $\mathbb{N}$ is the set of natural numbers. $\mathbb{R}$ is the set of real numbers. The set S is any non empty set.

Appendix C. Strong Extensionality

Basically, the reason why the *"Weak Extensionality"* rule of Figure 10 fails for cyclic terms is that it works *inductively*, starting from terms' leaves to their roots.

Consider, for example, an extensional sort $s \in \mathcal{E}$ such that $\textbf{ARITY}(s) = \{f\}$, and the terms:

$$X : s(f \Rightarrow X) \;\&\; X' : s(f \Rightarrow X') \tag{34}$$

or, even better, the terms:

$$X : s(f \Rightarrow X') \;\&\; X' : s(f \Rightarrow X). \tag{35}$$

Now, $\textbf{ARITY}(s) = \{f\}$ means that *"s denotes a singleton sort whenever its f feature denotes one as well."* Semantically, in both examples, variables X and X' denote therefore the same element (due to *all* the features in $\textbf{ARITY}(s)$ being consistently sorted as singletons). However, the "weak extensionality" rule will not transform either the terms in Examples 34 or 35 into one where X and X' are equal as they should be as per the semantics of arity and extensionality.

Clearly, this inductive manner of proceeding cannot work for cyclic extensional terms such as Examples 34 or 35. As was seen in Section 4.2, we may thus proceed *coinductively* from roots to leaves keeping a record of which extensional sorts appear with which variables. This is done by carrying a *context* Γ, a set of elements of the form $s : \{X_1, \ldots, X_n\}$, where $X_i \in \mathcal{V}$, for $i = 1, \ldots, n, (n \geq 0)$, where $s \in \mathcal{E}$ is extensional, and such that each such s occurs at most *once* in any such context Γ. A *contexted rule* is one of the form:

$(\mathcal{A}_n)$ **RULE NAME** :

$$\left[\texttt{Condition}\right] \qquad \frac{\texttt{Prior Context} \vdash \texttt{Prior Form}}{\texttt{Posterior Context} \vdash \texttt{Posterior Form}}$$

Appropriate extensional sort occurrences record-keeping is thus achieved using contexted Rule *Extensional Variable* in Figure 20. The "real" work is then done by contexted Rule *"Strong Extensionality"* in Figure 20. Using these two rules on weak normal forms will work as expected; *viz.*, it will merge any remaining potential cyclic extensional elements that denote the same individual.

$(\mathcal{O}_{14})$ **EXTENSIONAL VARIABLE**:

$$\left[\begin{array}{l}\textbf{if } s \in \mathcal{E} \textbf{ and } X \notin V \textbf{ and } \forall f \in \textbf{ARITY}(s) : \\ \{X.f \doteq X', X' : s'\} \subseteq \phi \text{ with } s' \in \mathcal{E}\end{array}\right] \qquad \frac{\Gamma \uplus \{s : V, \ldots,\} \vdash \phi \;\&\; X : s}{\Gamma \uplus \{s : V \cup \{X\}, \ldots\} \vdash \phi \;\&\; X : s}$$

$(\mathcal{O}_{15})$ **STRONG EXTENSIONALITY**:

$$\left[\textbf{ if } s \in \mathcal{E} \right] \qquad \frac{\Gamma \uplus \{s : \{X, X', \ldots\} \vdash \phi}{\Gamma \uplus \{s : \{X, \ldots\} \vdash \phi \;\&\; X \doteq X'}$$

Figure 20: $\mathcal{OSF}$**-constraint strong extensionality normalization rules**

References

Peter Aczel. *Non Well-Founded Sets.* Center for the Study of Language and Information, Stanford, CA, USA, 1988. [Available online [4]].

Hassan Aït-Kaci. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Types.* PhD thesis, University of Pennsylvania, Philadelphia, PA, 1984.

Hassan Aït-Kaci. An algebraic-semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

Hassan Aït-Kaci. An introduction to LIFE—Programming with Logic, Inheritance, Functions, and Equations. In Dale Miller, editor, *Proceedings of the International Symposium on Logic Programming*, pages 52–68. MIT Press, October 1993. [Available online [5]].

Hassan Aït-Kaci. "Go (Semantic) Web, young (CP) wo/man!". Panel presentation at the CP 2006 Workshop on the *"Next 10 Years of Constraint Programming,"* organized by Lucas Bordeaux, Barry O'Sullivan, and Pascal Van Hentenryck, Nantes, France, September 2006.

Hassan Aït-Kaci, Bruno Berstel, Ulrich Junker, Michel Leconte, and Andreas Podelski. Satisfiability modulo structures as constraint satisfaction. Research paper, submitted for publication, ILOG, Inc., December 2006.

Hassan Aït-Kaci, Bruno Berstel, Ulrich Junker, Michel Leconte, and Andreas Podelski. Satisfiability modulo structures as constraint satisfaction: An introduction. In Pierre-Etienne Moreau, editor, *Actes des Journées Francophones des Langages Applicatifs*, pages 1–8, Aix les Bains, France, January 2007. Insititut National de Recherche en Informatique et Automatique, INRIA. [Available online [6]].

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

Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, and Peter Van Roy. The Wild LIFE handbook. [Available online [7]], 1994a.

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

Hassan Aït-Kaci and Roger Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

Hassan Aït-Kaci, Roger Nasr, and Patrick Lincoln. Le Fun: Logic, equations, and Functions. In *Proceedings of the Symposium on Logic Programming (San Francisco, CA)*, pages 17–23, Washington, DC, 1987. IEEE, Computer Society Press.

4. `http://standish.stanford.edu/pdf/00000056.pdf`
5. `http://koala.ilog.fr/wiki/pub/Main/HassanAitKaci/ilps93.ps.gz`
6. `http://www.loria.fr/~moreau/jfla2007/`
7. `http://citeseer.ist.psu.edu/134450.html`

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

Hassan Aït-Kaci and Andreas Podelski. Functions as passive constraints in LIFE. *ACM Transactions on Programming Languages and Systems*, 16(4):1279–1318, July 1994. [Available online[9]].

Hassan Aït-Kaci, Andreas Podelski, and Seth C. Goldstein. Order-sorted feature theory unification. *Journal of Logic Programming*, 30(2):99–124, 1997. [Available online [10]].

Hassan Aït-Kaci, Andreas Podelski, and Gert Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, January 1994b. [Available online [11]].

Franz Baader and Werner Nutt. Basic description logics. In Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, chapter 2, pages 47–100. Cambridge University Press, 2003. [Available online [12]].

Franz Baader and Ulrike Sattler. Description logics with aggregates and concrete domains. In *Proceedings of the International Workshop on Description Logics*, Gif sur Yvette, France, 1997. [Available online [13]].

J. C. M. Baeten and W. P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, October 1990.

Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of progrramming. INRIA Technical Report 566, Institut National de Recherche en Informatique et Automatique, Le Chesnay, France, 1986.

Salima Benbernou and Mohand-Said Hacid. Resolution and constraint propagation for semantic web services discovery. *Journal of Distributed and Parallel Databases*, 18(1):65–81, July 2005.

Gérard Berry and Gérard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'90*, pages 81–94, New York, NY, USA, 1990. ACM Press.

Garrett Birkhoff. *Lattice Theory*, volume 25 of *Colloquium Publications*. American Mathematical Society, Providence, RI, USA, 3rd edition, 1979.

Ronald Brachman. *A Structural Paradigm for Representing Knowledge*. PhD thesis, Harvard University, Cambridge, MA, USA, 1977.

Alexander Brodsky. Constraint databases: Promising technology or just intellectual exercise? *ACM Computing Surveys*, 28(4):59, 1996.

8. `http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-11.pdf`
9. `http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-13.pdf`
10. `http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-32.pdf`
11. `http://www.hpl.hp.com/techreports/Compaq-DEC/PRL-RR-20.pdf`
12. `http://citeseer.ist.psu.edu/baader03basic.html`
13. `http://citeseer.ist.psu.edu/article/baader98description.html`

Martin Bucheit, Francesco M. Donini, and Andrea Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993. [Available online [14]].

Bob Carpenter. Typed feature structures: A generalization of first-order terms. In Vijay Saraswat and Kazunori Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 187–201, Cambridge, MA, 1991. MIT Press.

Bob Carpenter. *The Logic of Typed Feature Structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1992.

Alain Colmerauer. An introduction to Prolog III. *Communication of the ACM*, 33(7):69–90, 1990.

Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Mike Uschold, and Lora Aroyo, editors. *The Semantic Web—ISWC 2006*, number 4273 in Lecture Notes in Computer Science, 2006. 5th International Semantic Web Conference, ISWC 2006, Athens, GA, November 2006, Springer-Verlag.

Luís Damas, Nelma Moreira, and Giovanni B. Varile. The formal and computational theory of complex constraint solution. In C. Rupp, M. A. Rosner, and R. L. Johnson, editors, *Constraints, Language, and Computation*, Computation in Cognitive Science, pages 149–166. Academic Press, London, 1994.

Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Reasoning in description logics. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 191–236. CSLI Publications, Stanford, CA, 1996. [Available online [15]].

Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. $\mathcal{AL}$-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998. [Available online [16]].

Jochen Dörre and William C. Rounds. On subsumption and semiunification in feature algebras. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (Philadelphia, PA)*, pages 301–310, Washington, DC, 1990. IEEE, Computer Society Press.

Jochen Dörre and Roland Seiffert. Sorted feature terms and relational dependencies. In Bernhard Nebel, Kai von Luck, and Christof Peltason, editors, *Proceedings of the International Workshop on Terminological Logics*, pages 109–116. DFKI, 1991. [Available online [17]].

Martin C. Emele. Unification with lazy non-redundant copying. In *Proceedings of the 29th annual meeting of the ACL*, Berkeley, California, June 1991. Association for Computational Linguistics.

Martin C. Emele and Rémi Zajac. Typed unification grammars. In *Proceedings of the 13th International Conference on Computational Linguistics—CoLing'90*, Helsinki, Finland, August 1990.

14. `http://arxiv.org/PS\_cache/cs/pdf/9312/9312101.pdf`
15. `http://citeseer.ist.psu.edu/article/donini97reasoning.html`
16. `http://citeseer.ist.psu.edu/donini98allog.html`
17. `http://elib.uni-stuttgart.de/opus/volltexte/1999/93/`

Martin C. Emele and Rémi Zajac. A fixed point semantics for feature type systems. In *Proceedings of the 2nd International CTRS Workshop, Montreal (June 1990)*, number 516 in Lecture Notes in Computer Science, pages 383–388. Springer-Verlag, 1992.

Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, December 2000. [Available online [18]].

Bernd Fischer. Resolution for feature logics. In *GI-Fachgruppe über Alternative Konzepte für Sprachen und Rechner*, pages 23–34. GI Softwaretechnik Trends, April 1993. [Available online [19]].

Jurgen Frohn, Rainer Himmeroder, Paul-Thomas Kandzia, Georg Lausen, and Christian Schlepphorst. FLORID: a prototype for F-Logic. In *Proceedings of the 13th International Conference on Data Enginereeing—ICDE'97*, page 583*ff*, Birmingham, UK, April 1997. [Available online [20]].

Joseph Goguen. What is unification? In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures—Algebraic Techniques*, chapter 6, pages 217–261. Academic Press, 1989. [Available online [21]].

Benjamin Grosof, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proceedings of the World Wide Web Conference—WWW 2003*, pages 48–57, Budapest, Hungary, May 2003.

Torsten Grust. A versatile representation for queries. In P.M.D. Gray, L. Kerschberg, P.J.H. King, and A. Poulovassilis, editors, *The Functional Approach to Data Management: Modeling, Analyzing and Integrating Heterogeneous Data*. Springer, September 2003. [Available online [22]].

Jochen Heinsohn, Daniel Kudenko, Bernhard Nebel, and Hans-Jrgen Profitlich. An empirical analysis of terminological representation systems. *Artifical Inteligence*, 68(2):367–397, 1994. [Available online [23]].

Jacques Herbrand. *Logical Writings*. Harvard University Press, Cambridge, MA, 1971. Edited by Warren D. Goldfarb.

Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, IBM Deutschland, Stuttgart, Germany, October 1988. [Available online [24]].

Ian Horrocks and Peter F. Patel-Schneider. Optimising propositional modal satisfiability for description logic subsumption. In *Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation—AISC'98*, number 1476 in Lecture Notes in Computer Science, pages 234–246. Springer-Verlag, 1998. [Available online [25]].

18. `http://lambda.uta.edu/tods00.ps.gz`
19. `http://www.infosun.fmi.uni-passau.de/st/papers/resolution/`
20. `http://citeseer.ist.psu.edu/frohn97florid.html`
21. `http://www-cse.ucsd.edu/~goguen/pps/subs.ps`
22. `http://www.fmi.uni-konstanz.de/~grust/files/monad-comprehensions.pdf`
23. `http://citeseer.ist.psu.edu/article/heinsohn94empirical.html`
24. `http://citeseer.ist.psu.edu/hohfeld88definite.html`
25. `http://citeseer.ist.psu.edu/article/horrocks98optimising.html`

Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning—LPAR'99*, number 1705 in Lecture Notes in Computer Science, pages 161–180. Springer-Verlag, 1999. [Available online [26]].

Gérard Huet. *Constrained Resolution: A Complete Method for Higher-Order Logic.* PhD thesis, Case Western Reserve University, Cleveland, OH, USA, 1972.

Gérard Huet. Résolution d'équations dans des langages d'ordre 1, 2, . . . , ω. Thèse d'état, Université de Paris VII, Paris, France, 1976.

Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, W. Germany, January 1987.

Joxan Jaffar and Michael J. Maher. Constraint Logic Programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994. [Available online [27]].

Paris C. Kanellakis and Dina Q. Goldin. Constraint programming and database query languages. In *Proceedings the International Conference on Theoretical Aspects of Computer Software—TACS'94*, pages 96–120. Springer-Verlag, 1994.

Michael Kifer. Flora-2. `http://flora.sourceforge.net/`, September 9, 2007. Version 0.95 (Androcymbium).

Michael Kifer, Jos de Bruijn, Harold Boley, and Dieter Fensel. A realistic architecture for the Semantic Web. In *Proceedings of the International Conference on Rules and Rule Markup Languages for the Semantic Web—RuleML'05*, number 3791 in Lecture Notes in Computer Science, pages 17–29. Springer-Verlag, 2005. [Available online [28]].

Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object oriented and frame based languages. *Journal of the ACM*, 42(4):741–843, 1995. [Available online [29]].

Hans-Ulrich Krieger and Ulrich Schäfer. TDL—A type description language for constraint-based grammars. In *Proceedings of the 15th Conference on Computational linguistics*, pages 893–899, Morristown, NJ, USA, 1994. Association for Computational Linguistics. [Available online [30]].

Markus Krötzsch, Pascal Hitzler, Denny Vrandečić, and Michael Sintek. How to reason with OWL in a logic programming system. In Thomas Eiter, Enrico Franconi, Ralph Hodgson, and Susie Stephens, editors, *Proceedings of the 2nd International Conference on Rules and Rule Markup Languages for the Semanitic Web, Athens, GA—RuleML 2006*, pages 17–26. IEEE Computer Society, November 2006.

Patrick Lambrix. Description logics. `http://www.ida.liu.se/labs/iislab/people/patla/DL/`, 2006. Resource web page.

26. `http://citeseer.ist.psu.edu/article/horrocks99practical.html`
27. `http://citeseer.ist.psu.edu/jaffar94constraint.html`
28. `http://www.debruijn.net/publications/msa-ruleml05.pdf`
29. `ftp://ftp.cs.sunysb.edu/pub/TechReports/kifer/flogic.pdf`
30. `http://www.citebase.org/`

Daniel Le Métayer. Higher-order multiset programming. In *Proceedings of the DIMACS workshop on specifications of parallel algorithms*. American Mathematical Society, DIMACS series in Discrete Mathematics, 1994. Volume 18.

Alon Y. Levy and Marie-Christine Rousset. CARIN: A representation language combining Horn rules and description logics. In *European Conference on Artificial Intelligence*, pages 323–327, 1996. [Available online [31]].

Alon Y. Levy and Marie-Christine Rousset. Combining Horn rules and description logics in CARIN. *Artificial Intelligence*, 104(1-2):165–209, 1998.

Carsten Lunz. Description logics. `http://dl.kr.org/`, 2006. Resource web page.

Michael J. Maher. Complete axiomatizations of the algebras of finite rational and infinite trees. In *Proceedings of the 3rd Conference on Logic in Computer Science—LICS'88*, pages 348–357. IEEE Computer Society, June 1988a.

Michael J. Maher. Complete axiomatizations of the algebras of finite rational and infinite trees. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, 1988b. [Available online [32]].

Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, NATO ISI Series. Springer-Verlag, 1991. [Available online [33]].

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

Boris Motik, Ian Horrocks, Riccardo Rosati, and Ulrike Sattler. Can OWL and Logic Programming live together happily ever after? In I. Cruz *et al.*, editor, *Proceedings of the 5th International Semantic Web Conference—ISWC 2006*, number 4273 in Lecture Notes in Computer Science, pages 501–514. Springer-Verlag, November 2006.

Martin Müller, Joachim Niehren, and Andreas Podelski. Ordering constraints over feature trees. *Constraints*, 5(1–2):7–42, January 2000. Special issue on CP'97, Linz, Austria. [Available online [34]].

Martin Müller, Joachim Niehren, and Ralf Treinen. The first-order theory of ordering constraints over feature trees. *Discrete Mathematics & Theoretical Computer Science*, 4(2):193–234, 2001. [Available online [35]].

Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in AI and Logic Programming, Volume 5: Logic Programming*, pages 499–590. Oxford University Press, 1998. [Available online [36]].

31. `http://citeseer.ist.psu.edu/article/levy96carin.html`
32. `http://www.cse.unsw.edu.au/~mmaher/pubs/trees/axiomatizations.pdf`
33. `http://citeseer.ist.psu.edu/manna92fundamentals.html`
34. `http://www.ps.uni-sb.de/Papers/abstracts/ftsub-constraints-99.html`
35. `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/dm040211.ps`
36. `http://www-users.cs.umn.edu/~gopalan/papers/holp.ps`

Bernhard Nebel and Gert Smolka. Representation and reasoning with attributive descriptions. In K.H. Blasius, U. Hedtstuck, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 418 of *Lecture Notes in Artificial Intelligence*, pages 112–139. Springer-Verlag, 1990.

Bernhard Nebel and Gert Smolka. Attributive description formalisms and the rest of the world. In O. Herzog and C.-R. Rollinger, editors, *Text Understanding in LILOG: Integrating Computational Linguistics and Artificial Intelligence*, volume 546 of *Lecture Notes in Artificial Intelligence*, pages 439–452. Springer-Verlag, 1991.

Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Århus, Århus, Denmark, 1981. [Available online [37]].

Alun Preece, Stuart Chalmers, Craig McKenzie, Jeff Z. Pan, and Peter Gray. Handling soft constraints in the semantic web architecture. In Pascal Hitzler, Holger Wache, and Thomas Eiter, editors, *Online proceedings of the WWW 2006 Workshop on Reasoning on the Web*, Edinburgh, May 2006. [Available online [38]].

John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, January 1965.

Konstantinos Sagonas, Terrance Swift, and David Scott Warren. The XSB programming system. In *Proceedings of the ILPS'93 Workshop on Programming with Logic Databases*, pages 164–193, 1993.

Davide Sangiorgi. Coinduction in programming languages. Invited Lecture ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages—POPL'04, January 2004. [Available online [39]].

Vijay Saraswat. *Concurrent Constraint Programming*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, January 1989.

Manfred Schmidt-Schauß and Jörg Siekmann. Unification algebras: An axiomatic approach to unification, equation solving and constraint solving. Technical Report SEKI-report SR-88-09, FB Informatik, Universität Kaiserslautern, 1988. [Available online [40]].

Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

Yi-Dong Shen, Li-Yan Yuan, Jia-Huai You, and Neng-Fa Zhou. Linear tabulated resolution based on Prolog control strategy. *Theory and Practice of Logic Programming*, 1(1):71–103, 2001.

Gert Smolka. A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland, Stuttgart, Germany, May 1988.

37. `http://citeseer.ist.psu.edu/plotkin81structural.html`
38. `http://www.aifb.uni-karlsruhe.de/WBS/phi/RoW06/`
39. `http://www.cs.unibo.it/~sangio/DOC\_public/TalkPOPL.ps.gz`
40. `http://www.ki.informatik.uni-frankfurt.de/papers/schauss/unif-algebr.pdf`

Gert Smolka. Feature constraint logic for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

Gert Smolka. Residuation and guarded rules for Constraint Logic Programming. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 405–419, Cambridge, MA, 1993. The MIT Press. Chapter 22.

Gert Smolka. A foundation for higher-order concurrent constraint programming. In Jean-Pierre Jouannaud, editor, *Proceedings of the 1st International Conference on Constraints in Computational Logics*, number 845 in Lecture Notes in Computer Science, pages 50–72. Springer-Verlag, September 1994.

Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18 (3):229–258, April 1994. [Available online [41]].

Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, January 1997. [Available online [42]].

Jeffrey Ullman. Introduction to datalog and stratified negation. `http://infolab.stanford.edu/~ullman/cs345notes/cs345-1.pdf`, 2003. CS 345 Lecture Notes.

Rémi Zajac. Towards object-oriented constraint logic programming. In *Proceedings of the ICLP'91 Workshop on Advanced Logic Programming Tools and Formalisms for Natural Language Processing*, Paris, June 1991.

Rémi Zajac. Inheritance and constraint-based grammar formalisms. *Computational Linguistics.*, 18 (2):159–182, 1992.

41. `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/ST-jlp94.ps`
42. `http://www.lsv.ens-cachan.fr/Publis/PAPERS/PS/FeatArbStruct.ps`