Object-Oriented Logic Programming
with OBJECTLOG

Davide Martinenghi

Master's thesis at
Department of Computer Science
Technical University of Denmark

Supervisor
Professor Jørgen Fischer Nilsson

May 1998
Abstract

This project is concerned with integration of objects into logic programming. In the object-oriented logic programming language proposal OBJECTLOG the traditional term data structures are replaced with object-structured terms with an accompanying data type specificational structure. The data types constitute an algebraic lattice which provides multiple inheritance of attributes.

The project comprises the following phases:

1. Specification of OBJECTLOG on basis of the available description along with literature study concerning similar contemporary language proposals. Elaboration of a major example within artificial intelligence or expert systems (a small natural language demo translator).

2. Design of a compiler for OBJECTLOG (or an appropriate subset thereof) for compilation into ordinary Prolog. In the course of this, a small literature study concerning equational Horn clause logic and unification theory should be done.

3. Implementation of the compiler using Prolog (or OBJECTLOG itself through some bootstrapping stages) and test of the compiler. Report on the project.
Preface

This thesis presents the results of Davide Martinenghi's Master's project on Object-Oriented Logic Programming with the OBJECTLOG programming language. The work was done from October 1997 till May 1998 at the Department of Information Technology at the Technical University of Denmark under supervision of Professor Jørgen Fischer Nilsson.

I would like to thank Jørgen Fischer Nilsson for his invaluable assistance in writing this thesis. I am also extremely grateful to my parents for their precious help on a section of the first chapter and to my friends, who kept encouraging me during the whole period and showed sincere interest in this project. My deepest gratitude goes also to Andrea Mazzilli for being willing to read part of the manuscript and give me useful hints.
Contents

1.1 Demand for complex data structures ................................................................. 1
1.2 Objectives ......................................................................................................... 1
1.3 Organization ..................................................................................................... 2

Chapter 1 Overview of the tool ............................................................................... 5
1.1 Descriptions in Logic Programming ................................................................. 5
1.1.1 A two-level language with descriptions ....................................................... 5
1.2 Terms in ObjectLog .......................................................................................... 6
1.2.1 Frame terms and attributes ............................................................................ 6
1.2.2 Mapping Prolog's compound terms into ObjectLog ........................................ 7
1.2.3 Terms in general ........................................................................................... 7
1.3 The Lattice structure of Concept Algebra ....................................................... 8
1.3.1 Introducing lattices ....................................................................................... 8
1.3.2 Axioms for lattices ....................................................................................... 9
1.3.3 Axioms for attributions ............................................................................... 10
1.4 Interpretations of Concept Algebra ................................................................. 12
1.4.1 Set interpretation of concept algebra (powerset model) ................................. 12
1.4.2 Topological interpretation of concept algebra .............................................. 13
1.4.3 The inverse image ....................................................................................... 15
1.5 ObjectLog combines Prolog and Concept Algebra .......................................... 16
1.5.1 Concept algebraic terms in definite clauses ................................................ 16
1.5.2 Type equations and inequalities ................................................................. 17
1.5.3 Restrictions in type equations ..................................................................... 20
1.5.4 Defining individuals and querying ............................................................. 21
1.5.5 Ordered and unordered typed lists ............................................................. 22
1.6 A Recapulating Example .................................................................................. 23
1.6.1 Good and bad dinners ............................................................................... 23
1.6.2 Points of discussion .................................................................................... 25
1.7 Summary of the chapter .................................................................................. 26

Chapter 2 Unification of complex objects .............................................................. 29
2.1 Unification in ObjectLog .................................................................................. 29
2.1.1 Restricting the language .............................................................................. 29
2.1.2 Some remarks about unification in general ................................................. 30
2.1.3 An algorithm for unification in ObjectLog1 ................................................ 31
2.1.4 Examples of unification .............................................................................. 34
2.1.5 An algorithm for concept inclusion in ObjectLog1 ...................................... 35
2.1.6 Relaxing the pure/ground constraint ......................................................... 36
2.1.7 Concluding remarks ................................................................................... 38
2.2 Introducing type equations ............................................................................. 39
2.2.1 Defining types ............................................................................................ 39
2.2.2 Examples of type equations ....................................................................... 40
2.2.3 An algorithm for ground isa check in ObjectLog2 ...................................... 41
2.2.4 Examples of ground isa check .................................................................... 42
2.2.5 Definition of atomicity ............................................................................... 44
2.2.6 Simplest cases of unification with types ..................................................... 45
2.2.7 Unification for ObjectLog2 in the general case .......................................... 48
2.3 Towards ObjectLog ....................................................................................... 49
2.3.1 Including type inequalities ......................................................................... 49
2.3.2 Sums in the program .................................................................................. 50

Chapter 3 Links to related theories ....................................................................... 53
3.1 Similar proposals ............................................................................................. 53
3.1.1 Wisdom ....................................................................................................... 53
3.1.2 LIFE ............................................................................................................ 55
3.1.3 LOT ............................................................................................................. 57
3.2 An approach to Relational Databases ............................................................ 59
3.2.1 Recasting relational databases in ObjectLog .............................................. 59
3.2.2 Relational algebra operations ..................................................................... 60
3.2.3 Querying a database .................................................................................. 61
Introduction

1.1 Demand for complex data structures

High level programming languages traditionally offer abstract data types such as lists and trees. In particular, ordinary logic programming deals with the notion of terms. Artificial intelligence and knowledge representation call, however, for richer structures for expressing inheritance between complex objects.

OBJECTLOG fulfills these requirements by appealing to the object-oriented programming paradigm with accompanying is-a relationships. Terms from ordinary logic programming are replaced by more complex terms derived from concept algebra ([21] and [19]) where objects are understood as descriptions. The concept algebra is a logico-algebraic theory that merges inheritance capabilities of lattice theory (formalized in [6]) with attributions in the sense of frames, which are structures well known from AI. The resulting mathematical model is surprisingly simple and clean; yet, OBJECTLOG is a very expressive language that encompasses the fundamentals of relational database theory as a special case, and that captures the essence of conceptual graphs and semantic nets ([21] and [17]).

1.2 Objectives

This project comes along with a number of purposes.

First of all, we want to describe the OBJECTLOG programming language according to the theoretical papers, written by Prof. Nilsson at the Technical University of Denmark, this work is mainly based on. In addition, we want to provide a variety of examples that show how this language can deal with complex objects.

An extension of theoretical approach is also desired, especially for what concerns possible interpretations of the algebra. A sketchy description of the topological view of concept algebra is reported.

Special attention is devoted to the implementation problem. OBJECTLOG, considered in its entirety, is a very broad framework that cannot be easily handled without a preliminary study of subsets thereof. Our ambition is therefore that of cautiously identifying manageable sub-cases that can be sufficient for writing actual programs as well as efficiently implemented. In particular, limitations on the admissible forms of complex OBJECTLOG terms during unification need to be delineated with special care.

Finally, it is our intention to develop a major application written in OBJECTLOG showing how this language can be fruitfully exploited, and whether complex objects and inheritance can be advantageously used in logic programming for practical purposes. Particularly, we are going to present a small natural language translator from English to Italian and vice versa. The aim of this translator is to evaluate OBJECTLOG as a programming language and to establish what programming style it calls for, rather than providing a reliable tool for rendering sentences of a given language into another language.
Chapter 1 introduces OBJECTLOG as an extension of logic programming that embeds concept algebraic complex objects in definite clauses. Complex objects are organized in a lattice according to axioms establishing the behavior of the operators. Several views of the algebra are proposed, and formal interpretations complying with the axioms are explained. Types are introduced as a facility for specifying knowledge, and a one-page example shows how this can be done in practice.

Chapter 2 analyzes unification in OBJECTLOG and explains how this can be performed algorithmically for subsets of the language. Two subsets are outlined: OBJECTLOG\textsubscript{1}, and OBJECTLOG\textsubscript{2}. The former disregards sums and types, whereas the latter accepts type equations with sums but prevents from writing full OBJECTLOG terms in programs. Several constraints, both dynamic and static ones (such as the pure/ground constraint), are introduced along the chapter and some of them are partially relaxed. Algorithms for testing inclusion relationship are also described for both subsets, and many examples of unification and is-a clauses are shown and commented. Guidelines for achieving unrestricted OBJECTLOG are also suggested.

In chapter 3, OBJECTLOG is compared to programming languages with similar purposes, namely LIFE, LOT and Wisdom. Differences and similarities are pointed out, and some typical examples written using those tools are paraphrased in OBJECTLOG. An approach to relational databases is also attempted, a general method for recasting relational algebra is proposed, and limitations and drawbacks are mentioned.

Chapter 4 describes the design and the implementation of a compiler for the OBJECTLOG subsets presented in chapter 2. A general approach to compilation of clauses is first explained, then the problems of representation and dynamic adjustment of terms are introduced. The compiler comes along with an interpreter that makes it possible to have a user-friendly OBJECTLOG environment. Facilities such as definite clause grammars, and clauses for representation of texts and strings are also included.

Chapter 5 deals with the natural language translator mentioned above. A small application domain is first individuated, then a methodology for translating sentences, in which semantic and syntactic models are designed and implemented by means of type equations, is proposed. Several sentences are analyzed and represented as OBJECTLOG terms, and then tested with this program. A little analysis of the performance is also reported.

Appendix A contains the fully commented source code of the compiler from chapter 4 available in the version for both OBJECTLOG\textsubscript{1}, and OBJECTLOG\textsubscript{2}.
Appendix B reports the listing of the natural language translator from chapter 5 articulated in several modules corresponding to the various ends of the programs.
Chapter 1  Overview of the tool

1.1  Descriptions in Logic Programming

1.1.1  A two-level language with descriptions

The first order logic atomic formula
\[ p(t_1, \ldots, t_n) \]
expresses an \( n \)-ary relationship between individuals denoted, in the Herbrand universe [23], by the terms \( t_i \), in contrast with OBJECTLOG, whose clauses designate relationships between high-level concept algebraic terms. Semantically, these terms are descriptions, that is concepts whose denotations are subsets of the underlying mathematical universe, which meets the need of having relationships between classes as well. This is particularly important for concept inclusion (\textit{is-a} relationship) as shown in the following example: in ordinary logic programming, a sentence like
\[ \text{isa}(\text{whale}, \text{mammal}) \]
expressing that the class of whales is a subset of the class of mammals, is however understood as a relationship between individuals, although these are common names rather than proper names. Descriptions obviate this problem by denoting \textit{whale} and \textit{mammal} as sets. Individuals are then represented by singleton sets, e.g.:
\[ \text{isa}(\text{george}, \text{man}) \]
indicates that the individual \textit{george} belongs to the class of men, though there is no syntactical difference between \textit{george} and \textit{man}. From a denotation\footnote{Double square brackets are used for indicating denotations.} viewpoint:
\[
\begin{align*}
\llbracket \text{george} \rrbracket & = \{ \text{george} \} \\
\llbracket \text{man} \rrbracket & = \{ \text{george}, \text{peter}, \text{john}, \text{robert}, \ldots \}
\end{align*}
\]

Concept inclusion (subsumption) between descriptors corresponds then to set inclusion between denotations and is available in OBJECTLOG as the built-in predicate \text{isa}/2. Denotations of descriptors may be easily represented in Venn diagrams. In the simple example of fig. 1.1.1.1 \( U \) is the universe and the following relations hold:
\[
\begin{align*}
& \text{isa}(\varphi, \psi) \quad \text{and} \\
& \psi \times \xi = \bot
\end{align*}
\]
\( \perp \) is a special element called bottom whose denotation is the empty set. A consequence of this is that every description subsumes bottom. At the denotation level, the crux \( ^2 \) operator \( \times \), so-called to avoid confusion with the Cartesian product, is set intersection. Therefore the expression:
\[
\psi \times \xi = \perp
\]
means that descriptors \( \psi \) and \( \xi \) are disjoint.

\text{OBJECTLOG operates then at two different levels:}
\begin{itemize}
  \item an object level of concept algebraic terms;
  \item a formal logical meta-level of clauses.
\end{itemize}

Unification can consequently be reconceived as equality of concept algebraic terms, which is performed by means of rewriting rules expressing the axioms of the concept algebra shown in section 1.3.

### 1.2 Terms in ObjectLog

#### 1.2.1 FRAME TERMS AND ATTRIBUTES

Attributions are the algebraic counterpart of the binary relational product (the so-called Peirce product described in [19] and [7]). They are introduced here as unary functors in the following way:
\[
a(\phi)
\]
indicating that the value of the attribute \( a \) is the term \( \phi \). A frame descriptor term is an unordered collection of attribute-value pairs with an optional name constant \( c \):
\[
c \times a_1(\phi_1) \times a_2(\phi_2) \times \cdots \times a_n(\phi_n), \quad n \geq 1
\]
(1.2.1.1)
where the binary \( \times \) operator can be left implicit if no ambiguities arise from the context.

---

2 The etymology of this word is from Latin "crux", literally a cross. In Modern English its actual meaning is however that of "crucial point": (ex.: "the crux of the problem").
Alternative ways of representing such a frame are:

\[
\begin{bmatrix}
a_1 : \phi_1 \\
a_2 : \phi_2 \\
\vdots \\
a_n : \phi_n \\
\end{bmatrix}
\]

or

\[
c \begin{bmatrix}
a_1 : \phi_1 , \ a_2 : \phi_2 , \ \cdots , \ a_n : \phi_n \\
\end{bmatrix}
\]

As an example of frame term, consider the following:

\[
\begin{bmatrix}
\text{book} : \begin{bmatrix}
\text{title} : \text{ulysses} \\
\text{author} : \text{joyce} \\
\text{year} : 1922 \\
\end{bmatrix}
\end{bmatrix}
\]

Notice that the \( \phi_i \)'s are recursively frames (or, more generally, terms, as explained later on in the chapter), therefore, instead of the constant \textit{joyce}, we could use a more complex object, like:

\[
\begin{bmatrix}
\text{person} : \begin{bmatrix}
\text{first\_name} : \text{james} \\
\text{surname} : \text{joyce} \\
\end{bmatrix}
\end{bmatrix}
\]

thus giving the nested object

\[
\begin{bmatrix}
\text{book} : \begin{bmatrix}
\text{author} : \text{person} : \begin{bmatrix}
\text{first\_name} : \text{james} \\
\text{surname} : \text{joyce} \\
\end{bmatrix} \\
\text{year} : 1922 \\
\end{bmatrix}
\end{bmatrix}
\]

These objects closely resemble (possibly hierarchical) tuples, as described in [5].

### 1.2.2 Mapping Prolog’s Compound Terms into ObjectLog

A Prolog’s compound term of the form:

\[ f(t_1, t_2, \ldots, t_n) \]

can be expressed as a frame by introducing \( n \) attributes \( a_{f_i} \) whose meaning is \( \text{"i-th position in compound term } f \text{"} \). The translation becomes then:

\[ a_{f_1}(\phi_1) a_{f_2}(\phi_2) \cdots a_{f_n}(\phi_n) \]

where \( \phi_i \) is in turn the \texttt{OBJECTLOG} representation of the Prolog term \( t_i \). More is said about terms and mapping from Prolog in the remainder of the chapter.

### 1.2.3 Terms in general

A frame term can, in general, be expressed as:
\[
\prod_{j=1}^{m} c_j \times \prod_{j=1}^{n} a_j(\varphi_j)
\]

(1.2.3.1)

Notice that if \( c_i \) and \( c_j \) are two different constants, an assumption is made throughout that they are disjoint, that is

\[
i \neq j \Rightarrow c_i \times c_j = \bot
\]

Therefore it turns out that the most general frame term of interest for this context is of the form (1.2.1.1).

An OBJECTLOG term tout court is something more general than just a frame term, in that it introduces variables and sums. The sum operator + (plex\(^2\)) is the complementary of crux and it is understood from the denotation viewpoint as set union. Briefly, a term can be:

- a constant;
- a variable;
- an attribution: \( a(\varphi) \);
- a product of terms: \( \varphi \times \psi \);
- a sum of terms: \( \varphi + \psi \).

The most general expression of interest\(^4\) for a term is then:

\[
\sum_{l=1}^{q} \left( \prod_{i=1}^{m} c_{i,l} \times \prod_{j=1}^{n} a_{j,l}(\varphi_{j,l}) \times \prod_{k=1}^{p} X_{k,l} \right)
\]

(1.2.3.2)

where \( c_{i,l} \) is the \( i \)-th constant of the \( l \)-th sum term, and similarly for attributions and variables. As for frame terms, having only one constant in a term does not affect the generality of the model, as cruxing disjoint constants\(^5\) leads to inconsistency (\( \bot \)). Furthermore, it will be shown in the chapter about unification that having only one variable is an acceptable restriction for most purposes.

### 1.3 The Lattice structure of Concept Algebra

#### 1.3.1 Introducing Lattices

The operations presented so far, except for attribution, can be viewed as lattice operations in the sense explained in the theoretical paper [6]. In particular, plex can be viewed as lattice \textit{join} and crux as lattice \textit{meet}; in this way, the concept sum \( \varphi + \psi \) represents the most specific generalization of the concepts \( \varphi \) and \( \psi \) (lattice least upper bound), whereas the concept product \( \varphi \times \psi \) is the most general specialization (lattice greatest lower bound). In other proposals ([18] and [22]), the plural sum \( \ominus \), understood as aggregation, is also introduced.

---

\(^3\) The "-plex" noun combining form derives partly from Latin "-plex" (as in "duplex") and partly from "complex" [9]. It is akin to English "-fold" in the sense of "having (so many) parts" (ex.: "threefold aspect of the problem"). As for crux, it is used here both for its resemblance with "plus" and to avoid confusion with the "+" used in other context than concept algebra.

\(^4\) After the introduction of the axioms (section 1.3), every term can be reduced to an algebraically equivalent term of this form.

\(^5\) Denotations of constants can be overlapping after introduction of type definitions (see paragraph 1.5.2).
We know from the theory of lattices that every algebraic lattice is equivalent to an order theoretic lattice; in other words, every algebraic lattice has a lattice ordering on it. This establishes a link between the concept algebra described so far and the is-a hierarchies of concepts introduced in section 1.1. Mathematically:

\[ a \leq b \iff a = a \land b, \] or, equivalently,
\[ a \leq b \iff b = a \lor b, \]

where \( \land \) and \( \lor \) are lattice meet and, respectively, join; \( \leq \) is the lattice ordering and \( a \) and \( b \) range over all the elements in the lattice. In the concept algebra of discourse, this corresponds to:

\[ \text{isa}(\varphi, \psi) \iff \varphi = \varphi \times \psi, \] or, equivalently,
\[ \text{isa}(\varphi, \psi) \iff \psi = \varphi + \psi. \]

Let us now introduce the bounds of the concept lattice:

- \( T \) (*top*) as the universal concept (everything);
- \( \bot \) (*bottom*) as the null concept (nothing).

The concept lattice can thus be sketched as a Hasse diagram:

\begin{center}
\begin{tikzpicture}[->, thick, node distance=1.5cm, >=stealth, baseline=(current bounding box.center)]
  \node (T) {$\top$};
  \node (phi_plus_psi) at (2,1) {$\varphi + \psi$};
  \node (phi) at (1,0) {$\varphi$};
  \node (psi) at (3,0) {$\psi$};
  \node (phi_times_psi) at (2,-1) {$\varphi \times \psi$};
  \node (bot) at (2,-2) {$\bot$};

  \draw (T) -- (phi_plus_psi);
  \draw (phi_plus_psi) -- (phi);
  \draw (phi_plus_psi) -- (psi);
  \draw (phi) -- (phi_times_psi);
  \draw (psi) -- (phi_times_psi);
  \draw (phi_times_psi) -- (bot);
\end{tikzpicture}
\end{center}

*fig. 1.3.1.1*

The arrows of fig. 1.3.1.1 correspond to concept subsumption, therefore moving upwards means generalizing and moving downwards means specializing. Constants (individual concepts) are denoted by singleton sets and therefore their position in the concept lattice is "immediately above" bottom, as they do not subsume any other concept than bottom.

### 1.3.2 Axioms for Lattices

Next are the axioms that define the algebra of the distributive bounded lattices discussed so far. In the following, variables are used meaning every element in the lattice:

- **idempotency:** \( X + X = X \) and \( X \times X = X \)
- **commutativity:** \( X + Y = Y + X \) and \( X \times Y = Y \times X \)
• **associativity:** \( X + (Y + Z) = (X + Y) + Z \) and 
\[ X \times (Y \times Z) = (X \times Y) \times Z \]

• **absorption:** \( X = X + (X \times Y) \) and 
\[ X = X \times (X + Y) \]

• **distribution:** 
\[ X \times (Y + Z) = (X \times Y) + (X \times Z) \] or 
\[ X + (Y \times Z) = (X + Y) \times (X + Z) \]

• **bounds:** 
\[ X + \bot = X \quad X \times \bot = \bot \quad X + \top = \top \quad X \times \top = X \]

Concept algebra allows expressing relationships between concepts (classes or individuals) as equations (or inequalities in the form of *is-a* relationships) in a variable-free way. Let us clarify with an example. The first order predicate logic expression (with an implicit universal closure):

\[ \text{city}(X) \leftarrow X = \text{copenhagen} \]

can now be stated as the (variable-free) concept inclusion:

\[ \text{isa}(\text{copenhagen}, \text{city}) \]

Once again no distinction is formally made between classes (\textit{city}) and individuals (\textit{copenhagen}).

The double arrow \( \leftrightarrow \) of first order predicate logic obviously corresponds to concept algebra's equality viewed as *is-a* relationship in both ways:

\[ \text{human}(X) \leftrightarrow \text{male}(X) \lor \text{female}(X) \]

becomes

\[ \text{human} = \text{male} + \text{female} \]

One immediately notices that such expressions resemble type definitions. General expressions of type equations and type inequalities are shown in section 1.5.

### 1.3.3 Axioms for attributions

The rationale for having attributes, as it should be clear from the examples in section 1.1, is that they provide a means for forming tuples (records). The lack of tupling in ordinary lattices comes from the fact that the crux is neither Cartesian product nor conjunction of predicates; it is indeed a generalization of natural join (from the relational database algebra) that includes Cartesian product and conjunction as special cases. Further axioms are then needed to fit the attribution operation into the model of distributive bounded lattices:

• **annihilation:** \( a(\bot) = \bot \)

• **distribution of + over attribution:** \( a(X + Y) = a(X) + a(Y) \)

• **distribution of × over attribution:** \( a(X \times Y) = a(X \times a(Y) \)
The distribution of $\times$ over attribution makes attributions functional. This means that identical attributes merge their values (reading the axiom computationally from right to left); if they are incompatible, the whole frame reduces to bottom, according to annihilation. Distribution of $+$ over attribution naturally completes the scheme of concept algebra.

A direct result of the axioms is the monotonicity of attributions:

**Theorem:** Attributes are monotonic

$$X \leq Y \Rightarrow a(X) \leq a(Y)$$

**Proof:**

$$X \leq Y \iff X = X \times Y \text{ by definition of } \leq$$
$$\Rightarrow a(X) = a(X \times Y) \text{ by applying } a \text{ to both members}$$
$$\iff a(X) = a(X) \times a(Y) \text{ by distributivity of } \times \text{ over attribution}$$
$$\iff a(X) \leq a(Y) \text{ by definition of } \leq$$

End of the proof

One could also think of introducing the dual of annihilation, namely:

**totalization**:

$$a(T) = T$$

which can be useful in inductive concept formation. This axiom is however not strictly needed in **OBJECTLOG** and, furthermore, it gives rise to several problems, like non-uniqueness of the most general unifier (see chapter 2) and inconsistency in the set view of concept algebra.

Similar objects are described in [5], where explicit constructors are used for forming tuples and sets, which are constituted in the concept algebra by attributes and, respectively, sums. Furthermore, a formal distinction between individuals (atomic objects) and classes (complex objects) is made. Besides, both annihilation and totalization are used, but the boundary axioms behave exactly in the opposite way as concept algebra. Informally speaking, every object containing $T$ is equal to $T$ and every attribution not present in an object has $\bot$ as implicit value; therefore, having more attributes in a frame yields a more general object (in contrast with the intuition). This almost symmetric algebraic understanding presented in the paper by Bancilhon and Khosafian will be disregarded in the following.

There is an interpretation of attribution in the sense of lambda calculus that is introduced in several papers, for instance [17]. We shall now establish a correspondence between concepts and their denotation via the previously introduced [[ ]] function.

---

6 This name is just a proposal, as it derives from the latin "totum" intended as the dual of "nihil".
### 1.4 Interpretations of Concept Algebra

#### 1.4.1 Set Interpretation of Concept Algebra (Powerset Model)

The mapping from concepts to denotations (in a powerset model over the underlying mathematical universe $U$) can be done as follows. Bottom and top are by definition understood as the null concept (empty set) and, respectively, the universe:

$$[[\bot]] = \{\} \quad (1.4.1.1)$$

$$[[\top]] = U \quad (1.4.1.2)$$

The denotation of a constant is a set of members of $U$, that is, a member of $U^2$. The meet $\times$ operation for distributive lattices corresponds to intersection ($\cap$) of denotations; similarly lattice join $+$ corresponds to union ($\cup$):

$$[[\varphi \times \psi]] = [[\varphi]] \cap [[\psi]] \quad (1.4.1.3)$$

$$[[\varphi + \psi]] = [[\varphi]] \cup [[\psi]] \quad (1.4.1.4)$$

In general:

$$[[\prod_{i=1}^n \varphi_i]] = \bigcap_{i=1}^n [[\varphi_i]] \quad (1.4.1.5)$$

$$[[\sum_{i=1}^n \varphi_i]] = \bigcup_{i=1}^n [[\varphi_i]] \quad (1.4.1.6)$$

Lattice's partial ordering $\leq$ (isa in the concept algebra) clearly corresponds to set inclusion ($\subseteq$), whereas equality is set equivalence ($\equiv$):

$$[[\text{isa} (\varphi, \psi)]] = [[\varphi]] \subseteq [[\psi]] \quad (1.4.1.7)^7$$

$$[[\varphi = \psi]] = [[\varphi]] \equiv [[\psi]] \quad (1.4.1.8)^8$$

An attribute $a$ maps into a binary relation $g_a$, that is a member of $2^{U \times U}$, in the following way:

$$[[a(\varphi)]] = \{x \exists y (g_a (x, y) \land y \in [[\varphi]])\} \quad (1.4.1.9)$$

For a frame term (with no name) the following general correspondence is then stipulated:

$$[[\prod_{i=1}^n a_i (\varphi_i)]] = \left\{x \bigwedge_{i=1}^n (\exists y_i) (g_a (x, y_i) \land y_i \in [[\varphi_i]])\right\} \quad (1.4.1.10)$$

It can be shown that the above axioms are consistent with their set (or logical, if expressed in $\lambda$-calculus) interpretation. For instance, the interpretation of annihilation can be written in the following way:

$$[[a(\bot)]] = \{x (\exists y) (g_a (x, y) \land y \in [[\bot]])\} = \{x (\exists y) (g_a (x, y) \land y \in \{\})\}$$

$$= \{x (\exists y) (g_a (x, y) \land \text{False})\} = \{x \text{False}\} = \{\}\ = [[\bot]]$$

---

7 We are mapping in this case from operations on concepts to set operations and not from concepts to their denotations.

8 Notice that $\equiv$ takes precedence over $\equiv$. 
where the first step uses interpretation (1.4.1.9) with \( \varphi \) bound to bottom; the second exploits interpretation (1.4.1.1); steps 3 and 4 are simple logical rewritings and finally the last step uses (1.4.1.1) backwards.

Similar justifications can be given for all the other axioms, provided that functionality of attribution is assumed as from distribution of \( \times \) over attribution. The interested reader can refer to [17].

For the case of the totalization axiom a more restricted interpretation is needed, as it does not reduce in the expected way:

\[
[[a(T)]] = \{x(\exists y)(g_a(x,y) \land y \in [[T]])\} = \\
= \{x(\exists y)(g_a(x,y) \land \text{True})\} = \{x(\exists y)(g_a(x,y))\}
\]

A reasonable solution would be, for instance, to constrain all the \( g_a \)'s to be relations such that

\[
\forall x(\exists y)(g_a(x,y)) = \text{True}
\]

as the denotation of \( T \) is the universe:

\[
[[T]] = \bigcup = \{x|\text{True}\}
\]

1.4.2 **TOPOLOGICAL INTERPRETATION OF CONCEPT ALGEBRA**

Let us consider again the example frame term from paragraph 1.2.1. For the sake of simplicity, we shall disregard the frame name, so that the resulting frame is:

\[
\begin{array}{c}
\text{title: \textit{ulys}sses} \\
\text{author: \textit{joy}ce} \\
\text{year: 1922}
\end{array}
\]

(1.4.2.1)

It is unclear, at the moment, what the graphical representation of such a ground frame should be. In fact, in this case, unlike the example in paragraph 1.1.1 where we used Venn diagrams, we have attributions. Each attribute can be thought as a dimension (an axis of discreet elements) of the space of concepts. Let us forget for a moment the attribute \textit{year}. The unnamed frame derived from 1.4.2.1 would then be represented in a two-dimensional discreet space whose axes are \textit{title} and \textit{author}. The values of attributions are, in general, sets; in our case, \textit{ulys}sses and \textit{joy}ce are singleton sets, as they have a unique interpretation in the intended model. The pertinent scheme looks then as follows:
The circle in fig. 1.4.2.1 highlights the interpretation of the frame. If we now reintroduce the attribute *year*, a further axis must be included which enumerates all elements (years) of interest. The diagram changes to the one displayed in fig. 1.4.2.2:

![fig. 1.4.2.1](image1)

![fig. 1.4.2.2](image2)

An element on an axis individuates now a plane in a three-dimensional space; the intersection of all three planes (marked by a circle) gives the interpretation of the frame searched for. In the general case a frame is made of several attribute/value pairs, therefore the words "plane" and "space" are replaced by "hyper-plane" and, respectively, "hyper-space" when $n > 3$ in (1.2.1.1).

In example (1.4.2.1) all the constituents of the frame were singleton sets (inside or outside an attribution). In general, a concept is denoted by a set with one or more elements; we can consider, in this case, such a set as an interval on the axis it refers to\(^9\). The natural extension of a hyper-plane, when it comes to intervals, is a *degenerate* hyper-parallelepiped, meaning that it is infinite in all dimensions but one. We will though refer to cylinders instead of parallelepipeds as some authors in literature talk about *cylindrification*, as in [7]. A further example should clarify what explained so far.

Let us consider the following product of two frames:

$$
\begin{bmatrix}
  a_1 : x \\
  a_2 : y
\end{bmatrix}
\times
\begin{bmatrix}
  a_1 : z \\
  a_3 : t
\end{bmatrix}
$$

\(^9\) One can always think of grouping all the elements belonging to the denotation of a concept into an "interval" on the axis, which is of course discreet, so that the order does not matter.
which, of course, reduces to

\[
\begin{bmatrix}
  a_1 : x \times z \\
  a_2 : y \\
  a_3 : t 
\end{bmatrix}
\]  

(1.4.2.2)

thanks to the functionality of attribution. We have a three-dimensional conceptual space on which the concepts \(x, y, z\) and \(t\) individuate intervals. Each of the two frame factors corresponds to a cylinder on the space and each cylinder is infinite parallel to the axis nothing is said about. In this particular case, the topological view is representable "on paper" giving the diagram below:

![Diagram](#)

The intersection of the two cylinders (marked by the dashed lines in fig. 1.4.2.3) corresponds to frame (1.4.2.2).

The topological interpretation presented in this paragraph still needs to be refined. It is not clear, for instance, how a nested frame can be represented, that is whether we should consider an "inner" topological space each time we go inside an attribution or whether an attribution over another attribution just represents a further dimension in the (outmost and only) space

## 1.4.3 THE INVERSE IMAGE

In the powerset model described in paragraph 1.4.1, we associated an attribute with a binary relation over \(2^{\text{U} \times \text{U}}\). We can now extend this model in order to give a mathematical justification of the topological view presented above.

Let us associate, model-theoretically, attribute \(a\) with a function \(f_a : \text{U} \rightarrow \text{U}\) in virtue of the functionality of attributes. The denotation of \(a\) is then a (set) function \(2^\text{U} \rightarrow 2^\text{U}\) called inverse image and defined as follows:

\[
f_a^{-1}(Y) = \{x | f_a(x) \in Y\}
\]  

(1.4.3.1)
where \( Y \) is a set in \( 2^U \) and \( x \) is an element of \( U \). The denotation of an attribution is then:

\[
[[\alpha(\varphi)]] = f^{-1}_\alpha([[\varphi]]) = \{ x \mid f_\alpha(x) \in [[\varphi]] \}
\]

(1.4.3.2)

In other words, we set a mapping from the property (the \textit{quale}\textsuperscript{10}) and the things having the property (the \textit{quality}).

The correspondence set in paragraph 1.4.1 between:

- + and \( \cup \)
- \( \times \) and \( \cap \)
- \( \perp \) and \( \{ \} \)
- \( T \) and \( U \)

together with the inverse image yields a natural interpretation for the axioms for attribution:

\[
f^{-1}(\{ \}) = \{ x \mid f(x) \in \{ \} \} = \{ \}
\]

(1.4.3.3)

\[
f^{-1}(X \cup Y) = \{ x \mid f(x) \in (X \cup Y) \} = \{ x \mid f(x) \in X \} \cup \{ x \mid f(x) \in Y \} = f^{-1}(X) \cup f^{-1}(Y)
\]

(1.4.3.4)

\[
f^{-1}(X \cap Y) = \{ x \mid f(x) \in (X \cap Y) \} = \{ x \mid f(x) \in X \} \cap \{ x \mid f(x) \in Y \} = f^{-1}(X) \cap f^{-1}(Y)
\]

(1.4.3.5)

1.5 \textbf{ObjectLog combines Prolog and Concept Algebra}

1.5.1 \textbf{Concept algebraic terms in definite clauses}

The \textsc{ObjectLog} programming language merges Prolog's definite clauses with concept algebra by replacing ordinary Prolog terms with concept algebraic terms shown in the previous paragraphs.

Prolog's compound terms can be easily recast in a way shown in paragraph 1.2.2; however \textsc{ObjectLog} suggests a more abstract programming style in which \textit{is-a} relationships are used for establishing knowledge basis. \textit{Is-a} clauses are then part of the language and can appear in the body of ordinary clauses as well as alone for defining hierarchies in the underlying lattice (in the latter case they are indicated by \( \leq \)). Both ways \textit{is-a} clauses can be expressed with the shorthand \( =/2 \) predicate.

Unification of terms is thoroughly described in chapter 2, where it is reconsidered as equality in the sense of concept algebra and its accompanying axioms.

New categories of clauses, such as type definitions and built-in predicates, are outlined in the following.

\textsuperscript{10} The quale is a property (e.g. \textit{redness}) considered apart from things having the property (being red). In the attribution \textit{haircolor(red)}, \textit{red} is the quale and \textit{haircolor(red)} is the quality.
1.5.2 Type equations and inequalities

The variable-free expressions:

\[
\begin{align*}
\text{type}_\text{name} &= \text{frame}_\text{sum} \\
\text{type}_\text{name} &\leq \text{frame}_\text{sum} \\
\text{type}_\text{name} &\geq \text{frame}_\text{sum}
\end{align*}
\]  

(1.5.2.1)

where \textit{type\_name} is a constant and \textit{frame\_sum} is a sum of variable-free frames of the form (1.2.1.1), individuate a correspondence between the concept \textit{type\_name} and the complex object in the right-hand side of the equation (or inequality). This resembles type definitions of typed programming languages in that \textit{type\_name} is a concept that can subsume other concepts, found below it in the lattice, that inherit the properties of \textit{frame\_sum}; those concepts can therefore be considered of "type" \textit{type\_name}. In particular, we shall call an enumerative type definition an expression like (1.5.2.1) when the right-hand side is a sum of individual concepts (concepts denoted by a singleton set). Example:

\[
color = \text{red} + \text{green} + \text{blue} + \text{yellow} + \text{white} + \text{black} + \text{pink} + \ldots
\]

Notice that this is an extensional definition of the concept \textit{color}, as we enumerate all the elements corresponding to its denotation. This style is closer to the implementation level; however, there exists an intensional fashion as well for dealing with concepts (cf. e.g. [16]).

Type definitions can also be given by means of the \textit{is-a} relationship, though with a slightly different meaning. We could for instance re-express the above type definition in the following way:

\[
\begin{align*}
\text{red} &\leq \text{color} \\
\text{green} &\leq \text{color} \\
\text{blue} &\leq \text{color} \\
\ldots
\end{align*}
\]

In this way we state that red, green, blue etc. are colors, but we do not exclude the existence of other colors, in contrast with the definition with equality that was an exhaustive definition enumerating all possible colors.

More compactly, the latter style can be reformulated as

\[
\text{red} + \text{green} + \text{blue} + \ldots \leq \text{color}
\]

A few more examples should clarify what explained so far. Let us define a concept (type) \textit{person} whose attributes are the name, the date of birth and the sex. The definition looks as follows:

\[
\text{person} = \text{name(\top)} \times \text{birth(date)} \times \text{sex(male + female)}.
\]

\[\text{11}\] Two different aspects characterize concepts: the extension, which consists of all instances of the concept and the intension, which consists of the properties common to all instances of the concept. In other words, the intension contains the information that lets us recognize the individuals \textit{falling under} the concept. Notice that the \textit{is-a} relationship can be interpreted from an extensional set-theoretical (\(\subseteq\)) as well as from an intensional concept-theoretical (\(\preceq\)) viewpoint.
This is a complex expression stating that a person is a composition of a certain number of properties, namely the property of having a name, a date of birth and a sex. Consistently with the natural interpretation, the concept person is more specific than the concept name(T); in fact, objects can exist having a name but not being a person, for instance a firm. In the same way, birth(date) and sex(male + female) subsume person as well. In particular, the attribute name has values ranging over all the elements of the lattice structure, that is all elements subsumed by T^{12}.

Analogously, birth has values subsumed by the concept date and sex has values subsumed by the concept male + female. Notice that, interpreting male and female as individuals, the value of the attribute sex can be either male or female and nothing else; in other words, we condensed in a type definition an inner enumerative type definition for the sex of a person. Generally speaking, sex(male + female) is subsumed by sex(T) thanks to the property of monotonicity of attributions; if a more general concept of "living creatures" is to be introduced, which can in general be hermaphrodites (e.g. worms) or have no sex in the usual sense (e.g. monocellular organisms), this can be done with no effort, for instance by enriching the lattice in the following way:

\[ \text{living\_creature} \leq \text{sex}(T) \]

stating that a living creature has a sex, or, more precisely,

\[ \text{living\_creature} \leq \text{sex}(\text{male + female + hermaphrodite}^{13} + \text{none}) \]

which still subsumes sex(male + female) as

\[ \text{male + female + hermaphrodite + none} \geq \text{male + female} \]

The obvious consequence is that a person belongs then to the class of living creatures.

The date can in turn be specified as follows:

\[ \text{date} = \text{day}(T) \times \text{month}(T) \times \text{year}(T) \]

The same considerations hold for the use of T in this new definition.

At this point we might want to identify a subclass of person, namely the class of parents. This can be straightforwardly done with an inclusion relationship:

\[ \text{parent} \leq \text{person} \]

The fact of being a parent is not constrained, at the moment, to the fact of having a child as this can be specified at any time, for instance by means of a multiple inheritance. Our aim here is just to express the concept of parent as something more specific than the concept of person. Let us define now the concept of mother, in English, in the following way:

"a mother is a parent whose sex is female"

This precise definition can be reformulated in OBJECTLOG in a perfectly natural way:

\[ \text{mother} \leq \text{parent} \times \text{sex(female)} \]

\[^{12}\text{In the implementation of ObjectLog a few built-in concepts should be also introduced, such as strings, integers, real numbers etc. Instances of these concepts are individuals matching another built-in predicate called atomic. In this simple example no restrictions are made about the name of a person, though.}\]

\[^{13}\text{We understand here "hermaphrodite" as a sex different both from male and female. If we intended it as the conjunction of both, then we should use the "plural sum" instead: hermaphrodite} = \text{male} \oplus \text{female} \]
Analogously one defines a father as:

\[ \text{father} \leq \text{parent} \times \text{sex(male)} \]

Alternatively one could have split the concept of parent into mother and father in the following way:

\[ \text{parent} = \text{mother} + \text{father} \]

and then added further definitions like:

\[ \text{mother} \leq \text{sex(female)} \]
\[ \text{father} \leq \text{sex(male)} \]

The resulting lattice looks then as in fig. 1.5.2.1.

![fig. 1.5.2.1](image)

Notice that all the definitions given so far are compatible with one another and therefore need not be eliminated even if repeating a piece of information already given. In case of inconsistency, however, all the terms would collapse to one node, namely \( \bot = T \).

Let us assume now that the language includes some built-in concepts, such as numbers; one can easily define more specific concepts via inheritance. If for instance the concept of color defined accordingly to the red-green-blue palette scheme is needed, one could do it through the concept of \( \text{percentage} \) in the following way:

\[ \text{percentage} \leq \text{real} \]
\[ \text{rgb\_color} = \text{red(percentage)} \times \text{green(percentage)} \times \text{blue(percentage)} \]

where \( \text{real} \) is the built-in concept of real numbers. This definition could be refined for instance by means of multiple inheritance from a class of unit intervals (ranging from 0 to 1).

The predicate \( \text{atomic/1} \) is used to check if a concept is found immediately above the bottom of the lattice and cannot be used in the head of a clause. In other words, the user cannot force a concept to be atomic if, afterwards, a more specific concept found below it in the lattice is defined. If needed, it is always possible to assign some special properties to a concept by means of a user-defined predicate. The is-a relationship test can, however, be assigned a special meaning when checking inclusion in built-in concepts, so that instead of examining lattice inclusion, properties such as being a natural number or a string are tested.
1.5.3 RESTRICTIONS IN TYPE EQUATIONS

Let us define a human as either a male or a female:

\[ \text{human} = \text{male} + \text{female} \]

In this definition we classified humans depending on their sex. Another classification of interest could be the group of age, for instance:

\[ \text{human} = \text{child} + \text{adult} \]

One notices that the two definitions are perfectly compatible one another as the classes in the first definition are overlapping with the classes in the second definition. By summing up the two equalities, one gets the following:

\[ \text{human} + \text{human} = \text{male} + \text{female} + \text{child} + \text{adult} \]

which reduces with idempotency to:

\[ \text{human} = \text{male} + \text{female} + \text{child} + \text{adult} \quad (1.5.3.1) \]

In particular, the following sub-classes are of interest:

\[ \begin{align*}
\text{male} & = \text{boy} + \text{man} \\
\text{female} & = \text{girl} + \text{woman} \\
\text{child} & = \text{boy} + \text{girl} \\
\text{adult} & = \text{man} + \text{woman}
\end{align*} \]

and the first two definitions of human are replaced by the definitions of the underlying criteria:

\[ \begin{align*}
\text{sex} & = \text{male} + \text{female} \\
\text{age\_group} & = \text{child} + \text{adult}
\end{align*} \]

The corresponding definition of human with non-overlapping classes becomes:

\[ \text{human} = \text{boy} + \text{girl} + \text{man} + \text{woman} \quad (1.5.3.2) \]

The process of transformation from (1.5.3.1) to (1.5.3.2) via the definitions of the sub-classes is very complicated in the general case and therefore the following constraint is introduced when specifying types:

| every type definition must be unique |

This means that (1.5.3.1) and (1.5.3.2) are both acceptable definitions of human, but they cannot be given at the same time.

The need of uniqueness of type equations can be explained with another example. Let us consider the definition of an enumerative type, for instance:

\[ t = a + b + c \quad (1.5.3.3) \]

where \( a, b \) and \( c \) are atomic (and therefore non-overlapping) constants. If a new enumerative type equation for \( t \) is added, for instance:

\[ t = d + e \quad (1.5.3.4) \]

where \( d \) and \( e \) are also atomic constants, this should not make any sense, as \( t \) cannot be denoted at the same time by two disjoint sets, that is the set \( \{ a, b, c \} \) and the set \( \{ d, e \} \). The two definitions given would merge, however, into a new equation, namely:
\[ t = a + b + c + d + e \]

This results in the extension of the domain of type \( t \), which is clearly in contrast with the original definition (1.5.3.3), as, by definition of \( = \), the following relations (among others) hold at the same time:

\[
\begin{align*}
    t &\leq a + b + c \\
    t &\geq a + b + c + d + e
\end{align*}
\]

This is only possible if \( d + e \equiv \perp \), thus reducing (1.5.3.4) to

\[ t = \perp \]

It clearly results that \( a + b + c = \perp \) as the following relations (among others) now hold:

\[
\begin{align*}
    t &\geq a + b + c \\
    t &\leq \perp
\end{align*}
\]

By allowing at most one type equation for each type, this kind of inconsistency is avoided.

1.5.4 Defining Individuals and Querying

Let us consider the following definitions:

\[
\begin{align*}
    john's\_birthday &= \text{day}(7) \text{ month}(5) \\
    john &= \text{name}("john") \text{ sex(male) birth(john's\_birthday \times year(1965))}
\end{align*}
\]

The identifiers in the left-hand side of the equations meet the built-in predicate \( \text{atomic/1} \) because all the values inside attributions in the right-hand side do, being located "just above" \( \perp \). Let us assume that similar definitions are given for the persons \( \text{sarah, fred, linda} \) and \( \text{max} \).

Let us suppose that the following facts are defined for a predicate \( \text{taller/2} \) stating that the second argument is taller than the first:

\[
\begin{align*}
    \text{taller}(john, sarah) \\
    \text{taller}(john, fred) \\
    \text{taller}(fred, linda) \\
    \text{taller}(sarah, max)
\end{align*}
\]

plus an additional clause for transitivity:

\[ \text{taller}(X, Y) \leftarrow \text{taller}(X, Z), \text{taller}(Z, Y) \]

If we now want to inquiry the system in order to know the male persons taller than \( john \), this can be done with the following goal clause:

\[ \leftarrow \text{taller}(john, \text{sex(male)} \times X) \]

Notice that the order does not matter thanks to commutativity of \( \times \). The product term \( \varphi \times X \) obviates the call to \( \text{isa}(..., \varphi) \) since in concept algebra:

\[ \xi = \varphi \times X \iff \xi \leq \varphi \]

The answer would typically produce a list of bindings for the variable \( X \), namely:

\[ X = \text{name("fred") birth(...)} \]
\[ X = \text{name("max") \ birth(\ldots)} \]

If we are interested in the name only, the query can be reformulated as follows:
\[ \leftarrow \text{taller(john, sex(male) name(X) birth(_))} \]
where the anonymous variable is used in the same way as in Prolog and the variable \( X \) appears now inside an attribution. The answers become:
\[ X = "fred" \]
\[ X = "max" \]

### 1.5.5 Ordered and unordered typed lists

The list data type fits perfectly the kind of type definitions introduced in paragraph 1.5.2. Ordinary Prolog lists are expressed by means of a binary compound term, commonly represented as \( \text{.}/2 \), for composing lists element by element and with the empty list. Several syntactic sugars exist, namely square brackets ([]), commas (,) and pipes (|).

Following the scheme suggested in paragraph 1.2.2, a Prolog binary compound term can be mapped into OBJECTLOG to a term with two attributes, namely the head and the tail. A concept corresponding to the empty list (nil) needs also to be included in the definition, so that lists can be expressed as the CA concept:
\[ \text{list} = \text{nil} + \text{h}(\text{t(list)}) \]

A Prolog list \([t_1, t_2, \ldots, t_n]\) becomes then the nested frame term \( \text{h}(\varphi_1)t(\text{h}(\varphi_2)\ldots t(\text{h}(\varphi_n)t(\text{nil}))) \), where \( \varphi_i \) is the OBJECTLOG representation of \( t_i \). If syntactical variants \([\varphi|\psi] \) for \( \text{h}(\varphi)t(\psi) \), \([\phi] \) for \text{nil} and \([\varphi_0, \ldots, \varphi_n] \) for \( \text{h}(\varphi_0)t(\ldots t(\text{h}(\varphi_n)t(\text{nil}))) \) are introduced, then OBJECTLOG lists coincide syntactically and semantically with Prolog lists.

Clauses for membership are consequently expressed as:
\[ \text{member}(X, h(X) t(_)). \]
\[ \text{member}(X, h(_) t(Z)) \leftarrow \text{member}(X, Z). \]

The first clause can be replaced by another clause, as lists with a given head and a tail are more specific objects in the lattice than the object with just the given head, because, as recalled in paragraph 1.5.4, \( \text{isa}(L, h(X)) \leftrightarrow L = h(X) \times Y; \) so the first clause becomes
\[ \text{member}(X, L) \leftarrow \text{isa}(L, h(X)) \]

A more specific list whose elements are of a given type \( \tau \) (a \( \tau \text{list} \)) can be easily defined as:
\[ \tau \text{list} = \text{nil} + \text{h}(\tau)\text{t}(\tau \text{list}) \]
which sets the implicit relationship:
\[ \tau \text{list} \leq \text{list} \]
Unordered lists can be represented in yet another more attractive way, namely as concept algebraic sums. A table of entries \( e_1, \ldots, e_n \) of whatever kind may be formed as the concept sum
\[
e_1 + \cdots + e_n \quad (1.5.5.1)
\]
thereby reducing membership look-up to subsumption checking via \textit{is-a} clauses:
\[
is(a, e_1 + \cdots + e_n)
\]
Particularly interesting is the case of \( e_i \) taking the form of a frame term (1.2.3.1), because then the sum (1.5.5.1) represents an \( n \)-ary relational table. More on relational data base theory is said in chapter 3.

1.6 A Recapitulating Example

1.6.1 Good and Bad Dinners

In order to review in a practical example some of the ideas presented in this first chapter, we shall try in this section to model a typical menu that can be served in an Italian restaurant. For this purpose, we shall individuate categories of food and beverages and establish which combinations of such categories are good and which ones should be avoided.

An Italian meal basically consists of a first course (\textit{primo}, typically pasta) followed by a main course (\textit{secondo}); alternatively there is a unique course (\textit{piatto unico}), like \textit{lasagne} or pizza. The dinner is normally started with hors d’oeuvres (\textit{antipasti}) and concluded with a dessert; a dish is often accompanied by some vegetables or cheese (what is called in Italian a \textit{contorno}).

\[
\text{meal} = \text{fc(first course)} \times \text{mc(main course)} + \text{uc(unique course)}
\]
\[
\text{dinner} = \text{meal} \times \text{hd(hors d'oeuvre)} \times \text{vc(vegetables + cheese)} \times \text{d(dessert)}
\]
The basic constituents of a dinner are then arranged in a tuple formed with the attributions. A classification of food is therewith proposed by means of type equations and inequalities (names are translated to English wherever possible or accompanied with a little explanation in footnotes).

- \textit{hors d'oeuvre} = \textit{cold cuts} + \textit{bruschetta}\textsuperscript{14} + \textit{focaccine}\textsuperscript{15}
- \textit{bruschetta} = \textit{bruschetta with tomatoes} + \textit{bruschetta with garlic}
- \textit{first course} = \textit{pasta} + \textit{gnocchi}\textsuperscript{16} + \textit{rice}
- \textit{pasta} = \textit{dry pasta} + \textit{fresh pasta} + \textit{filled pasta} + \textit{baked pasta}
- \textit{dry pasta} = \textit{spaghetti} + \textit{penne}\textsuperscript{17} + \textit{fusilli}\textsuperscript{18}
- \textit{fresh pasta} = \textit{fettuccine}\textsuperscript{19} + \textit{spaghetti alla chitarra}\textsuperscript{20} + \textit{linguine}\textsuperscript{21}

\textsuperscript{14} \textit{bruschetta} = a savory toasted Italian bread.
\textsuperscript{15} \textit{focaccine} [pl. dim. of \textit{focaccia}] = a flat Italian bread typically seasoned with herbs and olive oil.
\textsuperscript{16} \textit{gnocchi} [pl. of \textit{gnocco}] = dumplings usually made with potato or semolina and served with sauce.
\textsuperscript{17} \textit{penne} [pl. of \textit{penna}, lit., quill, feather] = short thick diagonally cut tubular pasta.
\textsuperscript{18} \textit{fusilli} [pl. dim. of \textit{fuso}, spindle] = spiral-shaped pasta.
\textsuperscript{19} \textit{fettuccine} [pl. dim. of \textit{fetta}, slice] = pasta in the form of narrow ribbons.
filled pasta = ravioli\textsuperscript{22} + tortellini\textsuperscript{23}
baked pasta = lasagne\textsuperscript{24} + gratineed pasta
gnocchi = gnocchi with gorgonzola\textsuperscript{25} + gnocchi alla romana\textsuperscript{26}
spaghetti = spaghetti alla carbonara\textsuperscript{27} + spaghetti alla Bolognese\textsuperscript{28}
penne = penne with salmon + penne all'arrabbiata\textsuperscript{29}
rice = risotto\textsuperscript{30} with saffron + rice with pumpkin
ravioli with butter and sage ≤ ravioli
fettuccine with mushrooms ≤ fettuccine
linguine al pesto\textsuperscript{31} ≤ linguine

Notice that the three last definitions are expressed with inequalities because we make an open assumption about the types on the right-hand side, in that they can be afterwards enriched by means of other type inequalities. Type equations presuppose, of course, a closed assumption: for instance, the type of "spaghetti" only counts the types "carbonara" and "Bolognese" and no new kind can be added in the following.

main course = main course of meat + main course of fish
main course of meat = Milanese veal-cutlet + roasted veal + duck in orange
main course of fish = grilled dory + fish soupe + baked fish + fried fish

unique course = baked pasta + pizza
pizza = plain pizza + pizza with pepperoni + pizza with mushrooms

vegetables = backed vegetables + shrimp salad + mashed potatoes
cheese = Sardinian pecorino\textsuperscript{32} + fontina\textsuperscript{33} + parmesan\textsuperscript{34}

dessert = berry tart + tiramisù\textsuperscript{35} + sorbetto\textsuperscript{36} of strawberries

wine = white wine + red wine + rosé wine
white wine = dry white wine + sweet white wine
dry white wine = Soave del Veneto + Pinot bianco
sweet white wine = Marsala + Pinot Grigio
red wine = Barolo + Chianti + Grignolino
rosé wine = Cirò + Colli Altoatesini

\textsuperscript{20} spaghetti alla chitarra [fr. chitarra, guitar] = spaghetti made in thin solid squared strings.
\textsuperscript{21} linguine [pl. dim. of lingua, tongue] = narrow flat pasta.
\textsuperscript{22} ravioli [pl. of raviolo, lit., little turnip] = pasta in the form of little cases of dough containing a savory filling (as of meat or cheese).
\textsuperscript{23} tortellini [pl. of torta, cake] = pasta cut in rounds, folded around a filling (as of meat or cheese), formed into rings, and boiled.
\textsuperscript{24} lasagne = a baked dish consisting chiefly of layers of boiled pasta in the form of broad ribbons, with cheese, and a seasoned sauce of tomatoes and usually meat.
\textsuperscript{25} gorgonzola = a pungent blue cheese of Italian origin.
\textsuperscript{26} gnocchi alla romana [lit. in the Roman manner] = baked gnocchi with butter.
\textsuperscript{27} carbonara [lit., in the manner of a charcoal maker] = a dish of hot pasta into which other ingredients (as eggs, bacon or ham, and grated cheese) have been mixed.
\textsuperscript{28} spaghetti alla Bolognese = spaghetti in seasoned sauce of tomatoes and meat.
\textsuperscript{29} penne all'arrabbiata [lit., angry penne] = very pungent and spicy penne seasoned with hot peppers.
\textsuperscript{30} risotto = rice cooked in meat stock and seasoned (as with Parmesan cheese or saffron).
\textsuperscript{31} pesto = a sauce made especially of fresh basil, garlic, oil, pine nuts, and grated cheese.
\textsuperscript{32} pecorino [fr. pecora, sheep] = any of various cheeses of Italian origin made from sheep's milk.
\textsuperscript{33} fontina = a cheese that is semi-soft to hard in texture and mild to medium sharp in flavor.
\textsuperscript{34} parmesan = a very hard dry sharply flavored cheese that is sold grated or in wedges.
\textsuperscript{35} tiramisù [lit., pull me up] = a dessert made with ladyfingers, mascarpone, chocolate, and espresso.
\textsuperscript{36} sorbetto [akin to sherbet] = a fruit-flavored ice served as a dessert or between courses as a palate refresher.
Now we are able to define some general categories, like fish and meat:

\[
\begin{align*}
\text{fish} & \geq \text{shrimp salad} + \text{penne with salmon} + \text{main course of fish} \\
\text{meat} & \geq \text{spaghetti alla Bolognese} + \text{main course of meat} + \text{lasagne}
\end{align*}
\]

Good associations are normally fish with (possibly dry) white wine and meat with red wine. A dessert is best enjoyed with a sweet wine, whereas a rosé perfectly matches hors d'oeuvres. The first course is also commonly well suited for red or rosé wines. Having already defined the principal categories of food and wines, we can now link them with a predicate combine/2 establishing which compositions of dishes and beverages are particularly recommendable. The technique used in the following simply checks membership by means of isa clauses.

\[
\begin{align*}
\text{combine}(X, Y) & \leftarrow \text{isa}(X, \text{dry white wine}), \text{isa}(Y, \text{fish}) \\
\text{combine}(X, Y) & \leftarrow \text{isa}(X, \text{sweet white wine}), \text{isa}(Y, \text{dessert}) \\
\text{combine}(X, Y) & \leftarrow \text{isa}(X, \text{rosé wine}), \text{isa}(Y, \text{hors d'oeuvre + first course}) \\
\text{combine}(X, Y) & \leftarrow \text{isa}(X, \text{red wine}), \text{isa}(Y, \text{meat + first course})
\end{align*}
\]

So far we have determined a positive knowledge expressed by the predicate combine/2. In principle every pair not matching any of the clauses above should not be considered a good choice. However, it is also our intention to report what combinations are considered especially disagreeable. For instance, red wine and fish often collide. Furthermore, we suggest not to drastically change the style of the dinner between two courses: therefore one should avoid alternating fish and meat. We can then think of establishing such information with a predicate exclude/2.

\[
\begin{align*}
\text{exclude}(X, Y) & \leftarrow \text{isa}(X, \text{red wine}), \text{isa}(Y, \text{fish}) \\
\text{exclude}(X, Y) & \leftarrow \text{isa}(X, \text{first course} \times \text{fish}), \text{isa}(Y, \text{main course of meat})
\end{align*}
\]

It should be noticed that the category of "first course of fish" is simply obtained by cruxing the two general categories first course and fish; clearly, the product main course \times meat reduces to main course of meat.

### 1.6.2 Points of Discussion

The simple exercise proposed in the last paragraph could be reformulated in a number of different manners. For instance, the predicate combine can be expressed in a yet more compact way by packing together the is-a clauses with products with unbound variables:

\[
\text{combine}(\text{dry white wine} \times X, \text{fish} \times Y)
\]

yet better indicated by anonymous variables:

\[
\text{combine}(\text{dry white wine} \times _, \text{fish} \times _)
\]

A similar behavior is possible for exclude, for instance:

\[
\text{exclude}(\text{first course} \times \text{fish} \times _, \text{main course} \times \text{meat} \times _)
\]

Attributions have been used at the outmost level (that of meals and dinners) in order to make the tupling possible. The pertinent scheme is shown in fig. 1.6.2.1, with arrows going from the more specific element to the more general one:
"Inside" this level we inserted a flat classification of kinds of food with no attributions; in other words, the values enclosed between parenthesis are better defined by more specific concepts organized in a "taxonomy". As pointed out in section 1.4, there are problems in representing the whole scheme in a single sheet of paper, as we should have one spatial dimension for each attribution. Furthermore, the presence of sums and attributions introduces severe complications.

An important point is that in this formulation there is no information about the order of the courses in a dinner. This is the price one has to pay for having commutativity of attributions. There are basically two possible ways of stating this information:

- to organize all the components in a list;
- to establish knowledge about the order by means of a predicate, e.g.

\[ \text{precedes}(\text{first course} \times _, \text{main course} \times _) \]

The second solution does not require any changes in the definitions of \textit{meal} and \textit{dinner} and is perhaps the most convenient for this case.

Another possible variation in the development of the exercise concerns the understanding of the categories. For instance, it would have been perhaps more elegant to have very general categories, like \textit{white} and \textit{dry} and to use them for classifying atomic concepts:

\[ Soave \text{ del Veneto} + \text{Pinot bianco} \leq \text{dry} \times \text{white} \times \text{wine} \]

and similarly for \textit{red}, \textit{rosé} and \textit{sweet}. In this way sweet white wines and dry white wines are automatically white wines (and wines as well).

### 1.7 Summary of the chapter

Many ideas and definitions were introduced in the course of this chapter; therefore we shall give in this section a brief overview of the main notions presented so far, in order to reconsider them in the rest of this document in a broader perspective.

In section 1.1 we showed that OBJECTLOG is an extension of logic programming that exploits facilities common in object orientation, such as inheritance, expressed here as is-a relationships. The objects in question are concepts (descriptions) whose denotations are sets. Among other implications, OBJECTLOG is able to express relations between classes and makes no formal distinction between classes and individuals.

The complexity of the binary relational Peirce product is hidden behind attributions, which are easy-to-use algebraic facilities for forming tuples. Frames are defined in paragraph 1.2.1 as (possibly named) hierarchical tuples that cover Prolog's terms as a special case. General terms, in OBJECTLOG, also include variables and sums.
Lattices are introduced in section 1.3 and an operational correspondence is set between plex and least upper bound, and crux and greatest lower bound. Concepts are organized in a distributive bounded lattice whose bottom is $\bot$ and whose top is $T$. The concept algebra is governed by axioms, as explained in paragraphs 1.3.2 and 1.3.3, both for the underlying lattice and for the attributions. The axioms establish in which way terms can be combined (by means of sums, products and attributions) and in what cases they collapse to $\bot$. Furthermore, the axioms make the attributions functional and monotonic, as proved in paragraph 1.3.3.

Concepts can be interpreted in several ways. Paragraph 1.4.1 shows how concepts and their operations can be mapped over sets; for instance, $\bot$ corresponds to the empty set and $\times$ to set intersection. A comprehensive pattern is given for the general cases and a link to $\lambda$-calculus is also established. It is furthermore shown that this interpretation (the so-called powerset model) is consistent with all the axioms except for totalization, which requires further restrictions. Concepts can also be depicted in a topological view that extends ordinary Venn diagrams, as illustrated in paragraph 1.4.2. Another interpretation of attribution still complying with the axioms is given by the inverse image function defined in paragraph 1.4.3, which is an operation performing a mapping from qualia to qualities of concepts.

Section 1.5 introduces type equations and inequalities as a powerful tool for establishing knowledge and explains them by means of examples. Not carefully written type definitions easily lead to inconsistency; this is a good reason for constraining type definitions to be unique, which furthermore avoids very complicated algebraic transformations for reducing types to union of non-overlapping sets, as explained in paragraph 1.5.3. Paragraph 1.5.4 deals with individuals, that is concepts that are just above $\bot$ and match the predicate atomic, and a strategy for retrieving values (querying) is also presented. Finally lists are discussed and a syntactic sugar is proposed; in addition, more complicated data structures, such as typed lists and relational tables, are shortly described.

Section 1.6 develops a more complete example that carries out the techniques explained in the previous sections for modeling and classifying concepts. All the choices made are thoroughly discussed and possible alternatives are taken into account.
Chapter 2  Unification of complex objects

2.1  Unification in ObjectLog

In the first chapter we described OBJECTLOG as a powerful language dealing with complex objects (concepts), inheritance and lattices, special operators (crux and plex) and types. Implementationwise, such a framework, considered in its totality, appears to be too complicated for a first approach. We shall then take into account only subsets of OBJECTLOG, for which we shall describe a unification algorithm (in this chapter) and the design of a compiler (in chapter 4). We shall then start with a very limited subset called OBJECTLOG₁ and progressively add more features.

2.1.1  Restricting the Language

It will be shown in the following that having + and × at the same time gives rise to major problems concerning unification. OBJECTLOG₁ gets rid of this obstacle by cutting off the + from the language and by disregarding type definitions (which can have +, as shown in the first chapter). The resulting model brings to a clear semantics for unification (intended as an inclusion relationship in both ways) and is-a clauses but, of course, it does not include many of the interesting capabilities of the original language. Furthermore, it should be noticed that the objects are not organized in a lattice, as we do not have a least upper bound (plex). Still, it does make sense to have a clause for checking concept inclusion (is-a) because we can express inheritance by means of ×, as the object a × b is more specific than both a and b.

The (ground) objects we are dealing with in OBJECTLOG₁ are frames of the form 1.2.3.1, that is the most general expression one can write without + and without variables. It turns out, however, that all the constants represent atomic concepts, as we do not have type equations; therefore the product of two constants always reduces to ⊥ because of the assumption that atomic concepts are disjoint. The (ground) objects we can use in actual OBJECTLOG₁ programs are then of the form 1.2.1.1 (at most one constant outside attribution), but we will refer to the general form 1.2.3.1 in the unification algorithm for illustrative purposes.

We used the word "ground" between parenthesis in the last few lines because objects can also have variables. In particular, the frame form 1.2.3.1 can be enriched with variables giving the following expression:

\[
\prod_{j=0}^{m} c_j \times \prod_{i=1}^{n} a_i(\varphi_i) \times \prod_{k=0}^{p} X_k
\]

(2.1.1.1)

where the \( \varphi_i \)'s are recursively of form (2.1.1.1).

We will also show in paragraph 2.1.3 that having more than one variable is not strictly needed when performing unification and therefore we will put the (dynamic) constraint \( p \leq 1 \).
### 2.1.2 SOME REMARKS ABOUT UNIFICATION IN GENERAL

Let us forget for a moment (only in this paragraph) our restriction concerning the + and let us consider the following simple example of (full OBJECTLOG) unification:

\[ X \times b = a \times b \]  

(2.1.2.1)

Clearly, the substitution

\[ X / a \]

is the most obvious solution to (2.1.2.1), but there are also other solutions less evident at first glance. In particular, the law of idempotency ensures that

\[ X / a \times b \]

is a solution as well, as \( b \times b = b \). There are, though, other non-genuine substitutions fulfilling (2.1.2.1), which we will refer to as "spurious" solutions. In fig. 2.1.2.1 we represent \( a \) and \( b \) as sets in a Venn diagram; \( c \) is a set disjoint from both \( a \) and \( b \).

The substitution

\[ X / a + c \]

is a spurious solution in that we refer to something which is outside \( b \) and therefore annihilates in the product with \( b \). This argument can be generalized to all concepts, like \( c \), falling outside \( b \). In other words, more general solutions than the first two presented in the paragraph can be specified by introducing an intuitionistic negation in the sense of Heyting algebra defined as follows:

\[ \neg \varphi \text{ is the sum of all objects disjoint with } \varphi \]

The most general solution for this particular case can then be expressed with negation in the following way:

\[ X / a + \neg b \]

A firm point in this document is, though, that we do not want to cope with negation; OBJECTLOG, provides a first, yet primitive, means for disregarding those solutions by simply not allowing + in the language. Further issues concerning unification in the general case are given later on in the chapter. Now we shall have a closer look at unification for OBJECTLOG, programs.
2.1.3 AN ALGORITHM FOR UNIFICATION IN ObjectLog1

In order to illustrate how unification can be performed, we first need to think of a way of representing frames in a convenient way. We shall also assume that, dynamically, the two following constraints are respected:

- One of the two frames involved in unification is completely variable-free. In other words, one of the two frames is of form 1.2.3.1 and the other one is of form 2.1.1.1 (which is a generalization\(^{37}\) of 1.2.3.1). This constraint is also referred to as pure/ground constraint and can be forced to be static by assigning input/output modes to predicates. Under certain conditions, this allows replacing unification by iterated matching\(^{4}\).\(^{37}\)

- Not more that one (uninstantiated) variable figures in the outmost level of frames of form 2.1.1.1, and this constraint must hold recursively for frames inside attributions, that is \(p \leq 1\). That this is a reasonable assumption is shown down in the paragraph.

Having \(\times\) in the language also means that the axioms of idempotency, commutativity, associativity and distribution over attribution exist for such operator\(^{38}\). We can, however, express in a univocal way every frame, which we will refer to as canonical form, by:

- applying distribution of \(\times\) over attribution (operationally from right to left) wherever possible;
- applying idempotency (operationally from left to right) wherever possible, that is removing all duplicates;
- sorting all the factors according to a given principle, which disambiguates commutativity and associativity. For instance one can think of having all the constants first, then the attributions and finally the variables. Constants can for example be sorted alphabetically and so attributions and, recursively, frames inside them. Variables need no sorting, as there can be at most one.

Now we are ready to show how unification takes place. First of all, we assume that the terms that are to be unified have been brought to the canonical form described above and that products of disjoint constants are replaced by bottom, thus annihilating the whole frame. Furthermore, we imagine that the first term is ground and the second is non-ground, because if both are ground then unification trivially reduces to checking that the frames are identical. This means that they must have exactly all the same constants and the same attributions (inside which this principle is recursively applied), otherwise unification reports failure. The following cases are then possible:

- if the first frame is \(\bot\) then at least one of the variables present in the second frame must be bound to \(\bot\), which guarantees that it reduces to \(\bot\) by applying the boundary axioms and possibly annihilation. This is actually a critical case, because there might not be a unique most general unifier (mgu), as in

\[
\bot = c \times a(X) \times Y
\]

The term on the right-hand side respects the pure/ground constraint and has at most one variable in each level. Unification is obtained both for

\[
X / \bot
\]

and for

\[
Y / \bot
\]

\(^{37}\) From the point of view of notation, not in a hierarchical sense.

\(^{38}\) Of course, we have boundary axioms (two out of four) and annihilation as well.
In this specific circumstance we can still find an mgu provided that we interpret it in an algebraic sense (that is, not as a substitution variable / term). In particular, all the relevant substitutions are designated by the expression:
\[
X \times Y = \bot
\]

In general, if the first frame is \( \bot \), the mgu is represented by the product of all the variables in the second frame equaled to \( \bot \).

Another interpretation that still circumvents the problem of having more than one unifier is to interpret a substitution with \( \bot \) as a failure. This understanding of unification is going to be used at a higher level, where we deal with clauses (chapter 4) and accept unification of terms provided that they return non-bottom values; however, here we are only concerned with the mechanisms of unification and not with its interpretation.

For all the remaining cases we assume that the first frame is different from \( \bot \).

• If the second frame has a variable \( X \) in the outmost level, we compute the (ordered) set difference
\[
C_{12} = C_1 \setminus C_2
\]
where we indicated with \( C_i \) the (ordered) sequence of constants in the \( i \)-th frame. Notice that we require also that
\[
C_1 \supseteq C_2
\]
otherwise unification fails. Informally, \( C_{12} \) represents the frame in canonical form formed by the product of all the constants which are in the first frame and not in the second one. If the second frame has constant not present in the first one, unification fails.

Similarly, we compare \( A_1 \) and \( A_2 \), being \( A_i \) the (ordered) sequence of attributions in the \( i \)-th frame. The result of this comparison is either failure or an ordered sequence \( A_{12} \) of attributions, computed in a way that is explained below. At this point, if we did not incur in failure, we can return the following substitution:
\[
X / C°_{12} \times A°_{12}
\]
where \( C°_{12} \) and \( A°_{12} \) are the frames represented by \( C_{12} \) and, respectively, \( A_{12} \).

• if the second frame has no variable in the outmost level, we check that the previously described sequences \( C_1 \) and \( C_2 \) coincide (that is \( C_{12} \) is empty), otherwise unification fails. We then compare the sequences \( A_1 \) and \( A_2 \), and check that the returned sequence \( A_{12} \) is empty, otherwise unification fails. Notice that the comparison of \( A_1 \) and \( A_2 \) may give rise to substitutions for inner variables.

The set difference used for sets of constants is performed in the usual way and can also take advantage of the order we set on constants. Conversely, the comparison of sequences of attributions needs more attention, as frames can be recursively nested in attributions. In particular, the comparison takes place as follows:

• all the attributions present in the first frame and not in the second one are carried into \( A_{12} \) together with their values and no further analysis is needed for them;
• if there are any attributions in the second frames that do not appear in the first one, the comparison returns failure;
• if two equally-named attributions are found, we attempt unification of their values, which can result in inner variable / frame substitutions. If such unification fails, this means that there was no variable "immediately" inside the attribution (but there can be some in deeper levels). Therefore, the exceeding parts of the first value, which cannot be bound to a variable (since there are none) are added to the sequence \( A_{12} \) put
inside the attribution they come from. If the second value had exceeding parts than failure is returned. An example of this special case of unification is, for instance, the 6-th shown in table 2.1.4.1.

The strategy adopted for OBJECTLOG, as shown above, consists then of always retrieving the most general (with respect to \( \leq \)) solution, which is also the most intuitive one, so that in example 2.1.2.1 we get the result:

\[ X / a \]

Furthermore, it should be noticed that if two equally-named attributions are found, we always try first to unify their values and only if this unification fails, the residual bindings are passed to the outer variable (if any). In this way we can establish with no ambiguities what the substitutions should be for cases like 5 in table 2.1.4.1, where besides the obvious solution:

\[ X / c, Y / T \]

there is also\(^{39} \)

\[ X / T, Y / a(c) \]

Multiple solutions are then disregarded in this case by only considering the first one.

We can now conclude the paragraph by enunciating the following theorem:

**Theorem:** Unification of one ground frame and one non-ground frame succeeds with more than one variable if and only if it does with one.

A sketchy proof can be formulated as follows:

**If part (\( \Rightarrow \))**

It is sufficient to think that we can always bind all the variables, except one, to the top element and then unify by only considering the unbound variable, as it clearly results:

\[ X \times T \times T \times \ldots \times T = X \]

thanks to the boundary axioms.

**Only if part (\( \Leftarrow \))**

According to the procedure we described above, we can affirm that success or failure of unification only depends on the set of constants and attributions forming the two frames and on the presence (not the number) of an outer variable. Adding further variables cannot result in a successful unification for the cases where the non-ground frame has exceeding parts with respect to the ground frame.

\[ \square \text{End of the proof} \]

---

\(^{39} \) We claimed a few lines above that the unifier is the most general with respect to \( \leq \), but in this case we cannot say if the pair of substitutions \( X / c, Y / T \) is more or less general than the pair \( X / T, Y / a(c) \). We can, however, affirm that the first solution is more general for the outmost variable and therefore is the one that we choose, as the algorithm for unification assigns higher priority to outer levels.
2.1.4 Examples of unification

In the last paragraph we explained how unification is understood in OBJECTLOG1. We shall report a few examples in table 2.1.4.1 and, where necessary, comment them. The unifiers are indicated as a set of substitutions.

<table>
<thead>
<tr>
<th></th>
<th>ϕ</th>
<th>ψ</th>
<th>mgu</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
<td>c × X</td>
<td>{ X / T }</td>
</tr>
<tr>
<td>2</td>
<td>c₁ × c₂ × c₃</td>
<td>c₂ × X</td>
<td>{ X / c₁ × c₃ }</td>
</tr>
<tr>
<td>3</td>
<td>c₁ × c₂</td>
<td>c₂ × c₃ × X</td>
<td>failure</td>
</tr>
<tr>
<td>4</td>
<td>c₁ × c₂ × a(c₃)</td>
<td>c₂ × a(c₄) × X</td>
<td>failure</td>
</tr>
<tr>
<td>5</td>
<td>a(c)</td>
<td>a(X) × Y</td>
<td>{ X / c₁, Y / T }¹⁰</td>
</tr>
<tr>
<td>6</td>
<td>a(c₁ × c₂)</td>
<td>a(c₁) × X</td>
<td>{ X / a(c₂) }</td>
</tr>
<tr>
<td>7</td>
<td>c₁ × c₂ × a₁(c₃)</td>
<td>c₂ × a₂(c₄) × X</td>
<td>failure</td>
</tr>
<tr>
<td>8</td>
<td>c₁ × a₁(c₂ × c₃) × a₂(c₄)</td>
<td>a₁(c₃) × Y</td>
<td>{ X / c₂, Y / c₁ × a₂(c₄) }¹¹</td>
</tr>
<tr>
<td>9</td>
<td>⊥</td>
<td>c₁ × a(c₂ × X)</td>
<td>{ X / ⊥ }</td>
</tr>
<tr>
<td>10</td>
<td>⊥</td>
<td>a(X) × Y</td>
<td>{ X × Y = ⊥ }¹²</td>
</tr>
</tbody>
</table>

ϕ is the ground frame and ψ is the non-ground one; both are in canonical form. The first case shows that if the non-variable part of ψ coincides with ϕ, then the variable is bound to the universal value T. In case 2 the variable is substituted with the frame represented by set difference of the set of constants representing the two frames. In case 3, unification fails because of the constant c₃, which is in ψ but not in ϕ. Case 4 gives failure as well, because the constant c₄, which is in the attribution a in ψ is not found in the same attribution in ϕ, which shows that the principle is recursively applied. In the fifth example we show that unification is always first attempted between frames at the same level. In case 6, the same principle is applied, but unification between frames of the same level fails, and therefore the residual constant c₂ is unified with the variable (inside its attribution a). In case 7 we have failure because ψ has an attribution that ϕ does not have. Case 8 shows how the comparison of attributions locates the exceeding attributions. The

¹⁰ The substitution \( X / T, Y / a(c) \)
would be acceptable as well, according to the axioms. It is, however, not taken into account by the algorithm described in section 2.1.3 in order to have only one unifier and the most intuitive one (see also note 39). If we wanted to have a unique mgu, it should be written as the algebraic equality:

\[ a(c) = a(X) \times Y \]

which, on the other hand, coincides with the initial expression that we wanted to unify.

¹¹ The substitution \( X / T, Y / c₁ × a₁(c₂) × a₂(c₄) \)
is also possible. Consider then the algebraic mgu defined as:

\[ Y = c₁ × a₂(c₄) × Z, a₁(X) × Z = a₁(c₂) \]

where the attribution \( a₁(c₃) \) was "simplified" in both frames and the fictitious variable \( Z \) was introduced in order to reduce this case to the one commented in note 40.

¹² The equality symbol (=) used instead of the slash (/) indicates that this is not a substitution proper, but rather an algebraic expression.
ninth example results in the substitution of the variable with \( \bot \) as the whole frame annihilates if \( \bot \) is found at any place. In the last case we have unification if any of the variables is bound to \( \bot \), as explained in paragraph 2.1.3.

### 2.1.5 AN ALGORITHM FOR CONCEPT INCLUSION IN OBJECTLOG1

In this paragraph we are going to describe an algorithm implementing the is-a relationship. As for unification, we assume that both frame terms are in canonical form and that the two constraints of paragraph 2.1.3 are not violated. We shall, furthermore, distinguish three cases:

1) both frame terms are ground;
2) the first frame is ground and the second is non-ground;
3) the first frame is non-ground and the second is ground.

Notice that there is not a fourth cases where both frames are non-ground because this would correspond to violating the pure/ground constraint.

The first case merely corresponds to a test and therefore it can only fail or succeed; the last two cases give rise to substitutions (variables / frame values), which, as for unification, are the most general ones in the sense explained in note 39.

The first case can be dealt with by considering that, if the first term is different from \( \bot \), the is-a test succeeds if and only if the first term has at least all the constants and all the attributions as the second term. This can be checked for instance with a set difference on the sets of constants and a comparison of the sets of attributions, as described in paragraph 2.1.3. If the first term is \( \bot \), success is returned (nothing is more specialized than \( \bot \)).

The second case can be reduced to the first one if we consider that instantiating variables in a frame term means specializing it, as we only have \( \times \). Therefore we can ideally bind all the variables in the non-ground term to \( T \) and consequently make the failure or success of this case depend on the is-a test for the ground case (point 1). In fact, the second frame is now ground, having all the variables instantiated to a value. In this way we can make ourselves sure that we have the most general substitutions (nothing is more general than \( T \)). Notice that technically it could be possible to leave all the variable uninstantiated. However, that does not strictly correspond to having the most general substitutions, as later on in the execution of a program those variables could be instantiated to values lower than \( T \), thus possibly disobeying the is-a relationship in question.

The third case is more elaborated and we shall divide it in three sub-cases:

- if the second frame is \( \bot \), the inclusion relationship can only hold if at least one variable in the first frame is substituted with \( \bot \). As for unification, in this case there is more than one mgu if more than one uninstantiated variable is present in the first frame. Once again we can avoid this problem by interpreting instantiation to \( \bot \) as failure at the clausal level. For all the remaining cases we assume that the second frame is different from \( \bot \).
- if the first frame has a variable at the outmost level, there is always success, as such variable can be substituted with a value specific enough to make the first frame more specific than the second one. The substitution we are looking for is the most general and therefore we can proceed as follows. Indicating with \( C_i, A_i \) and \( C_j, A_j \) the sets and, respectively, the difference sets already described paragraph 2.1.3, we calculate \( C_{21} \).
with no requirement that $C_2 \supseteq C_1$. Similarly, we compare $A_2$ and $A_1$ as we did for unification, but with the variant that when recursively unifying frame terms, we use the is-a algorithm instead (and swap the values, as we calculated $A_{21}$ instead of $A_{12}$). All the variables in the first term that were not instantiated during the algorithm can be left unbound, as later instantiation will not affect the validity of the inclusion relationship.

- If there are variables only in inner levels, we check that $C_1 \supseteq C_2$ and similarly we compare $A_1$ and $A_2$ and check that there is no attribution in $A_2$ that does not exist in $A_1$ and recursively check is-a for values inside attributions. All the variables in the first term that were not instantiated can be left unbound, as in the last point. Notice that, when applying the pure/ground constraint, the principle of well-modedness requires that unification instantiates all the involved variables and therefore all the variables in this case should be instantiated to $T$. However, we are not going to uniformly use the pure/ground constraint in OBJECTLOG$_1$, as explained in the next paragraph and therefore well-modedness is disregarded.

In order to make things clearer, we shall show some examples of is-a in table 2.1.5.1. All the examples reporting success with unification succeed with is-a as well and yield the same variable substitutions, therefore such examples from table 2.1.4.1 are not mentioned again.

<table>
<thead>
<tr>
<th></th>
<th>$\varphi$</th>
<th>$\psi$</th>
<th>mgu for is-a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_1$</td>
<td>$c_2$</td>
<td>failure</td>
</tr>
<tr>
<td>2</td>
<td>$c_1 \times c_2$</td>
<td>$c_2$</td>
<td>success</td>
</tr>
<tr>
<td>3</td>
<td>$c_1 \times X$</td>
<td>$c_2$</td>
<td>${ X/ c_2 }$</td>
</tr>
<tr>
<td>4</td>
<td>$c_1 \times a(X)$</td>
<td>$c_2$</td>
<td>failure</td>
</tr>
<tr>
<td>5</td>
<td>$c_1 \times a(X) \times Y$</td>
<td>$a(c_2) \times c_1$</td>
<td>${ X/ c_2, Y/ c_3 }$</td>
</tr>
<tr>
<td>6</td>
<td>$c_1$</td>
<td>$c_2 \times X$</td>
<td>failure</td>
</tr>
<tr>
<td>7</td>
<td>$c_1 \times a_1(X \times a_2(c_2))$</td>
<td>$a_1(c_3)$</td>
<td>${ X/ c_3 }$</td>
</tr>
<tr>
<td>8</td>
<td>$c_1 \times a_1(c_2 \times c_3) \times a_2(c_4)$</td>
<td>$a_1(c_3) \times X \times Y$</td>
<td>${ X/ T, Y/ T }$</td>
</tr>
<tr>
<td>9</td>
<td>$c_1 \times a(c_2 \times X) \times Y$</td>
<td>$\perp$</td>
<td>${ X \times Y = \perp }$</td>
</tr>
<tr>
<td>10</td>
<td>$c_1 \times a_1(c_2 \times X) \times a_2(Y)$</td>
<td>$a_2(c_3)$</td>
<td>${ Y/ c_3 }$</td>
</tr>
</tbody>
</table>

**Table 2.1.5.1**

The first two examples are simple cases of is-a with both frames ground. Example 3 succeeds, whereas unification in such a case would fail. Case 4 fails because there is no variable in the outmost level and $\psi$ has one constant that is not in $\varphi$. Case 5 shows that the chosen substitutions are the ones that assign more general values to outer variables. Example 6 shows that is-a with $\varphi$ ground reduces to is-a for ground frames after removal of all variables in $\psi$ (and in this case it fails). Case 7 illustrates how is-a is recursively applied to inner values. Example 8 is like case 6, but in this case we have success and therefore all the variables are bound to $T$. Case 9 shows that the only way a frame can be less than or equal to $\perp$ is to annihilate or be bound to $\perp$. Finally, case 10 shows a succeeding is-a in which one variable remains uninstantiated.

### 2.1.6 Relaxing the Pure/Ground Constraint

One of the restrictions introduced in paragraph 2.1.3 can be partially removed without severely affecting the complexity of the previously presented algorithm for unification in OBJECTLOG$_1$. The aim of this paragraph is to explain in what cases the
If both frames to unify are non-ground, it is particularly interesting to consider the case where one of the two (let us assume the first one\(^{43}\)) is just a variable (say \(X\)). Let us indicate with \(\psi\) the second frame. In this specific situation the following sub-cases are possible:

- \(\psi\) is the same variable \(X\): nothing needs to be done;
- \(\psi\) is just a variable (say \(Y\)), but not the same as the first frame \((X)\): we set the variable substitution \(X / Y\), thus forcing the two frames to always be equal;
- \(X\) does not appear in \(\psi\). We add to the environment the substitution \(X / \psi\);
- \(X\) appears in \(\psi\) but does not (possibly recursively) occur in any of the attributions belonging to \(\psi\) (if any), that is to say, \(X\) appears in the outmost level of \(\psi\) and not elsewhere in \(\psi\). We (ideally) remove \(X\) from \(\psi\), thus yielding the auxiliary frame \(\psi'\), and add to the environment the substitution \(X / \psi'\). The equality of the two frames is guaranteed as the equation \(X = \psi\) can now be rewritten as:

\[
\psi' = \psi' \times \psi'
\]

which holds thanks to idempotency of \(\times\). Further variables in \(\psi'\) can be left unbound, as later instantiations of those variables do not affect the equality of \(X\) and \(\psi^44\).

- \(X\) appears in \(\psi\) and (possibly recursively) occurs in one or more of the attributions belonging to \(\psi\) the only possible solution is to set the substitution:

\[
X / \perp
\]

no matter what the other components of \(\psi\) are. If there are other variables in \(\psi\), they can be left unbound as later instantiations of those variables do not affect the equality of \(X\) and \(\psi\). This is a solution because \(\psi\) reduces to \(\perp\) in virtue of the annihilation and boundary axioms, as many times explained in the course of this document.

No other cases are possible and therefore this kind of unification never fails (unless in the mediated sense of \(\perp\) as failure), as there is always a substitution for \(X\) that makes \(\varphi\) equal to \(\psi\).

The last case is particularly interesting because it recalls the occur-check case well-known from Prolog. In example 5 of table 2.1.6.1, unification fails for Prolog systems with occur-check. On the other hand, Prolog systems usually disregard occur-check due to its computational complexity; the drawback is that when unification for a case like 5 is attempted an infinite loop is provoked, namely:

\[
X = a(X) \\
X = a(a(X)) \\
\vdots \\
X = a(a(a(...)))
\]

Furthermore, examples can be made of programs that result in unsoundness for systems that ignore the occur-check case (see for instance [23]). Other systems do not loop but

---

\(^{43}\) The order of the operands can be changed by virtue of the symmetry of unification (equality).

\(^{44}\) Notice that in this circumstance it is considered more general to leave a variable uninstantiated than to bind it to \(T\), as being uninstantiated means being potentially bound to any concept (included \(T\)). This is different from what we assumed, for instance, in the eighth example of table 2.1.5.1, because in that case there were instantiations of the variables that could have led to violation of the relation between the two frames (in this case an is-a relationship) and therefore we bound those variables to \(T\).
take advantage of a special notation for indicating infinite structures, that is they return an "infinite" unifier as solution:

\[ \{ X / a(\infty) \} \]

However, this measure requires generalization of the notions of term, substitution and unifier for the infinite case, which is not dealt with in classical logic.

In OBJECTLOG (and in this extended version of OBJECTLOG, as well) cyclic structures are therefore not prohibited as they give rise to the perfectly valid solution \( \{ X / \bot \} \).

<table>
<thead>
<tr>
<th>( \varphi )</th>
<th>( \psi )</th>
<th>mgu</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( X )</td>
<td>( X )</td>
</tr>
<tr>
<td>2</td>
<td>( X )</td>
<td>( Y )</td>
</tr>
<tr>
<td>3</td>
<td>( X )</td>
<td>( c \times a(Y) \times Z )</td>
</tr>
<tr>
<td>4</td>
<td>( X )</td>
<td>( c_1 \times a(c_2 \times Y) \times X )</td>
</tr>
<tr>
<td>5</td>
<td>( X )</td>
<td>( a(X) )</td>
</tr>
<tr>
<td>6</td>
<td>( X )</td>
<td>( c_1 \times a_1(c_3 \times X) \times X )</td>
</tr>
<tr>
<td>7</td>
<td>( X )</td>
<td>( c_1 \times a_1(c_2 \times X) \times a_2(Y) \times Z )</td>
</tr>
</tbody>
</table>

Table 2.1.6.1

In table 2.1.6.1 a few examples are reported showing what explained so far. Case 1 and case 2 deal with \( \psi \) being a variable. In case 3, \( X \) does not appear in \( \psi \). In case 4, \( X \) appears in \( \psi \) but does not occur in \( \psi \)'s attributions, and therefore we can take advantage of idempotency of \( \times \). Examples 5, 6 and 7 are occur-check cases. In 6 the variable appear more than once in \( \psi \) and in 7 \( \psi \) has other variables which remain uninstantiated.

2.1.7  CONCLUDING REMARKS

In paragraph 1.3.3 we described an axiom referred to as totalization and decided not to include it in OBJECTLOG because of problems with the interpretation of concept algebra (paragraph 1.4.1). Furthermore, there is a category of unification, not contemplated in table 2.1.4.1, that succeeds in a quite unnatural way by virtue of this axiom. Let us consider the case of unification where the frames are:

\[ \varphi = c \]
\[ \psi = a(X) \times Y \]

The totalization axiom makes this unification succeed with the following mgu:

\[ \{ X / T, Y / c \} \]

In other words, the attribute \( a \) can disappear if its value is instantiated to \( T \) so that unification can go on with only \( Y \) and \( c \). This behavior is not particularly intuitive, especially if attributions are regarded as a means for forming tuples in a relation, as shown in chapter 3. Such axiom is therefore discarded from our discourse in the rest of the document.

Classes can be connected by a relational product (Peirce product), introduced in OBJECTLOG as attribution. Such connections do not imply any inclusion relationships (unless differently stated by type definitions), that is, for any constant \( c \) and for any attribution \( a \) we can neither claim that
isa(c, a(c))
nor that

isa(a(c), c)

Therefore the is-a algorithm described in paragraph 2.1.5 fails in both cases. Different could be the case of recursive data structures, like:

\[ \text{list} = \text{nil} + h(T) \times t(\text{list}) \]

but in OBJECTLOG, no type definitions are allowed, therefore the problem is avoided.

### 2.2 Introducing type equations

In this section we shall propose an extension of OBJECTLOG, that incorporates type equations in a restricted form with respect to the one introduced in paragraph 1.5.2. We will refer to such language in the following as OBJECTLOG. Sums are allowed in type equations but not in clauses, so that OBJECTLOG terms are frames, augmented with the notion of type.

#### 2.2.1 Defining types

Three kinds of types can be defined with OBJECTLOG type equations:

- sum types;
- product types;
- attribution types.

Such type equations are respectively expressed in the following way:

\[
\begin{align*}
    t &= \sum_{i=1}^{n} t_i \\
    t &= \prod_{i=1}^{n} t_i \\
    t &= a(t_a)
\end{align*}
\]

where \( t, t_i \) and \( t_a \) are constants. \( t_i \) are either atomic constants or types (that can be also defined afterwards). \( t_a \) must be a defined type.

Furthermore, recursive types are prevented from our model by virtue of the following restrictions:

- no direct recursion is allowed;
- no indirect recursion can be defined.

Formally, these constraints mean that:

- all the \( t_i \)'s must be different from \( t \);
- none of the \( t_i \)'s is defined as a type depending on \( t \) or whose defining types depend (directly or indirectly) on \( t \). In other words, we check type dependencies for avoiding cycles in the definition of types.

On the contrary, \( t_a \) can depend on \( t \), or even be \( t \) itself, that is, recursion is allowed inside attribution (which permits defining recursive types as lists).
Not allowing direct recursion means that type inequations (which are not part of \textsc{ObjectLog}$_2$) cannot be defined by their algebraic counterpart as type equations, like in the (illegal) example:

\[
\text{person} = \text{person} + \text{parent}
\]

which would correspond to

\[
\text{person} \geq \text{parent}
\]

Indirect recursion is prevented for the same reason, because direct recursion is obtainable simply by substituting types with their definition. Let us consider the following equations:

\[
\begin{align*}
    a &= b + c \\
    b &= a + d
\end{align*}
\]

Clearly we have a case of indirect recursion, as \(a\) depends on \(b\) and vice versa. Let us try to replace instances of \(a\) and \(b\) in the right-hand side with their definitions. We get the following new set of equations:

\[
\begin{align*}
    a &= a + c + d \\
    b &= b + d + c
\end{align*}
\]

which evidently are directly recursive and therefore not allowed.

\subsection{Examples of Type Equations}

So far we have described what kind of type equations cannot be written. Now we shall give some examples of definitions accepted in \textsc{ObjectLog}$_2$.

Notice that the fact that the \(t_i's\) are constrained to be constants is just a syntactic limitation that does not prevent from defining complex types\footnote{The advantage is that the algorithm for checking of well-typedness is much simpler in this case than in the general case, in which, among other things, intermediate type definitions should be automatically generated.}. For instance, lists, that have sums, products and attributions in their definition, can be easily expressed with few intermediate steps in the following way:

\[
\begin{align*}
    \text{list} &= \text{head\_and\_tail} + \text{nil} \\
    \text{head\_and\_tail} &= \text{head} \times \text{tail} \\
    \text{head} &= h(\text{T}) \\
    \text{tail} &= t(\text{list})
\end{align*}
\]

"Flat" type equations can be defined with no effort, as for the following example (which we will refer to in the following as the "family" example):

\[
\begin{align*}
    \text{parent} &= \text{father} + \text{mother} \\
    \text{child} &= \text{son} + \text{daughter} \\
    \text{person} &= \text{parent} + \text{child} \\
    \text{male} &= \text{father} + \text{son} \\
    \text{female} &= \text{mother} + \text{daughter}
\end{align*}
\]

which corresponds to the following (lattice) structure:
The arrows in this example represent only the direct inclusion relationships expressed by the type equations. For instance, the types \textit{male} and \textit{female} are overlapping with the types \textit{parent} and \textit{child}; furthermore, it is easy to see that

\[ \text{male} + \text{female} = \text{parent} + \text{child} = \text{person} \]

Therefore, new arrows should be added in fig. 2.2.2.1 from \textit{male} to \textit{person} and from \textit{female} to \textit{person}. How this can be inferred is shown in the next paragraph.

\subsection{2.2.3 \hspace{1em} \textbf{AN ALGORITHM FOR GROUND ISA CHECK IN ObjectLog}_2}

It is now interesting to see how the restrictions we have described so far can be used for checking the inclusion relationship between two ObjectLog$_2$ terms in a clear way. For the sake of simplicity, we shall first analyze the case where both terms have no variables.

For every sum and product type equation we store clauses, called \textit{leq}, indicating direct inclusion between the defined type and the other types involved in the definition. In particular, for sum types defined as

\[ t = \sum_{i=1}^{n} t_i \]

the following set of \textit{leq} clauses is generated:

\begin{align*}
\text{leq}(t_1, t) \\
\text{leq}(t_2, t) \\
\ldots \\
\text{leq}(t_n, t)
\end{align*}

Similarly, for a product type definition:

\[ t = \prod_{i=1}^{n} t_i \]

we have the following \textit{leq} clauses:

\begin{align*}
\text{leq}(t, t_1) \\
\text{leq}(t, t_2) \\
\ldots \\
\text{leq}(t, t_n)
\end{align*}
We make the further assumption that every type equation (including attribution types) is available as fact of a special predicate eq/2.

Let us define the reflexive and transitive closure of leq as:

\[
\text{leq}^*(X, X) \\
\text{leq}^*(X, Y) \leftarrow \text{leq}(X, Z), \text{leq}^*(Z, Y)
\]

This predicate will be used for checking (possibly indirect) type inclusion with no inference on types.

We are now ready to define the clauses\(^\text{46}\) concerning the predicate isa for the ground case (we assume that + is available as a binary infix operator).

\[
\text{isa}(_, T). \\
\text{isa}(\bot, _). \\
\text{isa}(X, X). \\
\text{isa}(X_1 + X_2, Y) \leftarrow \text{isa}(X_1, Y) \land \text{isa}(X_2, Y). \\
\text{isa}(X, Y_1 + Y_2) \leftarrow \text{isa}(X, Y_1) \lor \text{isa}(X, Y_2). \\
\text{isa}(X, Y_1 \times Y_2) \leftarrow \text{isa}(X, Y_1) \land \text{isa}(X, Y_2). \\
\text{isa}(X_1 \times X_2, Y) \leftarrow \text{isa}(X_1, Y) \lor \text{isa}(X_2, Y). \\
\text{isa}(a(X), a(Y)) \leftarrow \text{isa}(X, Y). \text{ % monoticity of attribution} \\
\text{isa}(b(X), b(Y)) \leftarrow \text{isa}(X, Y). \\
\ldots \text{ % (for all attributions)} \\
\text{isa}(X, Y) \leftarrow \text{leq}^*(X, Y). \\
\text{isa}(X, Y) \leftarrow \text{eq}(X, Z) \land \text{isa}(Z, Y). \\
\text{isa}(X, Y) \leftarrow \text{eq}(Y, Z) \land \text{isa}(X, Z). \\
\]

Notice that the clauses after the ellipsis assume that either we have a check of the kind:

\[
\text{isa}(t_1, t_2)
\]

where \(t_1\) and \(t_2\) are constants (that is there is no sum and no product), in which case we reduce the clause to a \text{leq}^* check; or we simply got failure from all previous cases and the only possibility to make the test succeed is to replace a type by its definition (if any).

### 2.2.4 Examples of Ground Isa Check

Let us consider a few simple examples referring to the "family" example from the last paragraph. When checking:

\[
\text{isa}(\text{male}, \text{person})
\]

as there are no sums, no products and no attributions, we first try to see if \text{male} is directly or indirectly included in \text{person} by means of \text{leq}*, which is false as shown in fig. 2.2.2.1. We then find the type definition of \text{male} and reduce the check to:

\[
\text{isa}(\text{father} + \text{son}, \text{person})
\]

Now we have a sum and the check reduces to the conjunction of tests:

\[
\text{isa}(\text{father}, \text{person}) \land \text{isa}(\text{son}, \text{person})
\]

\(^46\) We will use a Prolog-like pseudo-code.
Both succeed in that they satisfy \( \text{leq}^* \), as a father is a parent and a parent is a person and, similarly, a son is a child and a child is a person.

The check:
\[
\text{isa(male, parent)}
\]
fails, as expected. If we proceed as in the previous example we reduce to the conjunction of is-a tests:
\[
\text{isa(father, parent)} \land \text{isa(son, parent)}
\]
which fails because a son is not a parent and this is because a son is neither a father nor a mother (according to the type equations).

The \( \text{leq}^* \) check is an essential means for preventing from infinite loops. If the third-to-last clause was replaced by:
\[
\text{isa}(X, Y) \leftarrow \text{leq}(X, Z) \land \text{isa}(Z, Y).
\]
we would still have the same positive answers, but, in some cases, instead of having failure we would go into an infinite loop. Let us reconsider the last example. When checking:
\[
\text{isa(son, parent)}
\]
we find that \( \text{leq}(\text{son}, \text{child}) \) holds and try the test
\[
\text{isa(child, parent)}
\]
which, in turn, reduces to
\[
\text{isa(person, parent)}
\]
as a child is a person. \( \text{person} \) is not \( \text{leq} \) than anything and therefore we use the type definition of \( \text{person} \) and try to resolve the test:
\[
\text{isa(parent + child, parent)}
\]
That a parent is a parent is immediately verified, but now we are back to the test
\[
\text{isa(child, parent)}
\]
which was used three steps earlier. Although we did not give a formal proof, the \( \text{leq}^* \) test prevents from this kind of infinite loops. On the other hand there are infinite cycles that cannot be avoided in this way. Let us consider the following definition of the type \( \text{family} \), which is supposed to accompany the previously given type equations from the "family" example and from lists.

\[
\text{family} = \text{nil} + \text{family\_head\_and\_tail} \\
\text{family\_head\_and\_tail} = \text{family\_head} \times \text{family\_tail} \\
\text{family\_member} = \text{parent} + \text{child} \\
\text{family\_head} = \text{h}(\text{family\_member}) \\
\text{family\_tail} = \text{t}(\text{family}).
\]

Clearly, \( \text{family} \) is a \( \text{list} \), as it is an instance of the \( \tau \text{list} \) type from paragraph 1.5.5, where \( \tau = \text{family\_member} \). However, if we check
\[
\text{isa(family, list)}
\]
we get an infinite loop. Intuitively this happens because we first expand the left term to the product of head and tail and then the right term as well, thus obtaining:
\[
\text{isa(family\_head} \times \text{family\_tail, head} \times \text{tail})
\]
which further reduces to
\[ \text{isa(family\_tail, tail)} \]
and finally to
\[ \text{isa(t(family), t(list))} \]
which is the same as the initial test by virtue of monotonicity of attributions. This is not surprising, nor are any special measures taken against this, as the is-a test is meant to check membership of an instance of a type to a given type, and not membership of a type to another type. In fact, tests made on instances of lists (that is finite terms) like
\[ \text{isa(h(c) \times t(nil), list)} \]
succeed with no problems.

### 2.2.5 Definition of atomicity

An assumption was made in the first chapter of denotations of concepts being disjoint unless differently stated by type equations or inequalities. This presupposition came along with the notion of atomicity of a concept, which we shall clarify in this paragraph. A concept is said to be atomic if it is above no other constant concept in the lattice except for \( \perp \). With constant concept we mean a concept that does not have any products, sums or attributions.

This statement can be made more precise by elucidating its implications with type definitions. First of all, we reduce our analysis to ground terms, as non-ground terms have variables that can be instantiated with any non-atomic values, thus rendering the whole term non-atomic. Then the following cases are considered:

- \( \perp \) is atomic;
- \( \varphi_1 \times \varphi_2 \) is atomic if both \( \varphi_1 \) and \( \varphi_2 \) are atomic\(^{47}\);
- \( a(\varphi) \) is atomic if \( \varphi \) is atomic, where \( a \) stands for any attribution;
- \( \varphi_1 + \varphi_2 \) is not atomic because it is above both \( \varphi_1 \) and \( \varphi_2 \);
- a constant \( c \) is atomic if it is not a defined type, that is if it does not appear in the left-hand side of a type definition. Otherwise, it is replaced by its definition and atomicity is checked on it.

Examples. In the "family" example from paragraph 2.2.2, the constants \( \text{mother} \) and \( \text{nil} \) are atomic as they do not appear in the left-hand side of a type equation. The product of attributions over atomic constants:
\[ h(\text{mother}) \times t(\text{nil}) \]
is atomic as well. Clearly, the constant \( \text{list} \) is not atomic, as it is a sum of concepts. Therefore the term
\[ h(c) \times t(\text{list}) \]
which indicates the type of all lists whose head is the constant \( c \), is not atomic.

\(^{47}\) Strictly speaking, \( \varphi_1 \times \varphi_2, \varphi_1, \) and \( \varphi_2 \) can be atomic in the sense of lattices only if \( \varphi_1 = \varphi_2 = \perp \). However, no concepts would be "permanently" atomic as long as one can multiply them by other concepts, thus rendering them more specific. The definition given here is therefore slightly different from that of lattice atomicity.
Atomicity is important for recognizing what concepts can be accepted as individuals by looking at the lattice. Care has to be taken with such a notion, as it could be misleading in some cases. For instance, atomic concepts still can designate classes even if no instances of such classes are defined by means of enumerative type equations. In the limit case, if no type definitions are present and no clauses are written, the lattice reduces to the scheme of fig. 2.2.5.1:

![fig. 2.2.5.1](image)

and therefore $T$ itself is atomic, even though it denotes the universe. Classes and individuals are not precisely distinct in OBJECTLOG, therefore the test for atomicity can basically help recognizing defined types from other concepts and discarding non-atomic concepts from being individuals.

### 2.2.6 Simplest cases of unification with types

Unification of ground terms is equivalent to ground is-a test applied in both senses. In clauses:

$$\text{unify}(\varphi, \psi) \leftarrow \text{isa}(\varphi, \psi) \land \text{isa}(\psi, \varphi). \quad \% \text{for the ground case}$$

This case can be important for unification of an atom with the head of a clause: for instance, the goal

$$\leftarrow p(\text{father} + \text{mother})$$

unifies with the head of the clause

$$p(\text{parent}) \leftarrow \ldots$$

if the type equations of the "family" example are given.

Another case that can be straightforwardly implemented in terms of ground is-a test is the one of is-a test with the first argument ground and the second argument non-ground. As we did in paragraph 2.1.5, this case can be reduced to ground is-a test by binding all the variables in the second term to $T$, because instantiating a variable in a frame term means specializing it and $T$ is the most general value in the lattice. Notice that is-a tests where the first argument is non-ground will not be considered for OBJECTLOG, except for the trivial case where the second argument is $\bot$.

More interesting is the case of unification when one of the two terms is non-ground (the pure/ground constraint is kept). In this paragraph we shall only deal with the case of non-ground terms having one variable in the outmost level and no variables elsewhere; the extension to the general case with one variable for each level is presented in the next paragraph.

In order to make our argument clear, we shall first look at some examples and then explain the algorithm performing unification for this simplified case. We will refer to the
ground term as $\varphi$ and to the non-ground term as $X \times \psi$, where $X$ is the variable and $\psi$ is the ground part of the second term in unification:

$$\text{unify}(\varphi, \psi \times X)$$

Let us consider the following set of type equations:

$$a = b \times c$$
$$d = c + e$$
$$f = g(h)$$

so that we have a product, a sum and an attribution type equation. The fundamental assumption about different constants being disjoint (unless differently stated by type equations) leads to the following Venn diagram of denotations, understood as sets, of the concepts mentioned in the type definitions:

![Venn Diagram](image)

fig. 2.2.6.1

Let us consider unification

$$\text{unify}(e, d \times X)$$

The algorithm proposed in paragraph 2.1.3 fails because the second term has a constant not present in the first one. However, we expect success from this unification, as

$$d \geq e$$

which is inferred from the second type equation in the set. The substitution

$$X / e$$

is a solution, as $d \times e = e$. The most general solution for this unification can be expressed, as recalled in the last section, with negation in the following way:

$$X / e + \neg d$$

Quantitatively, $-d$ in this case can be written as union of set differences. In particular:

$$-d = b \setminus d + f \setminus d = b \setminus a + f \setminus d$$

The problem with this approach is that we have to cope either with negation or with set difference. However, if we consider for a moment only the concepts (sets) involved in this unification, that is $d$, $e$ and $c$, it turns out that the most general solution, locally to these sets, is $e$ and no negation nor set difference is needed.
In the previous example the solution found at first was also (locally) the most general. In
\[ \text{unify}(a, d \times X) \]
this is also true, but the solution found with this technique is perhaps not the most intuitive one. As in the last case we have that
\[ d \geq a \]
and therefore \( X / a \) is a solution. The most general solution is, once again,
\[ X / a + \neg d = b + f \setminus d \]
Locally to \( a \) and \( d \), \( X / a \) is the most general solution, but if we consider the defining concepts of \( a \), that is \( b \) and \( c \), the most general solution for this unification turns out to be \( b \). As a matter of fact the second term unwinds as follows:
\[ d \times b = (c + e) \times b = c \times b + e \times b = a + \perp = a \]
which shows that \( X / b \) is a solution.

As last case, let us consider unification:
\[ \text{unify}(f \times a, g(h) \times b \times X) \]
It still holds that the ground part of the second term is more general than the first term, as shown in fig. 2.1.2.1, that is:
\[ g(h) \times b \geq f \times a \]
and therefore \( X / f \times a \) is a solution. The most general solution
\[ X / f \times a + \neg g(h) + b = f \times a + (c \setminus (f \times a)) + (f \setminus b) + (b \setminus f) + e = c + e + ((b + f) \setminus (f \times b)) \]
shows that, locally to the involved constants and their defining concepts (that is \( a, b, c \) and \( f \)) there is at least the substitution
\[ X / c \]
which is more general than \( X / f \times a \). In particular, this is the most general solution that can be given without using set difference (and sums\(^{48} \)) explicitly. Furthermore, such substitution is the most intuitive one, because if we replace \( f \) and \( a \) by their definitions, unification reduces to:
\[ \text{unify}(g(h) \times b \times c, g(h) \times b \times X) \]
which would generate the solution \( X / c \) if attempted with the algorithm of paragraph 2.1.3.

In all three cases, the wanted solutions (that is the most general ones that can be expressed without set difference) were at least as general as the first solution (corresponding to the first term) and, obviously, at most as general as the most general solution:
\[
\begin{align*}
a &\leq b \leq a + \neg d \\
e &\leq e \leq e + \neg d \\
f \times a &\leq c \leq f \times a + \neg (g(h) \times b)
\end{align*}
\]

\(^{48}\) We do not allow sums in solutions because such solutions could be afterwards combined with variables when further unifying, which would have the same effect as having sums in ObjectLog\(_2\) programs, thus contradicting one major hypothesis of such language.
We have available set intersection as × and inclusion relationship as ground is-a test from paragraph 2.2.3; the ¬ operator is, however, missing. The second inequality in the three cases can be respectively rewritten as:

\[
\begin{align*}
  b \times d &\leq a \\
  e \times d &\leq e \\
  c \times (g(h) \times b) &\leq f \times a
\end{align*}
\]

that is

\[
\text{wanted solution} \times \psi \leq \varphi
\]

which is the unification criterion we started from, where \(X\) has been replaced by the wanted solution and with \(\leq\) instead of \(=\). Furthermore, the preliminary condition

\[
\varphi \leq \psi
\]

must hold.

At this point we are ready for showing the algorithm that allows retrieving the solutions discussed so far.

- if \(\varphi \leq \psi\) does not hold, unification fails. In the following we assume that this condition is true.
- We remove from \(\varphi\) all components that are more general than \(\psi\). The reason for doing this is that they are of no use for the solution, as cruxing them with \(\psi\) would still yield \(\psi\). In particular, in the limit case where \(\psi \leq \varphi\), the solution is \(X / T\). Let us indicate with \(\varphi'\) the new version of \(\varphi\) without useless components.
- \(\varphi'\) is the solution searched for, unless a more general solution can be found, which can be checked in the following way: for each component \(\varphi'_i\) of \(\varphi'\) we check if it is specific enough to be a solution, that is we check if

\[
\varphi'_i \times \psi \leq \varphi
\]

If so, all other components of \(\varphi\) can be disregarded and we can also try to seek even more general solutions by replacing \(\varphi'_i\) with its type definition (if any), provided that it was a product type equation, and unification is now attempted between \(\varphi'_i\) and \(\psi\). If such unification results in failure, then \(X / \varphi'_i\) is the solution searched for. If the is-a test did not hold for any \(\varphi'_i\), then \(X / \varphi\) is the solution.

Notice that the assumption about different concepts being disjoint is fundamental. For instance, in unification between \(a\) and \(d \times X\) the solution would not be \(X / b\) if \(e\) and \(b\) were not disjoint: the product between \(b\) and \(d\) would generate the concept \(b \times e + a\), which is more general than \(a\) and therefore the solution would be the original \(X / a\). As a matter of fact, in the algorithm shown above, \(c\) would not be removed from the original solution \(X / a = b \times c\) because the relation

\[
b \times d \leq a
\]

would not hold anymore.

### 2.2.7 UNIFICATION FOR ObjectLog₂ IN THE GENERAL CASE

The consideration made in paragraph 2.1.6 about the relaxation of the pure/ground constraint still holds and therefore the same technique can be used with no
problems. The method described in the last paragraph can be generalized as follows (we still refer to unification of $\varphi$ and $\psi$, where $\varphi$ is ground and $\psi$ is non-ground):

- both $\varphi$ and $\psi$ are simplified by removing redundant components. This means that we first expand a term wherever possible by replacing its components with their product or attribute type equations. If after the expansion we have, for instance,

$$\varphi = \varphi_1 \times \varphi_2 \times \ldots \times \varphi_n$$

then we check if $\varphi_1 \geq \varphi_2 \times \ldots \times \varphi_n$, in which case we remove $\varphi_1$ from $\varphi$ and continue the test on the remaining components. This test is propagated inside attributions.

- a term $\psi'$ is created, which is the same as $\psi$ but with no variable. For example, if

$$\psi = c \times X \times a(d \times Y) \times b(Z)$$

then

$$\psi' = c \times a(d) \times b(T)$$

- if $\varphi \leq \psi'$ does not hold, then unification fails; otherwise we can go on with the next steps;

- all useless components of $\varphi$ are removed, giving the term $\varphi'$. With useless we mean components of $\varphi$ such that they are $\geq \psi'$. Each component of $\varphi$ is expanded with its product or attribute type definition (if any) and the removal is then recursively applied.

- a term $\psi''$ is created, which contains only the variable parts of $\psi$. For instance, in the example above we would have

$$\psi'' = X \times a(Y) \times b(Z)$$

- if any of the components of $\varphi'$ is an attribute type, it is expanded with its type equation, giving the term $\varphi''$;

- OBJECTLOG1 unification is then performed between $\varphi''$ and $\psi''$.

As for OBJECTLOG1, unification is first attempted inside equally-named attributions and, in case of failure, outer variables are involved.

In the "family" example

$$\text{unify}(fht, X \times t(Y))$$

results then in the pair of substitutions

$$X \mapsto h(family \_\_member)$$

$$Y \mapsto family$$

which is the most general solution in the sense explained in paragraph 2.1.3.

### 2.3 Towards ObjectLog

#### 2.3.1 Including type inequalities

Having prevented circular type definitions, type inequalities cannot be written in their algebraic reformulation. However, it is possible to deal with inequalities of the kind:

$$t \leq \prod_{j=1}^{n} t_j$$
where \( t \) and \( t_i \) are constants, as the following set of \( \leq \) clauses is generated:

\[
\leq(t, t_1) \\
\leq(t, t_2) \\
\ldots \\
\leq(t, t_n)
\]

Problems arise with sum type inequalities, like

\[
t \leq \sum_{i=1}^{n} t_i
\]

because no \( \leq \) relationships can in principle be stipulated between \( t \) and any single \( t_i \). Therefore the general problem might be difficult to handle, and probably new algorithms for is-a check and unification need to be designed.

### 2.3.2 Sums in the Program

What is really missing in OBJECTLOG\(_2\) is the possibility of writing unrestricted complex term in definite clauses. Furthermore, the constraints on unification carried along the chapter need to be completely relaxed in order to get the full potential of OBJECTLOG. Without such prerequisites, examples such as the table look-up described in paragraph 1.5.5 cannot run.

The approach we suggest for unification in the very general case of OBJECTLOG programs is the following. The terms involved in unification should be first expanded accordingly to the type equations defining their constituents (including sum type equations). Then each term should be brought to a (convenient) canonic form for sums and products and then simplified in a way similar to that explained in the last section for OBJECTLOG\(_2\) terms. Several solutions are possible. For instance one could think of always having sums of products, so that the term

\[
a + b \times (c + d) + e
\]

would be transformed into

\[
a + b \times c + b \times d + e
\]

and the term containing attributions

\[
a \times b(c + d + e(g + h))
\]

would become

\[
a \times b(c) + a \times b(d) + a \times b(e(g)) + a \times b(e(h))
\]

or, alternatively, itself, if we decided to apply this principle recursively inside attributions.

Once a canonic form has been fixed, unification can be performed by exploiting the order set on the elements in the sum and in the products. We believe that cases complying with the pure/ground constraint can be dealt with efficiently and with a little increase in the algorithmic effort. On the other hand, removing the pure/ground constraint could provoke a severe growth of the complexity and make the problem practically unmanageable. At that point, it might be more sensible to individuate classes of problems useful in practice (such as the table look-up for flat databases) and write ad hoc algorithms for unification in those cases, disregarding the general problem. Furthermore, the issue of
unification with type inequalities still has to be addressed, as discussed in the previous paragraph.

It should be noticed at this point that achieving the fully featured OBJECTLOG was outside the scope of this document. Instead, it was our purpose to identify and implement manageable subsets of the language, which is, as we believe, what has been done with OBJECTLOG₁ and OBJECTLOG₂.
Chapter 3  Links to related theories

3.1  Similar proposals

In this section we shall describe three logic programming packages that deal with object orientation and frames (or notions of the same kind): Wisdom, LIFE and LOT. For each of them we give a brief description and we try to point out similarities and differences with OBJECTLOG and, possibly, advantages and disadvantages. Some of the examples presented in the first chapter are re-discussed, where relevant, in a broader view; also, for significant aspects of each of the three tools, a reformulation in OBJECTLOG is proposed and commented.

3.1.1  WISDOM

The main intention of Wisdom, as described in [14], is that of filling the gap between the declarative and the procedural semantics of Prolog. Prolog carries along a number of limitations, in particular:

- it is *incomplete*, in that it can lead to infinite loops, because of the intrinsic limitations of a depth-first search;
- it is *unfair*, as the depth-first search strategy is not always able to find all the solutions;
- it is *unsound* for Prolog systems that do not perform the "occur-check" test;
- it uses *negation as failure*, also known as *closed world assumption*;
- it needs *control information* like *cut*, which can destroy the logic of a program;
- it needs *extra logical features*, like *assert/1*, which are outside the scope of first order predicate logic;
- it is not completely *multi-directional*, in that not all the arguments of a predicate can be used both as input and output.

Another issue is that it can be rather difficult to establish models of knowledge, because Prolog does not provide any guidelines for that purpose. Concerning this problem, Wisdom adheres to the requirements of the functional theory of the nature of knowledge. The functional approach is characterized by the *goal-oriented selection principle* (which consists of choosing one of the possible aspects of a concept depending on the *function* it is used for) and by the notion of *functional equivalence* (which considers equivalent two concepts that are even quite different but whose attributes perform the same function).

Among Wisdom’s highlights are *decision tables* and *frames*. An example of decision table is given below:

<table>
<thead>
<tr>
<th>Suitable car</th>
<th>Prize</th>
<th>≤20000</th>
<th>&gt;20000, ≤25000</th>
<th>&gt;25000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Sun roof</td>
<td>-</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>C2</td>
<td>Doors</td>
<td>-</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>C3</td>
<td>A1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R4</td>
<td>R5</td>
<td></td>
</tr>
</tbody>
</table>

*table 3.1.1.1*
Decision tables are clearly goal-oriented and self-explanatory; they can also be linked together to give a system of decision tables. Frames are non-goal-oriented tools used to represent structured objects that can be organized in hierarchies and have instances at the bottom of the hierarchy, like in the following example:

<table>
<thead>
<tr>
<th>car</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>price</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.1.1.2**

<table>
<thead>
<tr>
<th>middle-class car</th>
<th>IS-A Car</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>price</td>
<td>-</td>
</tr>
<tr>
<td>S2</td>
<td>sun roof</td>
<td>-</td>
</tr>
<tr>
<td>S3</td>
<td>doors</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table 3.1.1.3**

<table>
<thead>
<tr>
<th>my car</th>
<th>IS-AN-INSTANCE of middle-class car</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>price 25000</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>sun roof yes</td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>doors 5</td>
<td></td>
</tr>
</tbody>
</table>

**Table 3.1.1.4**

Between the frames car and middle-class car there is an is-a relationship (the price is inherited) and, finally, my car is an instance of a middle-class car. Frames have an obvious reformulation in OBJECTLOG, for instance:

\[
\begin{align*}
car &= \text{price}(T) \\
middle\text{-class\ car} &= car \times \text{sun\ roof}(T) \times \text{doors}(T) \\
my\ car &= \text{price}(25000) \times \text{sun\ roof}(yes) \times \text{doors}(5)
\end{align*}
\]

Notice that there is no distinction in OBJECTLOG between concepts and their instances and therefore the frame my car is formally not different from the others. Being an instance, it can be recognized as such by invoking the predicate atomic/1, which matches all the concepts just above ⊥. The following inheritance relationships are automatically inferred by the system:

\[
\begin{align*}
\text{isa(middle\ class\ car, car)} \\
\text{isa(my\ car, middle\ class\ car)} \\
\text{isa(my\ car, car)}
\end{align*}
\]

The goal-orientation of decision tables corresponds to constraints put on the variables representing the attributes. A possible (and straightforward) OBJECTLOG version of the decision table 3.1.1.1 for suitable cars expresses such constraints as conditions on a predicate suitable\_car/2:

\[
\begin{align*}
suitable\_\text{car}(\text{price}(P) \times \text{sun\ roof}(\_)) \times \text{doors}(\_), yes) & \leftarrow P \leq 20000. \\
suitable\_\text{car}(\text{price}(P) \times \text{sun\ roof}(yes) \times \text{doors}(3), no) & \leftarrow P > 20000, P \leq 25000. \\
suitable\_\text{car}(\text{price}(P) \times \text{sun\ roof}(yes) \times \text{doors}(5), yes) & \leftarrow P > 20000, P \leq 25000. \\
suitable\_\text{car}(\text{price}(P) \times \text{sun\ roof}(\_)) \times \text{doors}(\_), no) & \leftarrow P > 20000, P \leq 25000. \\
suitable\_\text{car}(\text{price}(P) \times \text{sun\ roof}(\_)) \times \text{doors}(\_), no) & \leftarrow P > 25000.
\end{align*}
\]

The Boolean result (yes or no) of the call to the query suitable\_car is given in the second argument of suitable\_car/2. One could of course think of writing the predicate in a more concise way, by removing the second argument and by keeping only the first and the third clauses (which state positive knowledge).
The main advantage in decision tables and frames is, in our opinion, the fact that they are easily understandable by non-experienced user thanks to the graphical interface they are based on. On the other hand, saying that decision tables are clearer or more succinct than their equivalent OBJECTLOG reformulation is probably not a very objective claim.

A drawback in the structure of Wisdom is that the graphical toolbox used for the specification of knowledge is not sufficient for most practical purposes. The proposed solution is an integration of Prolog to the system for inserting knowledge wherever it cannot be easily represented through decision tables or frames. This heterogeneity appears contradictory, as one of the purposes of the system was to get rid of Prolog's inadequacy for defining knowledge models.

3.1.2 LIFE

LIFE (Logic, Inheritance, Functions and Equations) is an extension of Prolog that provides
1. richer data structures (the so-called $\psi$-terms) and
2. interpretable functional expressions.
The first point has a number of consequences, as pointed out in [1]. For instance, similarly to OBJECTLOG, the notion of fully defined elements is, strictly speaking, no longer available. The second extension means that unification amounts now to normalizing a conjunction of equations. Those equations may be left in a not fully solved normal form if the arguments are not fully instantiated. A major difference with OBJECTLOG is then that LIFE takes advantage of a constraint solving technique (called relative simplification of constraints), whereas in OBJECTLOG unification is viewed as an algebraic operation based on the axioms. In other words, LIFE is a constraint logic programming (CLP) language that uses a concurrent resolution method based on a suspension strategy referred to as call-by-constraint-entailment, as opposed to Prolog's call-by-unification. Among other benefits of this technique, LIFE does not need Prolog's meta-predicates like freeze.

Let us explain this issue with an example. In LIFE one has the possibility of computing with partial information, like in the following clauses (rules):

\[
\begin{align*}
\text{minus}(\text{negint}) & \rightarrow \text{posint}. \\
\text{minus}(\text{posint}) & \rightarrow \text{negint}. \\
\text{minus}(\text{zero}) & \rightarrow \text{zero}.
\end{align*}
\]

Let us assume that the sorts $\text{negint}$, $\text{zero}$ and $\text{posint}$ are pairwise incompatible subsorts of the sort $\text{int}$, defined in LIFE as

\[\text{int} := \{ \text{posint}; \text{zero}; \text{negint} \}\]

Notice that the query

\[Y = \text{minus}(X:\text{int}), X = \text{minus}(\text{zero})\]

will first residuate on the variable $X$ after the first call to minus (which is inconclusive, as the sort int in the actual parameter is neither more specific than, nor incompatible with the sort negint of the first rule's formal parameter) and then as soon as more information is given for $X$ by means of the second call, $X$ is instantiated to zero and the query yields $Y = \text{zero}$. OBJECTLOG would deal with a case like this with a predicate minus/2 defined as follows:

\[
\begin{align*}
\text{minus}(\text{negint}, \text{posint}). \\
\text{minus}(\text{posint}, \text{negint}).
\end{align*}
\]
minus(zero, zero).

which on the query

minus(X, Y), minus(zero, X)

would of course answer, just like Prolog, with the result

\{ X = zero, Y = zero \}

The behavior is, though, different, as LIFE waits for instantiation on X, whereas OBJECTLOG just unifies X and Y with the values given in the first clause defining minus/2. When failing with the second call, it backtracks and takes the values from the second clause and, finally, from the third one, which gives the right answer.

LIFE’s data structures are sorts and features, used together to build \( \psi \)-terms. Sorts are very similar to OBJECTLOG’s constant concepts, as they are symbols denoting sets of values. In particular, T and \( \bot \) are sorts; values are assimilated to singleton sorts. The set of sorts has a partial ordering defined on it (corresponding denotationally to set inclusion) and sorts also have a greatest lower bound operation. Features are used to build \( \psi \)-terms by attaching attributes to sorts. \( \psi \)-terms can use variables to make the so-called coreferences:

\[ X: \text{person}(\text{age} \Rightarrow I: \text{int}, \text{spouse} \Rightarrow Y: \text{person}(\text{age} \Rightarrow I)) \]

This term denotes the set of all objects of sort person whose age is the same as their spouse’s, which is indicated by the sharing variable I (a coreference). Obviously this term can be reformulated in OBJECTLOG in the following manner:

\[ \text{isa}(X, \text{person} \times \text{age}(I) \times \text{spouse}(Y)), \text{isa}(I, \text{int}), \text{isa}(Y, \text{person} \times \text{age}(I)) \]

Graphically, \( \psi \)-terms can be viewed as commutative functional diagrams called order-sorted feature graphs (OSF-graphs), which differ from the lattices used in OBJECTLOG, as they can have cyclic structures expressed by coreferences. Furthermore the set of sorts is organized in a lower semi-lattice ([2]), while in OBJECTLOG we have distributive (full) lattices.

Unification between two \( \psi \)-terms is possible if there exists a GLB, which corresponds denotationally to set intersection. Furthermore, a \( \psi \)-term is associated with a logical formula so that unification of \( X_1: \psi_1 \) and \( X_2: \psi_2 \) corresponds to deciding satisfiability of

\[ \psi_1 \& \psi_2 \& X_1 \equiv X_2 \]

The reduction of such a logical formula takes place by means of normalization rules for equality, sorts, features and clash (understood as the presence of \( \bot \) in the structure).

The general proof-theoretic method for relative simplification of constraints using guarded rules is what really makes LIFE different from OBJECTLOG. On the other hand, the data structures are very similar, also from a set interpretation viewpoint. The special semantics of tagged sorts with coreference variables can be handled in OBJECTLOG, as previously shown, by means of \text{isa} (or \( = \) intended as both ways \text{isa}). Type equations and inequalities can also be expressed in LIFE with the := and, respectively, \(<\mid\) operators, as we did, for instance, with the int sort.
3.1.3 LOT

Integrating types in logic programming is an important issue, as the computers can check for type errors in the same way as they do with imperative languages, like Pascal. LOT, as outlined in [15], combines type checking with parametric polymorphism in a way that includes:

- feature types and constructor-based types;
- type hierarchies with possible multiple inheritance;
- statically checkable well-typing.

Constructor types are used to describe first-order terms, like in:

\[\text{ctype } \text{c\_date} := \{ \text{day\_month\_year} : \text{nat} \times \text{nat} \times \text{nat} \}.\]

where \text{nat} is the built-in constructor type for natural numbers. Each constructor has a fixed arity and type for each argument. Hierarchies, with possible multiple inheritance, can also be given for constructor types:

\[\text{ctype } \text{c\_parent} := \text{c\_father ++ c\_mother}.\]

Feature types describe record-like structures consisting of attribute-value pairs. An example of feature type paraphrasing an OBJECTLOG frame term given in the first chapter is:

\[\text{ftype } \text{person} <:\ [ \text{first\_name} : \text{string}, \text{surname} : \text{string} ].\]

\[\text{ftype } \text{book} <:\ [ \text{title} : \text{string}, \text{author} : \text{person}, \text{year} : \text{nat} ].\]

The feature type \text{person} is nested in the feature type \text{book} (together with the built-in types \text{string} and \text{nat}). A term of type \text{book} could then be:

\[\text{book} \{ \text{title} \Rightarrow \text{"Ulysses"}, \text{author} \Rightarrow \text{person} \{ \text{first\_name} \Rightarrow \text{"James"}, \text{surname} \Rightarrow \text{"Joyce"} \}, \text{year} \Rightarrow 1922 \}.\]

Inheritance can be expressed for feature types by means of the <:\ relationship. So, subtypes of a feature type \text{parent} could be defined as:

\[\text{ftype } \text{father} <:\ \text{parent}.\]

\[\text{ftype } \text{mother} <:\ \text{parent}.\]

In addition to this, if \text{date} is defined as a feature type instead:

\[\text{ftype } \text{date} <:\ [ \text{day} : \text{nat}, \text{month} : \text{nat}, \text{year} : \text{nat} ]\]

one can also define a subtype of it by adding an additional feature, for instance:

\[\text{ftype } \text{weekday} <:\ \text{date} \ast [ \text{name} : \text{string} ].\]

A term representing a weekday matching the structure defined above is subsumed by a term of type \text{date} that does not have the \text{name} field. Constructors do not allow this subsumption because name and arity of the constructor are fixed.

The difference between describing an application domain with constructor types or feature types is that:

- feature types are seen as intensional descriptions, while constructor types are defined extensionally by enumerating the constructors;
- unification corresponds to computing the greatest lower bound, if any, for constructor types, while it is performed by means of constraint solving (like in LIFE) for feature types:
• constructor types are based on the closed world assumption, whereas the interpretation for feature types is "open", as we saw for the type weekday.

In LOT the hierarchies containing feature types are always disjoint from hierarchies of constructor types, as the merging of those two categories is considered too complex and unnecessary. They can, though, be combined in an orthogonal way: the type of a constructor argument can be a feature type and the type of a feature can be any of the constructor types.

OBJECTLOG terms are very close to instances of feature types. Constructor types have, however, no counterpart in OBJECTLOG; as a matter of fact they do not add expressiveness to the language, as every constructor type can be restated as a feature type. Furthermore, it should be noticed once again that OBJECTLOG does not make any distinction between classes and instances of classes. Besides, OBJECTLOG allows both open and closed interpretations for types: as mentioned in the first chapter, we make a closed world assumption for type equations and an open one for type inequalities.

Another important topic in LOT is the parametric polymorphism. Let us consider the definition of a (feature) type for lists:

\[
\begin{align*}
\texttt{ftype list}. \\
\texttt{ftype elist <= list}. \\
\texttt{ftype nelist <= list} \left[ \text{head : term, tail : list} \right]. \\
\texttt{elist * nelist -> void.}
\end{align*}
\]

elist is the type for empty lists and nelist can have any term as head and recursively a list as tail. elist and nelist are disjoint. If we consider that both nat and string are subtypes of term, examples of well-typed lists, according to this definition, are for instance:

- \[\text{nelist} \{ \text{head} \Rightarrow 1, \text{tail} \Rightarrow \text{nelist} \{ \text{head} \Rightarrow 2, \text{tail} \Rightarrow \text{elist} \} \} \]
- \[\text{nelist} \{ \text{head} \Rightarrow 1, \text{tail} \Rightarrow \text{nelist} \{ \text{head} \Rightarrow "ab", \text{tail} \Rightarrow \text{elist} \} \} \]

The second case is that of a heterogeneous list, which is still well-typed because no restrictions are put on list elements to be of the same type. A parametric type definition can solve this problem:

\[
\begin{align*}
\texttt{ftype listp(T)}. \\
\texttt{ftype elistp <= listp(_).} \\
\texttt{ftype nelistp(T) <= listp(T) \left[ \text{head : T, tail : listp(T)} \right].} \\
\texttt{elistp * nelistp -> void.}
\end{align*}
\]

The general type allowed for the head is T, which is the same as for the tail. Alternatively one could consider inclusion polymorphism, that is, for instance, defining a type of lists of integers derived from the general type of lists:

\[
\begin{align*}
\texttt{ftype list\_int <= list}. \\
\texttt{ftype elist\_int <= list\_int * elist.} \\
\texttt{ftype nelist <= list\_int * nelist \left[ \text{head : int, tail : list\_int} \right].}
\end{align*}
\]

This type is very similar to type listp(int). The behavior in type inference is though different: if a relation

\[
\texttt{rel append : list \times list \times list.}
\]

is defined over lists, a call to append with the first two arguments of type list\_int infers that the third argument is of type list (but not list\_int). Instead, a parametric relation

\[
\texttt{rel appendp : listp(T) \times listp(T) \times listp(T).}
\]
called with the first two arguments of type \texttt{listp(int)} infers that the third argument is of type \texttt{listp(int)} as well.

In \textsc{ObjectLog} the parametric polymorphism can at most be simulated because the language does not support a real type system: types are facilities allowing to better specify relationships among concepts, but \textsc{ObjectLog}, differently from \textsc{Lot}, is not a typed logic programming language, as relations (predicates) are not defined over types. \textsc{ObjectLog} allows, however, an elegant formulation of typed lists as illustrated in paragraph 1.5.5.

### 3.2 An approach to Relational Databases

#### 3.2.1 Recasting relational databases in \textsc{ObjectLog}

A formal correspondence between relational algebra and concept algebra is established in [22]. Moreover, we have available here a yet more powerful tool, that is the \textsc{ObjectLog} programming language in its full expressiveness, which has concept algebraic terms installed in definite clauses. Our objective is therefore to represent in \textsc{ObjectLog} the constituents of relational databases in a consistent way and to show how this model can eventually be expanded.

A relational database, in the traditional understanding, is a collection of named relations. An \(m\)-ary relation is in turn a set of homogeneous \(m\)-tuples whose attributes are all distinct. Finally a tuple is a set of attribute-value pairs. Clearly, unnamed frames correspond to tuples in the following way:

\[
\prod_{i=1}^{m} a_i(o_i) \text{ is an } m\text{-tuple iff } \bigwedge_{i<j} a_i \neq a_j \Leftrightarrow i \neq j \land m \geq 1
\]

Besides, the values \(o_i\) inside the attributions must be atomic constants and therefore the whole tuple is ground; in [22] a temporary assumption is made about the concept algebra being a (simulated) 2-sorted algebra, with one sort \(O\) of individual objects and one sort \(R\) of relations. We drop this distinction as it was made with the only purpose of clarifying the relationship to conventional relational databases in which this 2-sortedness is explicitly assumed. A relation can be seen as a sum of homogeneous tuples; in order to recognize, later on, a relation in a database, we associate the relation with a name and store this information by means of a predicate, namely \texttt{dbrelation/2}. So, an \(m\)-ary relation \(r\) of cardinality \(n\) can be represented by the following \textsc{ObjectLog} clause:

\[
\text{dbrelation}(r, \sum_{j=1}^{n} \prod_{i=1}^{m} a_i(o_{ij})))
\]

or, more compactly,

\[
\text{dbrelation}(r, \sum_{j=1}^{n} \prod_{i=1}^{m} a_i(o_{ij}))
\]

\footnote{This means that all the tuples have the same field names (attributes).}
We shall furthermore assume that a relation with \( n = 0 \) (an empty relation) is \( \bot \), being \( \bot \) the neutral element for \( + \).

A collection of such clauses represents a relational database. Notice that storing relations in clauses prevents from loss of information. If we interpreted a relational database as a sum (a set) of relations, we would have had to face the problem of absorption. Let us consider, for instance, the two following (singleton) relations:

\[
\begin{align*}
& a_1(o_1) a_2(o_2) \\
& a_1(o_1) a_2(o_2) a_3(o_3)
\end{align*}
\]

Their sum simplifies as follows:

\[
a_1(o_1) a_2(o_2) + a_1(o_1) a_2(o_2) a_3(o_3) = a_1(o_1) a_2(o_2) \times (T + a_3(o_3)) = a_1(o_1) a_2(o_2)
\]

accordingly with the absorption axiom.

In this case, a further measure is needed, for instance to mark each relation with an external attribute representing the name of the relation, which avoids absorption:

\[
r_1(a_1(o_1) a_2(o_2)) + r_2(a_1(o_1) a_2(o_2) a_3(o_3))
\]

A relational database could then be algebraically reconceived in the following way:

\[
\sum_k r_k \left( \sum_{j=1}^n \prod_{i=1}^m a_{ik}(o_{jk}) \right) \text{ is a relational database iff } (r_i \neq r_j \iff i \neq j) \land k \geq 1
\]

### 3.2.2 RELATIONAL ALGEBRA OPERATIONS

Having seen how tuples, relations and relational database find their natural interpretation in OBJECTLOG (or just in the concept algebra, as for the second variant), we shall now explain how to reformulate common database operations.

- the union \( \cup \) of two relations (having identical attributes) is set-union and straightforwardly corresponds to \( + \);
- the natural join \( \bowtie \) of two relations is \( \times \), as already claimed in the first chapter;
- the intersection \( \cap \) is a special case of \( \bowtie \) with relations having identical attributes and therefore already covered by \( \times \);
- the Cartesian product (not indicated here with any symbol to avoid conflict of notation) is also a special case of \( \bowtie \) with relations having no attributes in common and therefore covered by \( \times \) as well;
- the selection \( \sigma_{a=v} \) with value \( v \) on attribute \( a \) can be rewritten as a natural join operation in the following way:

\[
\sigma_{a=v}(R) = R \bowtie R'
\]

where \( R' \) is the unary singleton relation containing the tuple \( a(v) \).
- projection cannot be easily treated algebraically. An algebraic formulation of projection can, however, be found in [22], based on the fact that presence of a tuple in a relation \( r \) can be checked with the equation

\[
r = r + \sum_{i=1}^m a_i(\varphi_i)
\]
which is the algebraic reformulation of

\[
\prod_{i=1}^{m} a_i(\varphi_i) \leq r
\]

We are interested here in a simpler version that takes advantages of the clausal level of OBJECTLOG. Projection \( r' \) of a relation \( r = \sum_{j=1}^{n} \prod_{i=1}^{m} a_i(\varphi_{ij}) \) over attributes \( a_{i_1}, a_{i_2}, \ldots, a_{i_k} \) \((i_p \in \{1, \ldots, n\}) (k \leq n)\) can be expressed with the following clause:

\[
\text{isa}(\prod_{p=1}^{k} a_{i_p}(X_{i_p}), r') \leftarrow \text{isa}(\prod_{i=1}^{n} a_i(X_i), r)
\]

where the variables are implicitly universally quantified variables. Such a clause reads as "if the tuple \( \prod_{i=1}^{n} a_i(X_i) \) is in \( r \) then the tuple \( \prod_{p=1}^{k} a_{i_p}(X_{i_p}) \) is in \( r' \), thus accomplishing projection. Renaming of attributes can be obtained by simply replacing \( a_{i_p} \) with new names \( a_{i_p}' \). The problem with this formulation is that we have an is-a clause at the right-hand side of the arrow, which is against what was established in paragraph 1.5.1, where is-a clauses were supposed to appear only in the body of clauses or alone in the head as variable-free \( \leq \) facts. However, for most cases projection finds its natural OBJECTLOG counterpart when querying, as shown below in the paragraph.

- if the notion of non-provability is introduced, there can be given a definition of relational difference in a way similar to projection. Such facility, commonly indicated by \( \leftarrow \), can be formulated in Prolog as

\[
\leftarrow (P) \leftarrow P, !, \text{fail}.
\]

Relational difference \( t \) between relations \( r \) and \( s \) can be then expressed as:

\[
\text{isa}(\prod_{i=1}^{n} a_i(X_i), t) \leftarrow \text{isa}(\prod_{i=1}^{n} a_i(X_i), r) \land \leftarrow \text{isa}(\prod_{i=1}^{n} a_i(X_i), s)
\]

If control information like \( \leftarrow \) is not comprised in OBJECTLOG, set difference has to be disregarded, which constrains us to Positive Relational Algebra (RA').

### 3.2.3 Querying a Database

Let us consider the database relation

\[
dbrelation(r, \sum_{j=1}^{n} \prod_{i=1}^{m} a_i(\varphi_{ij}))
\]

and the query with projection and selection

\[
\pi_{a_{i_1}, a_{i_2}, \ldots, a_{i_k}} \sigma_{a_r = a_q} r'
\]

where, for instance, \( a_r \) is not one of the attributes of the projection. This can be expressed as the OBJECTLOG goal:
\[ \text{dbrelation}(r, X \times a_{i_1}(\_), a_{i_2}(\_), \ldots \times a_i(o_{j_1}) \times \ldots \times a_i(\_)) \]

where the anonymous variable after the + indicates all the tuples discarded by the query and the attributions with an anonymous variable as value are the ones not included in the projection. \( X \) represents the "remaining part" of the selected tuples after projection and therefore it is the result of the query. If \( a_i \) was part of the projection, it would be sufficient to add the following goal to the previous one:

\[ Y = X \times a_i(o_{j_1}) \]

where = stands for OBJECTLOG unification and \( Y \) is the result searched for.

A simple example should clarify what explained above. The following table expresses the relation \textit{student}, which includes first name, surname and average grade of a student.

<table>
<thead>
<tr>
<th>surname</th>
<th>first name</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>John</td>
<td>A</td>
</tr>
<tr>
<td>Jones</td>
<td>Robert</td>
<td>B</td>
</tr>
<tr>
<td>Black</td>
<td>Vince</td>
<td>C</td>
</tr>
<tr>
<td>Jefferson</td>
<td>Thomas</td>
<td>B</td>
</tr>
</tbody>
</table>

\[ \text{table 3.2.3.1} \]

The relation in table 3.2.3.1 can be rewritten as the OBJECTLOG fact

\[ \text{dbrelation}(\text{student}, \quad \text{surname}(\text{smith}) \times \text{first_name}(\text{john}) \times \text{grade}(a) + \quad \text{surname}(\text{jones}) \times \text{first_name}(\text{robert}) \times \text{grade}(b) + \quad \text{surname}(\text{black}) \times \text{first_name}(\text{vince}) \times \text{grade}(c) + \quad \text{surname}(\text{jefferson}) \times \text{first_name}(\text{thomas}) \times \text{grade}(b)) \]

Suppose we want to know the surname of all students whose average grade is B, that is:

\[ \pi_{\text{surname}} \sigma_{\text{grade}=B} \text{student} \]

we can then write the OBJECTLOG goal:

\[ \text{dbrelation}(\text{student}, X \times \text{first_name}(\_) \times \text{grade}(b) + \_) \]

which generates the answer:

\[ X = \text{surname}(\text{jones}) + \text{surname}(\text{jefferson}) \]

obviously corresponding to the relational table

<table>
<thead>
<tr>
<th>surname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Jefferson</td>
</tr>
</tbody>
</table>

\[ \text{table 3.2.3.2} \]

More complicated cases are, however, not covered by this technique. For instance, the result is incorrect if the value used in the selection is a sum and we want to have the selecting attribute in the answer, like in the query:

\[ \pi_{\text{surname, grade}} \sigma_{\text{grade}=B \land C} \text{student} \]

which would be rewritten as
In relational table \( Y \), the original values of attribute \( \text{grade} \) have been forgotten and set to the object \( b + c \). One can solve this problem by creating one relational table for each term in the sum and then by summing up all the tables, like in:

\[
\begin{align*}
\text{dbrelation}(\text{student}, X \times \text{first_name}() \times \text{grade}(b + c) + ), & \quad Y = X \times \text{grade}(b + c) \\
\text{dbrelation}(\text{student}, X_1 \times \text{first_name}() \times \text{grade}(b) + ), & \quad Y_1 = X_1 \times \text{grade}(b) \\
\text{dbrelation}(\text{student}, X_2 \times \text{first_name}() \times \text{grade}(c) + ), & \quad Y_2 = X_2 \times \text{grade}(c) \\
Y & = Y_1 + Y_2
\end{align*}
\]

### 3.2.4 Extensions

The relational database model presented in the previous pages finds its natural generalization in concept algebra, which extends it with the "complex objects" introduced in the first chapter. In particular, the following issues are of interest:

- relations with **heterogeneous tuples**: in concept algebra a relation can be reconceived as a sum of tuples not necessarily having identical attributes;
- **nested complex objects**: individual constants can be replaced by generic (ground) OBJECTLOG terms.

In [22] the possibility of having multiple-valued attributes by removing functionality of attributions and introducing plural sum \( \oplus \) (for roles) is also addressed.

Notice that special functionalities of relational algebra, such as views and updates, could also find a (quite artificial) reformulation in OBJECTLOG, but extra-logical facilities such as assertions and retractions of clauses would be needed.
Chapter 4  A compiler for OBJECTLOG

4.1  Design of the compiler

In this section we shall describe compilation of OBJECTLOG programs into Prolog. Prolog has been chosen as target language because it is a logic programming language and so is OBJECTLOG. In this way we can focus on compilation of OBJECTLOG terms and installation of calls to OBJECTLOG unification (described in chapter 2) without having to reconsider the entire paradigm of logic programming.

4.1.1  COMPILING OBJECTLOG CLAUSES INTO PROLOG

An OBJECTLOG program is an ordered collection of definite clauses, goals and, possibly, type definitions. We concentrate here on clauses and goals, of which we give precise definitions, and draw a scheme for translating them into Prolog.

A definite clause is a logical formula of the form:

$$\alpha_0 \leftarrow \bigwedge_{i=1}^{m} \alpha_i$$

(4.1.1.1)

where $\alpha_i$ are atoms and, in particular, $\alpha_0$ is called the head of the clause and the conjunction of atoms to the right of the arrow is called body. Atoms are in turn expressions of the form:

$$\alpha = p(\varphi_1, \varphi_2, \ldots, \varphi_n)$$

(4.1.1.2)

where $\varphi_i$ are OBJECTLOG terms, $n$ is the arity and $p$ is the name of the atom. OBJECTLOG terms have been thoroughly described in the first chapter and we assume that they can always be represented by a Prolog term. How this can be done is described in paragraph 4.2.1.

An atom that is not a head of a clause can straightforwardly be translated into a Prolog atom with the same name and the same arity where the $\varphi_i$'s are replaced by their Prolog representations $t_i$.

The OBJECTLOG definite clause:

$$p(\varphi_1, \varphi_2, \ldots, \varphi_n) \leftarrow \bigwedge_{i=1}^{m} \alpha_i$$

where the head has been explicitly reported, can be reformulated in Prolog in the following way:

$$p(X_1, X_2, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{n} \text{unify}(X_i, t_i) \bigwedge_{i=1}^{m} A_i \bigwedge_{i=1}^{n} \text{nonnull}(X_i)$$

where $X_i$ are (fresh) Prolog variables, $t_i$ is the Prolog representation of the OBJECTLOG term $\varphi_i$, unify is the predicate performing unification of OBJECTLOG terms, $A_i$ is the translation of the atom $\alpha_i$ and nonnull is a predicate that fails if its argument is the representation of $\bot$.  

65
In other words, \texttt{nonnull} serves to discard \(\bot\) as an acceptable solution to unification (at the clausal level) thus interpreting \(\bot\) as failure.

\textbf{OBJECTLOG} programs can also contain explicit calls to \texttt{unify/2} and \texttt{isa/2}, which are translated in the same way as ordinary atoms.

Facts, that is definite clauses without body \((m = 0)\), are translated in the same way. Goal clauses are treated in the same way as definite clause bodies.

Type definitions can be stored as facts of the special predicates \(=/2\), \(\leq/2\) and \(\geq/2\), which cannot be written in bodies of clauses. In particular, the type definition

\[
\text{type\_name} = \text{ground\_type\_expression}
\]

can be translated into the Prolog fact:

\[
= (\text{type\_name}°, \text{ground\_type\_expression}°)
\]

where the symbol "°" stands for "Prolog representation of". Type inequalities are translated analogously.

So far we have described how to map every single type of clause into a Prolog clause. The overall compilation of an \textbf{OBJECTLOG} program simply consists of translating all its clauses in Prolog and writing them to a target file where the original order of clauses is kept.

\subsection{A compiler and an interpreter}

In order to execute compiled \textbf{OBJECTLOG} programs, one needs a Prolog interpreter. Furthermore, predicates \texttt{unify} and \texttt{isa} must be available together with predicates for transforming \textbf{OBJECTLOG} terms into Prolog terms. One could think of always adding the clauses defining such predicates to the code of a compiled \textbf{OBJECTLOG} program. This solution results however in an excessive waste of space, which could be saved if those predicates were available in another way.

Moreover, when querying the Prolog system, one has to put explicit calls to the predicates for representation of terms both for translation to Prolog and from Prolog, as the computed answers (variable substitutions) are Prolog terms.

The idea that we present here is therefore that of providing an \textbf{OBJECTLOG} environment, which can be loaded as a Prolog program on the Prolog system and therefrom used for executing compiled \textbf{OBJECTLOG} programs through a transparent user interface. The advantages of such stratagem can be sketched as follows:

- when querying the system, \textbf{OBJECTLOG} terms are automatically translated to Prolog (for computation) and retranslated back to \textbf{OBJECTLOG};
- \textbf{OBJECTLOG} facilities, such as the predicates \texttt{unify} and \texttt{isa}, are resident in the system and need not be included in the code of compiled programs;
- flexibility can be added on request by the user for several circumstances. For instance, the (back- or forth-) translation of terms can be disabled in order to monitor the execution of a program by looking at its internal representation instead of the \textbf{OBJECTLOG} sugared form for terms;
- all this operations are done in a completely transparent way.
Both the interpreter and the compiler are Prolog programs and their implementation is thoroughly described in the following sections.

4.2 Implementation of a compiler for ObjectLog

The implementation of the compilers for OBJECTLOG, and OBJECTLOG2 are described separately due to the substantial difference of algorithmic effort required. In this section, the focus is on OBJECTLOG1; OBJECTLOG2 will be described later on in the chapter as an enrichment of the program presented here.

The compiler is written in full Prolog as it takes advantage of extra-logical facilities (like cut) and term utilities for manipulating names, arguments and arity of terms and for checking instantiation of variables. The Prolog system used to develop the program is SICStus Prolog, which, among other things, made it possible to write a flexible ObjectLog interpreter.

4.2.1 Representation of OBJECTLOG1 terms

In paragraph 4.1.1 we outlined the translation of ObjectLog clauses into Prolog; however, the representation of ObjectLog terms was only assumed to be possible, but was not explained. Here we shall clarify how such a mapping can be done for OBJECTLOG1 terms, that is for frames of form (2.1.1.1). The following scheme is proposed:

- a constant is represented by a one-element list containing that constant, that is the constant $c$ is represented by the list $[c]$;
- similarly, a variable $X$ is represented by $[X]$;
- an attribute $a(\phi)$ is represented as $[a(\phi^o)]$, where $\phi^o$ is the representation of $\phi$;
- a product of terms $\phi_1 \times \phi_2 \times \ldots \times \phi_n$ is represented as the list concatenation $\phi_1^o \wedge \phi_2^o \wedge \ldots \wedge \phi_n^o$, where ^ indicates list concatenation (Prolog's append).

This mapping establishes a univocal way of representing OBJECTLOG1 terms in Prolog. The point is that an OBJECTLOG1 term is actually equivalent to an entire class of terms, which can be obtained by applying idempotency, commutativity, associativity and distribution over attribution. For this reason, we introduced in paragraph 2.1.3 a canonic form, which is representative for the whole class of terms equivalent to it. For instance, the term:

\[
 a_1(\{c_3 \times Y \times c_2\} \times c_4 \times c_1 \times X \times a_1(\{c_2 \times a_2(c_3 \times c_1)\}))
\]

is represented in Prolog by

\[
 [a_1(\{c_3, Y, c_2\}), c_4, c_1, X, a_1([c_2, a_2([c_3, c_1])])]
\]

Its OBJECTLOG1, canonic form is

\[
 X \times c_1 \times c_4 \times a_1(\{Y \times c_2 \times c_3 \times a_2(c_3)\})
\]

represented in Prolog by

\[
 [X, c_1, c_4, a_1([Y, c_2, c_3, a_2([c_3])])]
\]

This transformation is based on alphanumeric sorting of the list and removal of duplicates, which is done by the SICStus Prolog built-in predicate sort/2. Then, equally-named attributions are merged into one and the sorting is recursively applied to the value
of the attribution. Notice that the ordering is such that first come uninstantiated variables, then functors of increasing arity. In this way, attributions (represented by unary functors) are always the last in the list.

A special remark is devoted to the distinguished elements of the lattice, namely $T$ and $\bot$. It goes without saying that $T$ is represented by an empty list\(^{50}\); it is intuitively the limit product where the number of factors is zero, as $T$ is the neutral element for $\times$. For $\bot$ we use a constant called *bottom*, that is $[\text{bottom}]$. The notation could, of course, be assumed in the opposite way, as $\bot$ is the neutral element for sums; but, as a matter of fact, we have no $+$ in $\text{OBJECTLOG}$, and therefore the convention assumed is perhaps the most natural one.

### 4.2.2 Dynamic Adjustment of Terms

The dynamical evolution of variable instantiations gives rise to representational problems of terms. Let us consider, for instance, the following frame:

$[X, c_1, c_2]$

and let us assume that during the execution of a program the variable $X$ was instantiated to the representation of the constant $c$, that is $[c]$. The Prolog system will effectuate the variable substitution automatically, so that the original frame has dynamically evolved to:

$[[c_3], c_1, c_2]$

Not only the constants are in the wrong order, but also there is a list nested in the list. One could think then of changing the representation of $\text{OBJECTLOG}$ terms, for instance as in $[21]$, where variables and constants are represented unchanged and products are represented as lists instead of list concatenations. However, this would not solve the problem, as the order can still be affected and nested lists can still emerge for certain substitutions (for instance if a variable becomes instantiated to an $\text{OBJECTLOG}$, product).

The following actions are then necessary:

- "flattening" the list (by applying this flattening recursively inside attributions);
- bringing the flattened list to canonic form.

Such actions are accomplished by the predicates $\text{flatten}/2$ and, respectively, $\text{canonic_form}/2$; a predicate $\text{adjust}/2$ takes care of calling them in sequence. Term adjustment is needed before each call to $\text{OBJECTLOG}$ clauses and therefore the scheme of translation presented in the first section needs to be changed as follows. The $\text{OBJECTLOG}$ clause:

$\phi_0(\phi_{01}, \phi_{02}, \ldots, \phi_{0n}) \leftarrow \bigwedge_{i=1}^{m} p_i(\phi_{i1}, \phi_{i2}, \ldots, \phi_{in})$

is translated as

\(^{50}\text{This is, of course, just a convention of notation. Alternatively we could have represented it by a constant called } \text{top}.$
where $X_{ij}$, $\bar{X}_{ij}$ and $\bar{X}_{0j}$ are Prolog variables and $t_{ij}$ is the Prolog representation of $\varphi_{ij}$.

### 4.2.3 Compilation and Execution of Small Example Programs

The techniques for compiling OBJECTLOG\textsubscript{1} programs described so far might result rather abstract at first glance. In this paragraph we are going to show the translation of the well-known program for checking membership in a list, which looks in OBJECTLOG\textsubscript{1} as follows:

\[
m(X, h(X) \times t(_)) \leftarrow m(X, Z).
\]

This program translates to the following Prolog program:

\[
m(A, B) \leftarrow \text{adjust([C], D), unify(A, D),}
\]

\[
\text{adjust([h([C]), t([_])], E), unify(B, E), nonnull([A, B])}.
\]

\[
m(A, B) \leftarrow \text{adjust([C], D), unify(A, D),}
\]

\[
\text{adjust([h([C]), t([E])], F), unify(B, F),}
\]

\[
\text{adjust([C], G), adjust([E], H), m(G, H),}
\]

\[
\text{nonnull([A, B])}.
\]

The first clause is a fact and, as such, we translate it from the scheme from paragraph 4.2.2 with $m = 0$. Notice that every term is adjusted before calling any predicate, including unify. Moreover, we extended the syntax of nonnull to lists of terms, so that the adjustment and the non-null check can be performed with just one call.

The naming of fresh Prolog variables is automatically handled by the built-in predicate \texttt{portray_clause/1}, available in SICStus Prolog, which, furthermore takes care of adding a period at the end of the clause, so that it will be read correctly when consulting the program. No problems arise if the variables in the OBJECTLOG\textsubscript{1}, and in the Prolog program have different names as long as such names are mapped consistently (that is by keeping track of what variables should have the same name in the program).

It is interesting, at this point, to look closely at the resolution of a goal. We assume that an OBJECTLOG\textsubscript{1} interpreter is running (see paragraph 4.2.5) and translating OBJECTLOG\textsubscript{1}, terms into their Prolog representation when querying the system and vice versa when outputting answers. Let us consider the example OBJECTLOG\textsubscript{1} goal:

\[
\leftarrow m(c, h(c) \times t(nil)).
\]

It translates to the Prolog goal:
\[ \leftarrow \text{adjust}([c], V_1), \text{adjust}([h([c]), t(nil)], V_2), m(V_1, V_2). \]

The first two calls could actually be avoided because no adjustment is needed at the beginning of a goal\(^{51}\); in fact, \(V_1\) and \(V_2\), which are internal variables, are bound exactly to \([c]\) and, respectively, \([h([c]), t(nil)]\) after those calls. When the Prolog predicate \(m\) is called, the goal to be resolved is:
\[ \leftarrow m([c], [h([c]), t(nil)]). \]

The first clause of the translated program is then selected, which causes the following subgoals\(^{52}\):
\[ \leftarrow \text{adjust}([V_3], V_4), \quad \{ V_4 / [V_3] \} \]
\[ \leftarrow \text{unify}([c], [V_3]), \quad \{ V_3 / [c] \} \]
\[ \leftarrow \text{adjust}([h([c]), t([V_3])], V_6), \quad \{ V_6 / [h([c]), t([V_3])] \} \]
\[ \leftarrow \text{unify}([h([c]), t(nil)], [h([c]), t([V_3])]), \quad \{ V_5 / [nil] \} \]
\[ \leftarrow \text{nonnull}([c], [h([c]), t([nil])])), \quad \{ \} \]

The last goal succeeds because none of the (adjusted) members of the argument of \(\text{nonnull}\) are \(\bot\).

### 4.2.4 Limitations

During the execution of the very simple example from the last paragraph, the predicate \(\text{adjust}\) was called 4 times explicitly plus 2 other times by \(\text{nonnull}\), that is once for every \(\text{OBJECTLOG}_1\) term used in every atom. Clearly this constitutes a considerable overhead even for the simplest programs. On the other hand, the adjustment of terms is unavoidable, as shown in paragraph 4.2.2 and we do not have a better solution at the moment.

Another remarkable limitation is the pure/ground constraint, even if partially relaxed, as introduced in paragraph 2.1.6. Although the non-ground \(\text{OBJECTLOG}_1\) goal
\[ \leftarrow m(c \times X, h(c \times d) \times t(nil)). \]
would succeed and return the substitution \(X / d\), and the goal
\[ \leftarrow m(X, h(c_1) \times t(h(c_2) \times t(...h(c_n) \times t(nil)))). \]
would return the sequence of substitutions
\[ X / c_1 ; X / c_2 ; ... ; X / c_n \]
there are, however, goals that violate the constraint. Let us consider, for instance, the goal:
\[ \leftarrow m(c, h(X) \times t(nil)). \]
It should succeed and return the substitution \(X / c\), but it returns an error message instead, as we tried the \(\text{OBJECTLOG}_1\) unification:
\[ \text{unify}(h(X) \times t(nil), h(c) \times t(V_1)) \]

\(^{51}\) Conversely, if there were other atoms in the goal, the adjustment between two atoms would be necessary.

\(^{52}\) We shall enclose between curly brackets the substitution(s) issued by the Prolog system after the resolution of each subgoal.

\(^{53}\) The semicolon is commonly used to commit a backtracking on Prolog SLD resolution and finding alternative answers to the goal.
where both terms are non-ground and neither is a variable. Still, the predicate \( m \) was designed for membership check, and not for generation of lists, therefore this drawback was not unexpected.

### 4.2.5 The Interpreter and the Command Line

In SICStus Prolog there is a facility called `term_expansion/2` which is a redefinable predicate allowing to change the way the system reads terms. The first aim of the interpreter is to hide to the user the internal representation of `OBJECTLOG1` terms, so that advantage can be taken of a syntactic sugar provided by the compiler. The scheme adopted for the interpreter looks as follows:

- first are handled the cases which lead to errors, as, for instance, a variable as a goal;
- then meta-goals are considered. Having an interpreter actually means that there are two levels: an object-level (the level of `OBJECTLOG1` clauses) and a meta-level (Prolog, which controls the execution of `OBJECTLOG1` programs). We let the user express goals at the meta-level basically because some facilities of the Prolog system can be used, for instance, for monitoring the execution of a program or for debugging it (e.g. with a `trace`-ing). Meta-goals are preceded by a question mark, used here as an `unquoting operator`, if one considers the whole interpreter as a meta-program whose arguments are `OBJECTLOG1` programs and goals. However we do not allow mixing meta- and object-code, as this is outside the scope of this document and therefore a goal preceded by a `?` is considered a pure Prolog goal.
- there are two directives allowed by the interpreter for deciding whether the output is to be shown in pure Prolog or with the `OBJECTLOG1` syntactic sugar. The first one is called `prolog` and the second one `objectlog`.
- there is a command for compiling `OBJECTLOG1` programs into Prolog programs and one for consulting compiled files in order to be able to write some queries at the command line.
- finally, goals are translated according to the schemes presented in the previous paragraphs.
- if none of these actions was possible, then there has been an error, which we report with a message on the screen.

The idea is then that a term (goal) can be manipulated by the interpreter and, after the manipulation, given to the Prolog system as an ordinary term. The scheme for the expansion of terms is simply the following:

```
term_expansion(?-(Goal), ?-(TGoal)) ← translate_formula(Goal, TGoal).
```

where `translate_formula` is a predicate that translates a conjunction of `OBJECTLOG1` atoms into a Prolog formula, and the special Prolog functor `?-` is used to indicate queries at the command line.

All the goals could have been written directly in the `OBJECTLOG1` program (and executed while consulting its Prolog translation). On the other hand, this adds flexibility to the system and allows the user trying his/her own goals directly at the command line and with no need of putting them in the same file as the program.

Furthermore, the other advantage of having an interpreter is that terms can be translated back to `OBJECTLOG1`, although the whole computation takes place in Prolog. For

---

54 With Prolog term, in this context, one indicates any expression ended with a period, so that clauses are terms as well.
this purpose, SICStus Prolog provides a predicate called \texttt{portray} which allows changing the way Prolog outputs answers. If this predicate does not succeed, then the answers are outputted in the normal way. The back-translation can be schematized in the following way:

\[
\text{portray}(X) \leftarrow \text{objectlog}(\text{on}), \text{back_translate}(X, Y).
\]

to be read as "if the user wants answers to be in OBJECTLOG, then back-translate them". \texttt{objectlog(on)} is a fact that is asserted by default and when the user types the \texttt{objectlog} directive at the command line. Similarly, the fact \texttt{objectlog(off)} replaces \texttt{objectlog(on)} if the directive \texttt{prolog} is typed at the command line.

\texttt{OBJECTLOG} terms are constructed by means of the \* operator (used as \texttt{x}), which is defined as left-associative (as well as the dot operator for lists). The representation of such a term happens with a recursive predicate called \texttt{crux2list} that simply takes care of going through the \* structure and transforming it into a list structure. This operation is performed recursively for values of attributions. The back-translation is realized symmetrically.

\section*{4.2.6 A TINY MANUAL FOR USERS OF THE OBJECTLOG\textsubscript{1} COMPILER}

Once SICStus Prolog is running, one can load the \texttt{OBJECTLOG} environment by consulting the file "\texttt{objectlog1}"

\texttt{[objectlog1]}.  

Now the system is ready either for directly answer queries or for compiling and consulting programs. We assume that an \texttt{OBJECTLOG} file has the extension ".ol", as all \texttt{OBJECTLOG} files, and its compiled version has the same name, but extension ".pl" (that of Prolog files). Compilation is cast with the command:

\texttt{compile(filename)}

where \texttt{filename} is without extension, which creates the compiled version in the same directory with the appropriate file name and extension. In order to be able to use a compiled file one needs to consult it. This happens in the same way as ordinary Prolog files for SICStus Prolog, that is:

\texttt{[filename]}

with no need of indicating the ".pl" extension. Answers are normally printed out in \texttt{OBJECTLOG} style. If, however, one wants to have them in Prolog, the following directive can be typed at the command line:

\texttt{prolog}.

\texttt{OBJECTLOG} style for answers is restored with the directive:

\texttt{objectlog}.

If one does not want to use the \texttt{OBJECTLOG} interpreter for a certain goal, the goal has to be preceded by the ? operator. Notice that answers are printed out in \texttt{OBJECTLOG}, or in Prolog independently from the presence of ? in the goal. Execution of Prolog goals and errors during the compilation or the execution are monitored by \texttt{OBJECTLOG} messages.
4.3 Implementation of a compiler for ObjectLog$_2$

ObjectLog$_2$ allows sums in type equations. For ease of notation, sums can also be written in the outmost level of ground isa clauses. However, no special representation of general sum terms is provided. The frames in the sum are just linked by means of the binary left-associative operator $+/2$ and represented in the way explained in paragraph 4.2.1.

The compiler is organized in four modules listed in appendix A. In particular, the module objectlog2.pl contains the clauses for compilation of ObjectLog$_2$ programs and consults the other modules. The module unification.pl contains the clauses for unification and is-a check; interpreter.pl installs the interpreter in the system; representation.pl contains all the clauses for representation, adjustment and simplification of terms.

4.3.1 Compiling type equations

All the considerations made in the last section about the compiler for ObjectLog$_1$ still hold for ObjectLog$_2$. New issues are, however, taken into account. In this paragraph we shall explain how compilation of type equations takes place.

Any clause whose main functor is $=$ (instead of $←$) is considered a type equation and compiled as such. The following conditions must hold:

- the type equation is a completely ground expression;
- the left-hand side is a constant;
- the right-hand side is either a sum or a product of constants, or an attribution (see also paragraph 2.2.1).

Sum type equations are translated as seq/2 facts. Similarly, product type equations and attribute type equations are translated as peq/2 and, respectively, aeq/2 facts. If the translation succeeded, the type name is stored in a clause def/1, which indicates that the type has been defined. Such clause is written in the compiled file, whereas the same information is asserted in memory as the fact def_/1 (we used a different name, because at the end of the compilation all such transitory clauses can be retracted, thus allowing further compilations).

When recognizing type equations as sum, product or attribute type equations (see also paragraphs 2.2.1 and 2.2.2), the following checks are made:

- the type name must not be that of a type already defined, including basic (built-in) types (see paragraph 4.3.2), otherwise the compiler reports an error of "duplicate type definition";
- the type in the left-hand side must not appear in the right-hand side, otherwise there is direct recursion in the type equation;
- there must not be any circular dependency between the type in the left-hand side and the types in the right-hand side, otherwise we have a type definition with indirect recursion. If the definition is not circular, then we assert in memory all the new dependencies as dep_/2 facts (which are used in the algorithm for dependency checking). At the beginning of the compilation there are no dep_/2 facts in memory and at the end all such facts are retracted.
- in an attribute type equation, the value inside the attribution must be a defined type.
• if the definition was a sum or a product type equation, all the relevant leq/2 clauses (see paragraph 2.2.3) are written to the compiled file.

4.3.2 ATOMICITY AND BASIC TYPES

The definition of atomicity for OBJECTLOG2 terms has been given in paragraph 2.2.5. Special attention has to be paid to the case of basic types (or built-in concepts), because they are not defined by any type equations. Therefore, we shall make the assumption that built-in concepts are not atomic, as, ideally, they are defined by a (possibly infinite) sum of all individuals belonging to them.

The basic types, besides T and ⊥, dealt with by OBJECTLOG2 are natural numbers (with 0) and real numbers, represented by nat and, respectively, real. The is-a test is assigned a particular meaning when performed on basic types (as second argument): before trying to check lattice inclusion, possible numeric properties of the first argument are tested. The clauses defining the is-a test shown in paragraph 2.2.3 should then be enriched with the following ones:

\[
\text{isa}(N, \text{nat}) \leftarrow \text{integer}(N), N \geq 0.
\]

\[
\text{isa}(R, \text{real}) \leftarrow \text{float}(R).
\]

where integer, float and ≥ are Prolog built-in predicates.

4.3.3 REPRESENTATION OF TEXTS

No built-in concept for strings is available in OBJECTLOG2. However, a way is provided for easily representing texts. We can imagine that a text is a list of words, so that the sentence "hello world" is represented by the OBJECTLOG2 term:

\[
h(\text{hello}) \times t(h(\text{world}) \times t(\text{nil}))
\]

where each of the elements of the list is a constant. In other words, a string without spaces is associated to the name of a constant and a text is a list of constants.

The reversible meta-predicate text/2 is available in OBJECTLOG2 and can be used for transforming a text into a list of constants and vice versa. One of the two arguments must be ground and the other one must be a variable. The text must be enclosed between single quotes, as usually done for special atom names in Prolog. In the example above, the list representation can be obtained with a call to

\[
\text{text('hello world', } T).
\]

text/2 is used in chapter 5 as a facility for entering sentences to be translated.

4.3.4 TRANSLATION OF DCG RULES

In many Prolog systems DCG notation is available as a syntactic sugar for writing DCG rules. Definite clause grammars are an extension of the well-known context-free grammars. A grammar rule in Prolog takes the general form (see, for instance [24]):

\[
\text{head} \rightarrow \text{body}.
\]
meaning "a possible form for head is body". Grammar rules are merely a convenient "syntactic sugar" for ordinary Prolog clauses. Each grammar rule takes an input string, analyzes some initial portion, and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output strings are not written explicitly in a grammar rule, but the syntax implicitly defines them. Such rules can be expanded to Prolog clauses by making explicit the extra arguments. For instance, a rule such as

\[ p(X, Y) \rightarrow q(X), r(X, Y), s(Y). \]

translates into

\[ p(X, Y, S_0, S) \leftarrow q(X, S_0, S_1), r(X, Y, S_1, S_2), r(Y, S_2, S). \]

Terminals are translated using a built-in predicate for indicating connection, such as

\[ 'C'(S_1, X, S_2) \]

which reads as "point \( S_1 \) is connected by terminal \( X \) to point \( S_2 \)", and defined by the single clause

\[ 'C'(\[X|S], X, S). \]

Then, for instance, the rule

\[ p(X) \rightarrow [\text{term}], q(X). \]

is translated by

\[ p(X, S_0, S) \leftarrow 'C'(S_0, \text{term}, S_1), q(X, S_1, S). \]

Extra conditions expressed as explicit procedure calls (indicated by curly brackets) naturally translate as themselves.

In SICStus Prolog it is possible to translate DCG rules as clauses, according to the principle explained above, with the predicate \texttt{expand_term/2}. Therefore, wherever a DCG rule is found, it is first expanded to a clause, and then the OBJECTLOG translation proper can take place as for the other cases. Instances of the 'C' atom are translated verbatim, except for the fact that the name 'C' is replaced by 'C2'. 'C2'/3 is a predicate defined for OBJECTLOG that first converts the first and the third arguments from OBJECTLOG lists to Prolog lists, then calls 'C' with the translated arguments, and finally translates back the resulting first and third argument to OBJECTLOG lists. Instances of =/2 are translated with unify/2.
Chapter 5  A natural language translator in
OBJECTLOG

5.1  Aim of the example

5.1.1  INTRODUCTION AND RATIONALE

In this chapter we shall introduce a small natural language translator from English to Italian and vice versa. Several quite powerful tools, commonly referred to as "machine translators", are already available nowadays for converting sentences from a language to another. However, much more still has to be done before sensible results can be obtained without the intervention of a human.

With no claim of independence or usefulness of the program presented in these pages, whose results are though reasonably accurate thanks to the very limited application domain it deals with, we propose a different approach to the problem, as explained in the following lines. Furthermore, it should be clear that the aim of this chapter is to show how OBJECTLOG can be used for writing programs and which style it calls for, and not to provide the reader with a translator he/she can take advantage of for practical purposes.

The particular context we want to stick to is the one relevant to railways and airports, as they are usual themes in pocketbooks for travelers, where sentences like "is this train going to Milan?" are reported together with their respective translation in the other language. The interest in operating within such a limited range is that it is possible to handle all the relevant sentences with a relatively small grammar and a clear semantics, so that the words can be first understood before being rewritten in the second idiom. By considering that Italian- and English-speaking people have a common understanding (what is sometimes called *forma mentis*), at least in this case where all the involved concepts belong to everyday life for both, the translation can then take place as follows. First the sentence is brought into a syntactical representation specific for its original language, then a semantic form is abstracted from the syntax and represented with a complex object, which ends the phase of analysis. The synthesis of the sentence in the target language goes, of course, in the opposite way, from the semantics to the syntax and from the syntax back to plain text. In principle, an English sentence can even be translated back into English itself, meaning that it can be restated in a way that is maybe syntactically different but semantically equivalent, and similarly for Italian.

The vision at the semantic level tends to be less precise than in the levels below in fig. 5.1.1.1, as all the details about the particular words or expressions used to formulate the sentence are lost. So, in our model, saying that a train is leaving for Milan or that its destination city is Milan is semantically the same, though the formulation is different. One

---

55 This would normally be a strong assumption, especially because different people can have different ideas about the same thing or even use concepts that are unknown outside their specific context. For instance, the name of a particular dish is of no use for a person who never tasted it and therefore it does not represent any concept (or a very vague one). Also, a car running on the left-hand side of a street would be considered perfectly normal in England, although dangerous in Italy. Still, in our case such assumption can be made without any further concern.
might feel that this abstraction could provoke a (slight) loss of information or even a change of the meaning; all these problems, and others, are pointed out in the next pages.

5.1.2 **COMPUTERS DO NOT THINK**

In a chapter about the limits of conceptualization [25], Sowa explains that the verbs "to understand" or "to think" are not suitable for computer systems. This is a consequence of the inability of machines in general of experiencing emotions, which can at most be simulated. Several AI systems can map an English sentence to a logical formula, derive other formulae from it, and then map those formulae back to English. Still, we cannot say that such a system understands English. Therefore our claim about understanding of sentences is to be interpreted in a restricted sense: the semantic representation that we obtain from a natural language expression is an abstraction capturing the intended meaning of the sentence in the world of interest. Such abstraction can, in turn, be used for other purposes, like retranslating to another natural language, querying or stating facts in a knowledge base etc. This is different from what is currently being done by "machine translators" that try to map combinations of words in the source language into other combinations of words in the destination language without taking into consideration the semantics.

5.2 **The semantic structure**

5.2.1 **SCHEME OF THE SEMANTICS**

As pointed out in the previous section, a decision has to be taken about what kinds of sentences are accepted and in which way they can be written. But first we are going to outline the scheme of the semantics that applies to this example.
As natural for an ordinary conversation, both questions and positive sentences (assertions) are allowed, which can be expressed in OBJECTLOG as follows:

\[ \text{semantics_of_sentence} = \text{semantics_of_assertion} + \text{semantics_of_question} \]

When a statement is asserted, the speaker describes a context, an ensemble of circumstances that characterize the fact he/she wants to inform of. We shall call this fact an event.

\[ \text{semantics_of_assertion} = \text{designated} (\text{event}) \]

A question is somewhat similar to an assertion, in that it is related to an event as well; but not all the aspects of this event are known. Such aspects pertain to the interest of the question, that is what is asked for.

\[ \text{semantics_of_question} = \text{asked} (\text{interest}) \times \text{context} (\text{event}) \]

With the next two definitions we are going to fix the bounds of the set of sentences belonging to this example, as interest and event are explained in all their details. What characterizes an event is:

- the vehicle (a train or an airplane), as the event can either refer to a railway station or to an airport;
- the source city;
- the destination city;
- the time (of departure or arrival) from the schedule;
- the gate (for airports) or track (for railways), here referred to as point;
- the frequency at which the event occurs;
- the duration of the event.

What one can ask for are each of the elements listed above, except for the vehicle, because it should be known if the question is formulated in an airport or not. Furthermore, if the given contextual event has no missing part, one can ask whether it is true or not. For example, in the sentence "does this train go to Milan?", the questioner mentions the destination, but does not know if it is the right one. This is of course different from the case of a question like "where does this train go to?", in which the destination is asked and not known at all. The following equations can thus be written:

\[ \text{interest} = \text{truth} + \text{source} + \text{destination} + \text{schedule} + \text{point} + \text{frequency} + \text{duration} \]
\[ \text{event} = \text{vehicle} \times \text{source} \times \text{destination} \times \text{schedule} \times \text{point} \times \text{frequency} \times \text{duration} \]

A frame or sum of frames is then associated with all the constituents of an event in an obvious way, commented when needed:

\[ \text{vehicle} = \text{by}(\text{object}) \]
\[ \text{source} = \text{from}(\text{city}) \]
\[ \text{destination} = \text{to}(\text{city}) \]
\[ \text{schedule} = \text{at}(\text{time}) \]

The frequency of an event can be expressed either by the number of times it happens in a given period (e.g. "five times a week") or by the time interval elapsed between two such events (e.g. "every 20 minutes").

\[ \text{frequency} = \text{times} (\text{nat}) \times \text{per} (\text{period}) + \text{every} (\text{time}) \]

To keep on with the same style, one could have introduced an "external" attribute enclosing the whole frame written above, so that frequency would be always represented by the same attribute, as for all the other constituents of an event described so far. This alternative form could for instance be:
\[ \text{frequency} = \text{repeated}(\text{times}(\text{nat}) \times \text{per}(\text{period}) + \text{every}(\text{time})) \]

The same consideration holds for the definition of \textit{point}. A point is a \textit{from_point} if the track or gate is the one where a departure takes place, e.g. "the train is leaving from track 3"; otherwise it is a \textit{to_point}.

\[ \text{point} = \text{from_point}(\text{nat}) + \text{to_point}(\text{nat}) \]
\[ \text{duration} = \text{for}(\text{time}) \]

The last definitions complete the scheme by introducing some singleton descriptions in enumerative type equations.

\[ \text{object} = \text{train} + \text{airplane} \]
\[ \text{city} = \text{bologna} + \text{genoa} + \text{milan} + \text{parma} + \text{turin} \]
\[ \text{time} = \text{hour}(\text{nat}) \times \text{minute}(\text{nat}) \]
\[ \text{period} = \text{minute} + \text{hour} + \text{day} + \text{week} \]

\text{nat}, as shown in the past chapters, is the built-in concept for integers.

### 5.2.2 REPRESENTATION OF EXAMPLE SENTENCES

In this paragraph we are going to write some sentences, both in English and in Italian, and then to represent them in the common semantic form introduced above.

Let us consider, as a starter, the very simple assertion:

1) \textit{The train from Parma goes to Milan.}

The given parts of the event in question are the vehicle (a train), the source city (Parma) and the destination city (Milan). Nothing is said about schedule, point, frequency or duration. This kind of situation is easily treatable in this model by taking the general concept of \textit{event} and by specializing it by means of products with the known elements. Sentence 1 looks then as follows:

1) \textit{designated(event \times \text{by}(\text{train}) \times \text{from}(\text{parma}) \times \text{to}(\text{milan}))}

The complex term “hidden” by this frame and composed of individual concepts “just above” the bottom of the lattice, unwinds as follows:

\[ \text{designated}(\text{by}(\text{train}) \times \text{from}(\text{parma}) \times \text{to}(\text{milan}) \times \text{at}(\text{hour}(\text{nat}) \times \text{minute}(\text{nat})) \times \text{repeated}(\text{times}(\text{nat}) \times \text{per}(\text{minute} + \text{hour} + \text{day} + \text{week}) + \text{every}(\text{hour}(\text{nat}) \times \text{minute}(\text{nat}))) \times \text{for}(\text{hour}(\text{nat}) \times \text{minute}(\text{nat}))) \]

The intricacy of this expression is a justification itself for having terms with inheritance capabilities, as a specification from a more general concept can save much work and keep the code in a clean and elegant form.

An alternative solution would have been to introduce for each of the attributes of an event a value indicating the "absence" of that attribution. So, for instance, \textit{source} would be rewritten as:

\[ \text{source} = \text{from}(\text{city} + \text{no_source}) \]

In this case, \textit{frequency} needs to be reformulated in the alternative way presented in the previous paragraph, as we can then insert a concept \textit{no_frequency} in its definition:
\[ \text{frequency} = \text{repeated}(\text{times(nat)} \times \text{per(period)} + \text{every(time)} + \text{no_frequency}) \]

Sentence 1 should then be represented by a frame like:

\[ \text{designated(by(train)} \times \text{from(parma)} \times \text{to(milan)} \times \text{at(no_schedule)} \times \text{repeated(no_frequency)} \times \text{for(no_duration)}) \]

It should be observed that the product with the concept event is dropped as all its attributions are already specified.

The Italian version of sentence 1 - but this is just a curiosity at this point of the discussion - is:

1) Il treno proveniente da Parma va a Milano.

An interesting remark is that the same sentence, though very simple, already presents some differences in the syntax of the two languages. A word-to-word mapping (something a small child could do by looking up a dictionary) would have produced the same thing without the word "proveniente", which means "coming". Any Italian would understand the latter version, but the former one is the one the Italian speaker would use and, therefore, the most convenient one. The importance of the role of a syntactical layer is then clear and more of this will be shown in the next section. If now the small child repeats the same process in the opposite sense, what comes out is the perfectly legal English sentence:

\[ \text{The train coming from Parma goes to Milan.} \]

Also, this sentence is semantically absolutely equivalent to sentence 1. In our model variants are available that have different syntactical formulations and slightly different meanings, as one might object, but that represent the same event. For instance:

\[ \text{The train goes from Parma to Milan.} \]

(which in this case translates to Italian without the word "proveniente"). One immediately notices that the prepositional phrase "from Parma" is now linked to the verb "goes", whereas in the first version it was attached to the determined noun "the train". The syntax-to-semantics end takes care of getting rid of all the variants and mapping them into a unique representation. For the semantics-to-syntax end there will then be, as explained later on in the chapter, the opportunity of choosing between either using one sentence representative for all its variants or giving multiple answers enumerating all the variants.

If the event designated in sentence 1 is regarded as the context of a question whose interest is the "truth", in the sense explained in the previous paragraph, as in the frame:

\[ \text{asked(truth)} \times \text{context(event } \times \text{by(train)} \times \text{from(parma)} \times \text{to(milan)}) \]

then the represented sentence is:

2) Does the train from Parma go to Milan?

which translates in Italian to:

2) Il treno proveniente da Parma va a Milano?

One may notice that, in Italian, the only difference with the assertion is the presence of the question mark (and the intonation, when the sentence is pronounced, the manner of speaking being characterized by the rise and fall of the pitch of the voice). In fact, the
Italian language lacks auxiliary interrogative locutions such as “do” and the inversion of verb and subject is seldom used, except for rare cases yet with a possible slight change of meaning. For instance, the alternative formulation:

*Va a Milano il treno proveniente da Parma?*

where the verb phrase is placed before the noun phrase, puts the stress on the word “Milano”, meaning that what the speaker is interested in is the fact that the train goes to Milan and not anyplace else. Once again this is a syntactical issue and all the details are postponed to the next section. However, it should be remarked that nuances of this kind disappear at the semantic level.

Let us consider now the sentence:

3) **Where is the train from Parma going to?**

In this case the destination is unknown and therefore is the interest of the question, which maps then to the frame:

3) \( \text{asked(destination)} \times \text{context(event} \times \text{by(train)} \times \text{from(parma)}) \)

Several other possibilities exist for restating this sentence by keeping (almost) the same meaning: for instance one could omit the word “to” at the end or use the interrogative form with “does” instead of the present continuous. One way of writing a semantically equivalent sentence in Italian is the following:

3) **Dove va il treno proveniente da Parma?**

One could argue that it would be more precise to translate the English construct

“to be” + verb in “ing” form

with the Italian

“stare” + gerund

so that “va” would be replaced by “sta andando”\(^{56}\). However, this concern does not apply here, as all these syntactical details appear blurred at the semantic level.

The last example we shall illustrate thoroughly is the following:

4) **Where is the train from Parma?**

This is interesting because the word “track”, though not mentioned, is what “where” means in this context. A less ambiguous formulation could, for instance, be:

*Which track is the train from Parma arriving to?*

Both sentences map, of course, to the same frame, namely:

4) \( \text{asked(to-point(nat))} \times \text{context(event} \times \text{by(train)} \times \text{from(parma)}) \)

One valid Italian translation is

\(^{56}\) “Andare” is the infinitive of the irregular verb form “va”, which is the verb form for third person singular of the present simple; “andando” is the gerund.
4) *Dov’è il treno proveniente da Parma?*  
(literally “where’s the train coming from Parma?”), or, alternatively,  
*A quale binario arriva il treno proveniente da Parma?*
(literally, “to which track arrives the train coming from Parma?”).

Having shown the general pattern and the most important problems, other example sentences are just put in table 5.2.2.1 together with their semantic representation and the Italian translation. Further issues are discussed in the next section.

<table>
<thead>
<tr>
<th>English and Italian text</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 The train from Parma goes to Milan. Il treno proveniente da Parma va a Milano.</td>
<td>designated(event × by(train) × from(parma) × to(milan))</td>
</tr>
<tr>
<td>2 Does the train from Parma go to Milan? Il treno proveniente da Parma va a Milano?</td>
<td>asked(truth) × context(event × by(train) × from(parma) × to(milan))</td>
</tr>
<tr>
<td>3 Where is the train from Parma going to? Dove va il treno proveniente da Parma?</td>
<td>asked(destination) × context(event × by(train) × from(parma))</td>
</tr>
<tr>
<td>4 Where is the train from Parma? Dov’è il treno proveniente da Parma?</td>
<td>asked(to_point(nat)) × context(event × by(train) × from(parma))</td>
</tr>
<tr>
<td>5 Where does this train come from? Da dove viene questo treno?</td>
<td>asked(source) × context(event × by(train))</td>
</tr>
<tr>
<td>6 When is the train for Parma leaving? Quando parte il treno per Parma?</td>
<td>asked(schedule) × context(event × by(train) × to(parma))</td>
</tr>
<tr>
<td>7 Where is gate 7? Dov’è l’uscita 7?</td>
<td>asked(in(7)) × context(event × by(airplane))</td>
</tr>
<tr>
<td>8 How long does it take for the train from Parma to go to Milan? Quanto impiega il treno proveniente da Parma per andare a Milano?</td>
<td>asked(duration) × context(by(train) × from(parma) × to(milan))</td>
</tr>
<tr>
<td>9 How often does a train go to Milan? Ogni quanto un treno va a Milano?</td>
<td>asked(frequency) × context(by(train) × to(milan))</td>
</tr>
<tr>
<td>10 The train for Milan leaves every 35 minutes. Il treno per Milano parte ogni 35 minuti.</td>
<td>designated(event × by(train) × to(milan) × every(hour(0) × minute(35)))</td>
</tr>
</tbody>
</table>

*table 5.2.2.1*
5.3 **The syntax**

5.3.1 **Scheme of the syntax**

In the previous section we saw what meanings can be expressed but we did not see how. Basically, we did not define all the English and Italian words that can be used (we will propose just a very small dictionary) neither did we show the constructions allowed for building a sentence. In other words, we need now to define the grammar of our model.

First we describe a conceptual model by means of **OBJECTLOG** frames, as we did for the semantics, and then, in the next section, we illustrate how a text can be converted to such a frame with little effort thanks to the well-known DCG-grammar principle. Two different schemes are presented, one for English and one for Italian, as, even though many similarities are present, some remarkable incompatibilities prevent us from treating them as a whole.

5.3.2 **English syntax**

As for the semantics, a sentence can syntactically be either an assertion or a question.

\[ \text{sentence} = \text{assertion} + \text{question} \]

This type equation is the same as the first type equation given in the scheme for the semantics, with the only difference that we dropped the prefix `semantics_of_`.

An assertion is a noun phrase and a verb phrase put together plus zero or more complements. A complement is a part of a sentence that gives more information about the subject, or, in some structures, about the object ([26]). Complements can be linked both to noun and verb phrases. Furthermore, in the cases we are interested in, the verb can be in -ing form or in the third person singular of the present simple tense. This last variation in the verb form is often referred to as concord or agreement. This relation can be expressed in a number of different ways. Our choice here is to introduce an attribute for each type; inside the attribution we refer to other elements that are either constants or recursively types defined in the same way. The definition of `assertion` looks then as follows:

\[ \text{assertion} = a(\text{noun_phrase} \times (\text{agreement_verb_phrase} + \text{ing_verb_phrase})) \]

It would have been, of course, equivalent to introduce a type equation for the inner sum term enclosed between parenthesis, like:

\[ \text{assertion_verb_phrase} = \text{agreement_verb_phrase} + \text{ing_verb_phrase} \]

and then to restate the definition in the following way:

\[ \text{assertion} = a(\text{noun_phrase} \times \text{assertion_verb_phrase}) \]
but this does not really constitute a change of style. A different way of writing these type equations would be, for instance, that of omitting the outer attributes and to put immediate attributes for the inner types:

\[
\text{assertion} = \text{np}(\text{noun phrase}) \times (\text{agr}_\text{vp}(\text{agreement}_\text{verb phrase}) + \text{ing}_\text{vp}(\text{ing}_\text{verb phrase}))
\]

Another possibility is to drop, at this level, the distinction between the various verb forms and to identify them below in the lattice by marking them with a "flag" (in this case form):

\[
\text{assertion} = \text{np}(\text{noun phrase}) \times \text{vp}(\text{verb phrase} \times \text{form}(\text{agreement} + \text{ing}))
\]

All the variants are equally good for our purposes; the first one is, however, the one we shall stick to in the following.

A question is somewhat more complicated than an assertion, in that it can have an optional interrogative part. The verb can be either in plain form (when the auxiliary construct "does" is used) or in -ing form; yet another case is when the verb "to be" ("is") is used alone as copula to link a subject to its complements. An optional part of a sentence can be schematized in several ways; the technique used in the following is to write a disjunction between the optional part and an "empty" attribution defined as:

\[
\text{empty} = \text{no}_\text{structure}(\text{void})
\]

where void is just a concept different from \perp. The following definition has now a clear meaning:

\[
\text{question} = q((\text{interrogative} + \text{empty}) \times \text{noun phrase} \times (\text{plain}_\text{verb phrase} + \text{ing}_\text{verb phrase} + \text{copula}))
\]

Notice that one could define question in a simpler way by considering that the presence of an interrogative can be expressed by specializing the following concept:

\[
\text{simplified_question} = q(\text{noun phrase} \times (\text{plain}_\text{verb phrase} + \text{ing}_\text{verb phrase} + \text{copula}))
\]

\[
\text{simplified_question} \times q(\text{interrogative})
\]

is, in fact, a specialization of the concept simplified_question. However, we avoid this formulation because if there were many optional parts, the definition would be very poor, even empty if all the elements were optional. As a consequence of absorption, it turns out that a type definition with a mandatory attribute \( b \) and an optional attribute \( a \) is not expressible if there is no means of indicating the absence of the optional attribute, as clarified from the following line:

\[
b(x) + a(x) \times b(x) \equiv b(x) \times (\top + a(x)) \equiv b(x) \times \top \equiv b(x)
\]

The empty attribution will be used several times in this section. An alternative technique would be that of using a different "empty flag" for every optional field. So, in this case, we would have a concept similar to no_interrogative(void) and other names would be used for the other occurrences of empty. This approach turns out to be not particularly useful here and therefore we will always use empty wherever it is needed.

An interrogative can be one of the following constructions:
- an interrogative adjective and a noun, like "what time", "which track", etc.;
- an interrogative adverb modified by another adverb, like "how often" and "how long";

\[57\] Notice that, in order to compile this program with the OBJECTLOG\(_2\) compiler, the type definitions must respect the constraint shown in paragraph 2.2.1. Every type definition given in this chapter can be rewritten by means of OBJECTLOG\(_2\) type definitions. The interested reader can consult appendix B.
• an interrogative adverb alone, like "when" or "where". and therefore be defined as

\[
\text{interrogative} = \text{int(} \text{interrogative_adjective} \times \text{noun} + \text{interrogative_adverb} \times \\
(\text{modifying_adverb} + \text{empty})\text{)}
\]

A noun phrase can be:
• a (possibly determined) noun and a list of zero or more complements linked to that noun, like "the train for Parma";
• a pronoun, like "it" and "this";
• a proper noun, like "Milan".
We distinguish between \text{noun_complement_list} and \text{verb_complement_list} because we consider a subordinate clause as a complement and this cannot be attached to a noun phrase. Furthermore there are some constructions for verb phrases that are not allowed for noun phrases.

\[
\text{noun_phrase} = \text{np(} (\text{determiner} + \text{empty}) \times \text{noun} \times \text{noun_complement_list} + \\
\text{pronoun} + \text{proper_noun})\text{)
\]

The three kinds of verb phrase unwind as follows:
\[
\text{plain_verb_phrase} = \text{pvp(verb_phrase)}
\]
\[
\text{agreement_verb_phrase} = \text{avp(verb_phrase)}
\]
\[
\text{ing_verb_phrase} = \text{ivp(verb_phrase)}
\]
\[
\text{verb_phrase} = \text{verb} \times \text{verb_complement_list}
\]

Lists are treated, as explained in the first chapter, by means of two attributions, one for the head and one for the tail:

\[
\text{noun_complement_list} = \text{ncl(nc_list)}
\]
\[
\text{nc_list} = \text{nil} + \text{h(noun_complement)} \times \text{t(nc_list)}
\]

A Prolog-like syntactic sugar for lists is allowed:

\[
\text{verb_complement_list} = \text{vcl(vc_list)}
\]
\[
\text{vc_list} = [\] + [\text{verb_complement} | \text{vc_list}]
\]

A noun complement can be:
• a prepositional phrase, like "from Parma", "to gate 1", etc.;
• an adjective (see below the definition of an adjective);
• a "quantified" clause, like "every 2 hours", so-called because of the usage of "every".
The same holds for verb complements, with the additional case of subordinate clauses and a slightly different behavior of prepositional phrases:

\[
\text{noun_complement} = \text{prepositional_phrase} + \text{adjective} + \text{quantified_clause}
\]
\[
\text{verb_complement} = \text{verb_prepositional_phrase} + \text{adjective} + \text{quantified_clause} + \\
\text{sub_clause}
\]

The variant in prepositional phrases for verbs is that a preposition can be used which does not refer to a noun phrase, like in the sentence "where is this train going to?". The phrase "going to" is the verb "going" plus the preposition "to"; this is not to be confused with a prepositional verb (often also referred to as "phrasal verb"), like "go for", where the preposition modifies the meaning of the verb. This is the reason why in the next definition the noun phrase is treated as an optional.
verb_prepositional_phrase = pp(preposition \times (noun_phrase + empty)).
prepositional_phrase = pp(preposition \times noun_phrase)

An adjective can be:
• a present participle, like "coming";
• a numeral, like "7";
• a quantifier, like "every". The reason for treating "every" differently from all the other adjectives is that it can introduce a complement (a time expression answering the question "how often?"), whereas in all the other cases the complements are introduced by a preposition.

\[
\text{adjective} = \text{adj}(\text{present_participle} + \text{numeral} + \text{quantifier})
\]

The only kind of subordinate clause we allow here is an objective clause in implicit form, that is an infinitive used after another verb. For instance, in the sentence "how long does it take to go to Genoa", "to go" is the subordinate clause. Therefore the only thing we need to know about a subordinate clause is its verb:

\[
\text{sub Clause} = \text{sc}(\text{verb})
\]

The time complement introduced by "every" has also an optional numeral and a noun, like "every 2 days" or "every hour":

\[
\text{quantified Clause} = \text{qc}(\text{quantifier} \times (\text{numeral} + \text{empty}) \times \text{noun})
\]

The remaining definitions are the ones for the elements closest to the bottom. Except for \text{numeral}, which is defined on the built-in concept \text{nat}, all the others are based on "dictionaries" of words, that is concepts defined by an enumerative type equation whose right-hand side is a disjunction of constants (the words).

\[
\begin{align*}
\text{present_participle} &= \text{pres_part}(\text{present_participle_dictionary}) \\
\text{numeral} &= \text{num}() \\
\text{quantifier} &= \text{quant}(\text{quantifier_dictionary}) \\
\text{verb} &= \text{v}(\text{verb_dictionary})
\end{align*}
\]

A noun can be either singular or plural:

\[
\begin{align*}
\text{noun} &= \text{n}(\text{noun_dictionary}) + \text{plur}_n(\text{noun_dictionary}) \\
\text{pronoun} &= \text{p}(\text{pronoun_dictionary}) \\
\text{preposition} &= \text{prep}(\text{preposition_dictionary}) \\
\text{proper_noun} &= \text{pn}(\text{proper_noun_dictionary}) \\
\text{determiner} &= \text{det}(\text{determiner_dictionary}) \\
\text{interrogative_adjective} &= \text{int_adj}(\text{interrogative_adjective_dictionary}) \\
\text{interrogative_adverb} &= \text{int_adv}(\text{interrogative_adverb_dictionary}) \\
\text{modifying_adverb} &= \text{mod_adv}(\text{modifying_adverb_dictionary})
\end{align*}
\]

Next are the dictionaries. Please, notice that not all the possible words belonging to the category indicated by the left-hand side are listed. So, for instance, the article "an" is not present because we do not have any noun beginning with a vowel.

\[
\begin{align*}
\text{verb_dictionary} &= \text{leave} + \text{depart} + \text{arrive} + \text{go} + \text{come} + \text{take} \\
\text{noun_dictionary} &= \text{train} + \text{flight} + \text{airplane} + \text{track} + \text{gate} + \text{city} + \text{time} + \text{minute} + \text{hour} + \text{day}
\end{align*}
\]
The syntax we deal with is extremely simplified, as words' flexion and order are, in Italian, a major issue, and here we prevented the user from stating particularly complex sentences. In order to have different names for the Italian syntactic concept, we shall use the prefix "i_"; the attribution names do not need to be changed.

As already mentioned in the previous paragraph, we will not use any form corresponding to the English "-ing" form, so there is basically only the third person singular as available verb form. Furthermore, this does not change when asking questions, as Italian does not use any special auxiliary verb (like "does") or interrogative construction (like "est-ce que" in French).

In assertions, the verb phrase normally follows the noun phrase. In questions, the order is commonly the same as assertions if there is no interrogative (like "quando"); otherwise the verb phrase usually precedes the noun phrase.

\[
\begin{align*}
i_{\text{sentence}} &= i_{\text{assertion}} + i_{\text{question}} \\
i_{\text{assertion}} &= a(i_{\text{noun phrase}} \times i_{\text{verb phrase}}) \\
i_{\text{question}} &= q((i_{\text{interrogative}} + \text{empty}) \times i_{\text{noun phrase}} \times i_{\text{verb phrase}})
\end{align*}
\]

Unlike for the English syntax, we allow verb phrases and noun phrases to have exactly the same complements. It should be noticed that in an Italian question the preposition (which is normally attached to a verb in English) is found at the beginning of the sentence together with the interrogative. So, for instance, in the English version of sentence 5, the preposition "from" is at the end of the sentence attached to the verb "come", whereas in the Italian version the preposition "da" (from) is at the beginning preceding the adverb "dove" (where). This means that we can keep the same kinds of complements for verb and noun phrases, but we have to introduce new kinds of interrogative.

\[
\begin{align*}
i_{\text{noun phrase}} &= np((i_{\text{determiner}} + \text{empty}) \times i_{\text{noun}} \times i_{\text{complement list}} + i_{\text{proper noun}}) \\
i_{\text{verb phrase}} &= vp(i_{\verb} \times i_{\text{complement list}})
\end{align*}
\]

In particular, we consider four kinds of interrogatives:

- interrogative adverbs alone, like "quando" (when), "dove" (where), etc.;
- interrogative adverbs preceded by a preposition, like "da dove" (from where), etc.;
- interrogative adverbs preceded by a quantifier, like "ogni quanto" (how often). Here the Italian formulation is considerably different from the English one, as "ogni" means "every" and "quanto" means "how much";
• a noun preceded by an interrogative adjective and a preposition, like "da quale binario" (from which track), etc.

\[\text{interrogative} = \text{int}(\text{i_int1} + \text{i_int2} + \text{i_int3} + \text{i_int4})\]
\[\text{i_int1} = \text{int}(\text{i_interrogative_adverb})\]
\[\text{i_int2} = \text{int}(\text{i_preposition} \times \text{i_interrogative_adverb})\]
\[\text{i_int3} = \text{int}(\text{i_quantifier} \times \text{i_interrogative_adverb})\]
\[\text{i_int4} = \text{int}(\text{i_preposition} \times \text{i_interrogative_adjective} \times \text{i_noun})\]

The kind of subordinate clause we are interested in\(^{58}\) can be translated in Italian with the construction "per" + infinitive, like in sentence 8. The other complements behave like in English.

\[\text{i_complement_list} = \text{cl}(\text{i_c_list})\]
\[\text{i_c_list} = [\text{i_complement} | \text{i_c_list}]\]
\[\text{i_complement} = \text{i_prepositional_phrase} + \text{i_adjective} + \text{i_quantified_clause} + \text{i_sub_clause}\]
\[\text{i_prepositional_phrase} = \text{pp}(\text{i_preposition} \times \text{i_noun_phrase})\]
\[\text{i_adjective} = \text{adj}(\text{i_present_participle} + \text{i_numeral} + \text{i_quantifier})\]
\[\text{i_sub_clause} = \text{sc}(\text{i_verb})\]
\[\text{i_quantified_clause} = \text{qc}(\text{i_quantifier} \times (\text{i_numeral} + \text{empty}) \times \text{i_noun})\]

When a preposition and an article are put together, we have in Italian the so-called contraction of the preposition. Examples of contracted prepositions are "dal" (= "da" + "il"), "al" (= "a" + "il"), "alla" (= "a" + "la") and others. Articles and contracted prepositions vary depending on the gender (masculine or feminine) of the word they determine; furthermore, an apostrophe may be needed if the first letter of the word that follows is a vowel. So, one says "il treno" (the train, masculine), "l'aereo" (the airplane, masculine beginning with vowel), "la città" (the city, feminine) and "l'uscita" (the gate, feminine beginning with vowel).

\[\text{i_prep} = \text{i_simple_preposition} + \text{i_contracted_preposition}\]

The definitions concerning the "leaves" of the lattice are written in the same style as the English syntax and therefore are not reported here. The reader interested in the Italian words used for the translator can find the listing of the program in appendix B.

5.3.4 SYNTACTICAL VIEW OF EXAMPLE SENTENCES

We can now complete the translation of the example sentences by showing what the syntactic representations are, both for English and Italian. It is worth noticing that one of the aims of the syntactic layer was that to be able to represent "perfectly" (that is without loss of information) the text it abstracts from. This is also the reason why the schematization of the syntax is much more complicated than the one for the semantics.

The representation of sentence 1 is shown below in sugared form, both for the Italian and the English version.

---

58 Referred to by the Italian grammars as "proposizione subordinata di tipo finale".
5.4 From text to syntax

5.4.1 The DCG Principle

The general context-free grammar (CFG) rule:

\[ N_0 \rightarrow V_1, V_2, \ldots, V_n \]

can be axiomatized as:

\[ V_i(p_0, p_1) V_2(p_1, p_2) \ldots V_n(p_{n-1}, p_n) \Rightarrow N_0(p_0, p_n) \]

where each \( p_i \) represents a "position". So, there is a grammar symbol (nonterminal) \( N_0 \) between positions \( p_0 \) and \( p_n \) if there is a grammar symbol (terminal or nonterminal) \( V_i \) between positions \( p_{i-1} \) and \( p_i \) for \( 1 \leq i \leq n \). This logical expression can be easily stated in logic programming in the following way:
\[ n_0(P_0, P) \leftarrow v_1(P_0, P_1), v_2(P_1, P_2), \ldots, v_n(P_{n-1}, P_n) \]

Usually a predicate `connect/3` is used to indicate that a terminal (the first argument) is between two consecutive positions (the second and third argument). Any CFG can then be fully re-expressed in `OBJECTLOG` with very little effort.

It is possible to extend this principle in a simple way: the formulation can be enriched by allowing additional arguments further specifying the expression type. This technique is known as the `definite-clause grammar` (DCG) principle and allows a number of syntactic sugars that we will assume in the following\(^\text{59}\).

DCGs are used here for transforming a list of words (the text) in a syntactic concept (as described in the previous section). We have rules where the positions are not mentioned and the only argument is used for each nonterminal for constructing its syntactic concept. For instance, the rule for prepositional phrases looks as follows:

\[ pp(pp(P \times NP)) \rightarrow \text{prep}(P), \text{np}(NP) \]

This means that if we have a preposition whose syntactic representation is \(P\) followed by a noun phrase whose representation is \(NP\), then we recognize a prepositional phrase and crux \(P\) and \(NP\) to form the new concept \(pp(P \times NP)\) as its syntactic representation. If there is an occurring terminal symbol, this is enclosed between square brackets, like in the following rule, where a full stop (which is a terminal symbol) is expected at the end of the sentence:

\[ i\_assertion(a(NP \times VP)) \rightarrow i\_np(NP), i\_vp(\text{VP}), [.] \]

This technique can be applied to all the concepts shown in the previous section both for English and Italian syntax and the complete listing is reported in appendix B. However, for the terminal symbols we will take advantage of another facility of DCGs, as explained in the following paragraph.

### 5.4.2 Word Flexion

In this paragraph, we shall explain how plural of nouns and concord of verbs are formed. Simple rules could be written according to those principles for automatically generating plurals and according verbs. However, although English is particularly well-suited for this purpose, as verbs and nouns are rather simple, Italian, being more irregular and various in the formation of plural and concord, gives rise to several problems. Furthermore, as we have only few words that require flexion, it is more convenient to store this information in facts. Therefore, we did not consider worth the effort writing a predicate for the flexion of nouns and verbs considered in this example.

A sufficiently ample (yet restricted) rule for formation of plural and verb agreement in English is the following:

- if the word ends in \(s, sh, ch, x\) or \(z\), then the plural/agreement form is made by adding `es` to the word (ex.: `catch` \(\rightarrow\) `catches`, `miss` \(\rightarrow\) `misses`, `box` \(\rightarrow\) `boxes`, etc.);
- if the last letter is \(y\) preceded by vowel, then \(s\) should be added (ex.: `day` \(\rightarrow\) `days`, `buy` \(\rightarrow\) `buys`, etc.);

\(^{59}\) The interested reader can refer to [24] for a detailed explanation of how DCGs can be translated in Prolog. The general technique is however very similar to the one shown for CFGs. As a matter of fact, many Prolog systems include DCGs as a built-in facility. The ObjectLog formulation would almost coincide with Prolog, therefore we need not give any further clarification. More on DCGs is said in paragraph 4.3.4.
• if the last letter is y preceded by consonant, then y is replaced by ies (ex.: fly → flies, city → cities);
• do and go become does and, respectively, goes;
• in all the other cases s is added.

An even simpler rule holds for the "-ing" form:\footnote{60}{It should be noticed that not all the cases are considered. There are many verbs that double the ending consonant, depending also on the pronunciation.}
• if the verb ends in e not preceded by another e, then the e is replaced by ing (ex.: leave → leaving);
• in all the other cases ing is added.

In Italian we shall consider a very restricted rule for the formation of the plural that does not hold for all the categories of nouns:
• città (city) is invariable;
• nouns ending in o (masculine gender) replace o by i (ex.: treno → treni);
• nouns ending in a (feminine gender) replace a by e (ex.: uscita → uscite).

The formation of the third person singular of single present of verbs from the infinitive takes place as follows:
• verbs with an exceptional behavior are the following: essere (to be) → è, andare (to go) → va, venire (to come) → viene;
• there is a category of verbs (all in this example) ending in ire that replace ire by e (ex.: partire → parte);
• all the others (in this example) leave out the ending re (ex.: arrivare → arriva).

We shall now focus on another topics, namely the embedding of calls in DCGs. DCGs provide a mechanism for specifying arbitrary computations over the logical variables through execution of goals, which are distinguished from the grammatical elements notationally by being enclosed between curly brackets. We shall make this clearer by means of an example.

If a predicate det/1 is fulfilled by all the determiners of the language, that is it defines the dictionary of determiners, then the relevant rule can be written as:
\[
\text{det}(\text{det}(D)) \rightarrow [D], \{ \text{det}(D) \} \footnote{61}{Predicates are distinguishable from attributes as the latter are written in Italics.}
\]
meaning that if a terminal D is found which fulfills the predicate det/1, then the concept det(D) is formed. The same principle can be used for all the categories of dictionaries. When it comes to plural formation and verb agreement, the rules can be written with the following style:
\[
\text{n}(\text{plur}_n(\text{SN})) \rightarrow [N], \{ \text{n}(\text{SN}), \text{agreement}(\text{SN}, N) \}
\]
This means that we can form a noun whose concept is plur_n(SN) if SN is in the dictionary of (singular) nouns and N is the plural form of SN, being N the terminal in question. agreement is the predicate whose clauses are facts associating plain verb forms with their agreement verb form.
The DCG ability to add arguments to the nonterminals in a grammar is useful in a variety of ways. As shown in the previous paragraph, the argument-passing in DCGs is used in our grammar not only to recognize the strings of the English or Italian text, but also to build a representation of the concept designated by the sentence, encoded as an \texttt{OBJECTLOG} term.

Terms can here be seen as partially specified parse trees in which variables correspond to yet unspecified subtrees. For example, the term

\[
a(np(det(DET) \times n(N) \times ncl(NCL)) \times avp(VP))
\]

corresponds to the (partial) tree shown in fig. 5.4.3.1 in which the variables may be replaced by any trees.

```
   a
  /   \
 np   avp
   |
  det n ncl
   |
 DET N NCL
```

\textbf{fig. 5.4.3.1}

Every nonterminal predicate has an argument representing the tree for the portion of the string covered by the nonterminal. For each nonterminal we introduce a homonymous symbol to represent a node of that type. All the sentences considered in this chapter can be then associated with a tree like the one shown in fig. 5.4.3.2 for sentence 1, where the terminals are marked in boldface.
5.5 From syntax to semantics

5.5.1 Understanding the actions

In this section we are going to explain how the syntactical representation of a sentence maps to its semantics. The key point in this is to determine which combinations of syntactical constituents are valid and can delineate meanings belonging to our example. In this paragraph the general procedure for English sentences is sketched and some of the clauses are also reported in order to illustrate the main idea.

The predicate `syntax2semantics/2`, whose first (input) argument is the syntactic representation and whose second (output) argument is the semantics, considers separately questions and assertions.

\[
syntax2semantics(q(Q), \text{asked}(I) \times \text{context}(C \times \text{by}(Object))) \leftarrow \text{question2interest}(Q, I, Object), \text{question2context}(Q, I, C).
\]

A question is semantically characterized by the `interest` and the `context`, as shown in section 5.2. With the predicate `question2interest/3` we map a question to its interest and to the vehicle (train or airplane) the question refers to. The predicate `question2context/3` takes
the context out of a question and its interest. The formulation for assertions is similar (and simpler):

\[
syntax2semantics(a(A), designated_event(E)) \leftarrow assertion2designated_event(A, E).
\]

A question can either have an interrogative expression, in which case we get the interest from it and possibly specify it more precisely from the rest of the sentence:

\[
question2interest(int(Interrogative) \times Rest, Interest, object \times O) \leftarrow get\_interest\_and\_object(Interrogative, Int, O), refine\_interest(Int, Rest, Ref\_Int), min(Int, Ref\_Int, Interest).
\]

or present an empty structure, in which case the interest is truth.

\[
question2interest(empty \times np(_), VP, truth, object \times _) \leftarrow isa(VP, plain\_verb\_phrase + ing\_verb\_phrase + empty).
\]

Notice that the value returned for the vehicle is object \times O, which reduces to train or airplane if O is specified or to something less or equal to object if O is unbound. Conversely, the interest is returned as Int possibly refined by Ref\_Int; if, in some way, those two values are in conflict, their product reduces to \perp. The refinement takes place with the \times operator, which returns the most specific between the two concepts. This procedure corresponds, in this particular case, to intersection of attributions, which is what is desired. However, the crux does not work in the general case, as cruxing over attributions means tupling. Let us consider the case where the interest is source + destination and the refined interest is source; the product of these two concepts unwinds as follows:

\[
(source + destination) \times source = (from(city) + to(city)) \times from(city) = from(city) + to(city) \times from(city)
\]

which reduces to from(city) as expected, being this concept more general than the other one in the sum, namely to(city) \times from(city). It is possible to obtain the same result with the predicate min/3\textsuperscript{62} explained below, which furthermore has the advantage of returning \perp if the two concepts are not in an inclusion relation. In the example, one comes to the same conclusion by simply noticing that source is the most specific of the two concepts in question. min/3 is defined as follows:

\[
min(A, B, B) \leftarrow isa(B, A).
min(A, B, A) \leftarrow isa(A, B).
min(A, B, \perp).
\]

We propose two different styles for the predicate get\_interest\_and\_object/3. With the first one we use is-a clauses to test the presence of a given structure in the concept:

\[
get\_interest\_and\_object(Interrogative, point, train) \leftarrow isa(Interrogative, n(track)).
\]

so that in this case if the interrogative contains the noun "track" (like in "which track"), then the interest is a point and the vehicle is a train. The second style does not need is-a clauses because the whole interrogative structure is already specified:

\[
get\_interest\_and\_object(int\_adj(what) \times n(time), schedule, _).
\]

\textsuperscript{62} In the general case, the predicate min/3 does not work either, but it is good enough for the cases we need to consider.
Notice that in this last case the vehicle is undetermined and therefore an unbound variable is returned. Instead of showing all the clauses of the predicate in question, we exhibit table 5.5.1.1, which reports the relations among interrogatives, interests and vehicles.

<table>
<thead>
<tr>
<th>Notion about the interrogative</th>
<th>Interest</th>
<th>Object (vehicle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>isa(Interrogative, n(track))</td>
<td>point</td>
<td>train</td>
</tr>
<tr>
<td>isa(Interrogative, n(gate))</td>
<td>point</td>
<td>gate</td>
</tr>
<tr>
<td>isa(Interrogative, n(city))</td>
<td>source + destination</td>
<td>_</td>
</tr>
<tr>
<td>int_adj(what) × n(time)</td>
<td>schedule</td>
<td>_</td>
</tr>
<tr>
<td>int_adv(when) × empty</td>
<td>schedule</td>
<td>_</td>
</tr>
<tr>
<td>int_adv(where) × empty</td>
<td>source + destination + point</td>
<td>_</td>
</tr>
<tr>
<td>int_adv(how) × mod_adv(long)</td>
<td>duration</td>
<td>_</td>
</tr>
<tr>
<td>int_adv(how) × mod_adv(often)</td>
<td>frequency</td>
<td>_</td>
</tr>
</tbody>
</table>

**table 5.5.1.1**

If the interest is already atomic, no further specification is needed.

```
refine_interest(Int, _, Int) ← atomic(Int).
```

In cases 1, 2, 3 and 6 the interest is not fully specified and therefore we need to refine it with the predicate `refine_interest/3`.

```
refine_interest(_, Rest, Ref_Int) ← get_main_action(Rest, Action),
refine_from_action(Action, Ref_Int).
```

An "action" is understood from the non-interrogative part of the sentence (the variable `Rest`) and then the refined interest is taken from the action. An action is basically the verb in the verb phrase possibly together with a preposition representing a movement of the vehicle. Each action is associated with an interest. For instance, in the sentence "where is the train coming from Parma going to?", the action is `go_to`. All the possible combinations are shown in table 5.5.1.2:

<table>
<thead>
<tr>
<th>Action</th>
<th>Refined interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 arrive_to</td>
<td>to_point(nat)</td>
</tr>
<tr>
<td>2 arrive_from</td>
<td>source</td>
</tr>
<tr>
<td>3 go_to</td>
<td>destination</td>
</tr>
<tr>
<td>4 leave_for</td>
<td>destination</td>
</tr>
<tr>
<td>5 leave_from</td>
<td>source</td>
</tr>
<tr>
<td>6 depart_from</td>
<td>from_point(nat)</td>
</tr>
<tr>
<td>7 come_from</td>
<td>source</td>
</tr>
<tr>
<td>8 be</td>
<td>point</td>
</tr>
<tr>
<td>9 be_from</td>
<td>to_point(nat)</td>
</tr>
<tr>
<td>10 be_to</td>
<td>from_point(nat)</td>
</tr>
</tbody>
</table>

**table 5.5.1.2**

Wherever ambiguities arise, they are resolved arbitrarily; for instance a sentence like "where is this train leaving from?" (case 5) could also mean "which city ...", but it is understood here as "which track ...". These associations are simply dealt with by the predicate `refine_from_action/2`, whereas the action itself is obtained by `get_main_action/2`.
by means of some structure manipulations. Assumptions are also made that verbs like "go" and "leave", if written without an explicit preposition, mean "go to" and, respectively, "leave for". "Be" is considered an exception to the above cases, in that it is an action without having a preposition. Cases 9 and 10 require inspection of the noun phrase, like in sentence 4.

Now the matter is to understand the context of a question, which is done by the predicate question2context/3. The interest is also passed to this predicate, so that particular cases, like sentences with "be" or questions about duration, can be treated separately. Notice that questions about duration ("how long does it take ...") return as context the term \( \text{CNP} \times C \times (\text{to}(\text{City}) + \text{from}(\text{City})) \), where \( \text{CNP} \) is the context taken out of the noun phrase, \( C \) is the context refined from the action contained in the verb of the subordinate clause and \( \text{City} \) is the city mentioned in the question. For example sentence 8 \( \text{CNP} = \text{from}(\text{parma}), \ C = \text{destination} \) and \( \text{City} = \text{milan} \). The product in question is \( \text{from}(\text{parma}) \times \text{destination} \times (\text{to}(\text{milan}) + \text{from}(\text{milan})) \), which clearly reduces to \( \text{from}(\text{parma}) \times \text{to}(\text{milan}) \) as expected by virtue of the annihilation of \( \text{parma} \times \text{milan} \).

However, OBJECTLOG cannot handle such an expression because there is a sum of terms containing variables. Therefore, instead of having the clause:

\[
\text{question2context}(\ldots, \text{duration}, \text{CNP} \times C \times (\text{to}(\text{CITY}) + \text{from}(\text{CITY})) \leftarrow \ldots)
\]

we need to make explicit the disjunction expressed by the + by writing two separate clauses

\[
\text{question2context}(\ldots, \text{duration}, \text{CNP} \times C \times \text{to}(\text{CITY}) \leftarrow \ldots) \\
\text{question2context}(\ldots, \text{duration}, \text{CNP} \times C \times \text{from}(\text{CITY}) \leftarrow \ldots)
\]

For ordinary cases, the verb phrase and the noun phrase are separated and transformed to a context, whose product is the resulting context. The transformation is handled by predicates np2context/2 and vp2context/2.

The procedure shown above for questions is very similar to the one used for assertions and therefore the predicate assertion2designated_event/2 need not be commented.

### 5.5.2 Actions in Italian

The scheme used for Italian perfectly adheres to the one presented in paragraph 5.5.1 for English. Similar predicates are then written for the Italian interface, distinguishable by the prefix "i_". We only show here table 5.5.2.1 and table 5.5.2.2, which report the correspondences among interrogatives, actions, vehicles and interests.

<table>
<thead>
<tr>
<th>Notion about the interrogative</th>
<th>Interest</th>
<th>Vehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>int1(int_adv(dove))</td>
<td>destination + to_point(nat)</td>
<td>_</td>
</tr>
<tr>
<td>int2(prep(da) × int_adv(dove))</td>
<td>source + from_point(nat)</td>
<td>_</td>
</tr>
<tr>
<td>isa(Interrogative, int4(prep(a) × n(binario))))</td>
<td>to_point(nat)</td>
<td>train</td>
</tr>
<tr>
<td>isa(Interrogative, int4(prep(a) × n(uscita))))</td>
<td>to_point(nat)</td>
<td>airplane</td>
</tr>
<tr>
<td>isa(Interrogative, int4(prep(da) × n(binario))))</td>
<td>from_point(nat)</td>
<td>train</td>
</tr>
<tr>
<td>isa(Interrogative, int4(prep(da) × n(uscita))))</td>
<td>from_point(nat)</td>
<td>airplane</td>
</tr>
<tr>
<td>isa(Interrogative, int4(prep(a) × n(città))))</td>
<td>destination</td>
<td>_</td>
</tr>
</tbody>
</table>
Actions and interests are associated as shown in the following table. Redundancies are cancelled by means of the min/3 predicate applied to interest and refined interest. For instance the sentence "da dove parte il treno per Milano?" has interest source + from_point(nat) (case 2) and refined interest from_point(nat), which clearly reduces to from_point(nat).

The predicate i_question2context/3, as for English, treats particular cases separately, like questions about duration. A little converter from Italian words to English concepts (called eng2ita/2) is also needed, as shown, for instance, in appendix B for predicate i_get_from_qc/2.

### 5.6 Synthesis of sentences

#### 5.6.1 From semantics to syntax

In this section we are going to discuss how the semantics of a sentence can be mapped back to its syntactic representation. As a nice property of DCGs, the predicates sentence/3 and i_sentence/3 are perfectly reversible, so that the syntactic concept can be given as input and the word list is returned as output. This means that the syntax-to-text end need not be programmed.

Similarly to the other paragraphs, we present a predicate semantics2syntax/2 that does the following work:

- distinguishes between questions and assertions in the usual way and preliminarily retrieves the vehicle (which is always mentioned in the semantic formulation);
- for questions, composes the interrogative related to the interest and, if possible, gives the action to be used with it. For instance, if the interest is a destination the most
suitable interrogative would be "which city" with the action "go to". For other interests, like truth, the action is not immediately determinable;

- draws the syntactic representation of the event based on the interest (for questions only), the rest of the sentence, the suggested action (if any) and the vehicle. This operation is performed by the predicate express_event/5.

Interests, interrogatives and actions are related as shown in table 5.6.1.1:

<table>
<thead>
<tr>
<th>Interest</th>
<th>Interrogative</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 truth</td>
<td>empty</td>
<td>none</td>
</tr>
<tr>
<td>2 source</td>
<td>int(int_adj(which) × n(city))</td>
<td>come_from</td>
</tr>
<tr>
<td>3 destination</td>
<td>int(int_adj(which) × n(city))</td>
<td>go_to</td>
</tr>
<tr>
<td>4 schedule</td>
<td>int(int_adj(what) × n(time))</td>
<td>none</td>
</tr>
<tr>
<td>5 point</td>
<td>int(int_adv(where) × empty)</td>
<td>be</td>
</tr>
<tr>
<td>6 from_point(nat)</td>
<td>int(int_adj(which) × n(Point))</td>
<td>depart_from</td>
</tr>
<tr>
<td>7 to_point(nat)</td>
<td>int(int_adj(which) × n(Point))</td>
<td>arrive_to</td>
</tr>
<tr>
<td>8 frequency</td>
<td>int(int_adv(what) × mod_adv(often))</td>
<td>none</td>
</tr>
<tr>
<td>9 duration</td>
<td>int(int_adv(how) × mod_adv(long))</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 5.6.1.1

For cases 6 and 7, Point is bound to track or gate depending on the vehicle; for cases 1, 4, 8 and 9 nothing can be said about the action and a special concept (none) is used for this purpose.

express_event/5 considers separately cases 5 and 9, as the syntactical formulation is more complicated than elsewhere, and handles the other cases as follows:

- if the action is none, the action and the verb complements are retrieved from the event with the predicate take_vcl_and_action/4. The noun phrase is "the train" or "the flight", depending on the vehicle, with no complements; all the complements are linked to the verbs;
- if the action is already determined, the complements are formed with the predicate take_ncl/4 and then attached to the noun phrase, which is again "the train" or "the flight". The verb phrase is composed by the verb plus a preposition, like "go to". An action is also determined by this predicate; if the two actions are conflicting, their product annihilates to ⊥.

take_ncl/4 (and take_vcl_and_action/4) picks an attribution and puts the corresponding prepositional phrase in a list and then recursively continues with the remaining part of the event. The recursion stops when the remaining part of the event is T. take_vcl_and_action/4 behaves in the same way, but it furthermore considers frequencies, departure points and arrival points. The action is not always univocally determined: if, for instance, a source event constituent is found, the possible actions are "come from", "go to" (if afterwards a destination constituent is found) or "arrive to" (if afterwards an arrival point constituent is found). The predicate action2verb/3 takes care of the cases where the action is not an atomic concept by reducing it to the most suitable atomic one.
5.6.2 **Synthesis for Italian**

The Italian version of the predicate for synthesis, namely \text{i\_semantics2syntax/2}, is very similar to the English one. The only important changes concern:

- the absence of verbs with prepositions (like "go to") used in questions. Italian uses, in general, the same preposition for introducing the interrogative;
- a particular care for the use of present participles, namely "proveniente" (coming), used in noun phrases but not with verb phrases. This resembles English, in that one would say "the train coming from Parma...", but not "the train ... comes coming from Parma", where with the ellipsis we indicate a possible missing part of the sentence. The distinction between noun phrases and verb phrases is here stressed by introducing two different predicates: \text{i\_take\_cl/4} and \text{i\_take\_cl\_adj/4}, which differ only for the case where a source event constituent is present;
- the need of a tiny English-Italian dictionary, for concepts are conventionally expressed with English words, whereas Italian terms are needed here. In particular, the ground predicate \text{eng2ita/2} is given along with 11 facts covering the mapping \text{(train, airplane, hour, minute, day, week, bologna, genoa, milan, parma, turin)} \leftrightarrow \text{(treno, volo, ora, minuto, giorno, settimana, bologna, genova, milano, parma, torino)}.

5.7 **Conclusion**

5.7.1 **Last Remark About Implementation**

The predicates presented so far are a collection of tools for translating to and from both Italian and English. Simple predicates, whose arguments are the source text (as input) and the destination text (as output), can be programmed with the following style:

\[
e2i(E\_Text, I\_Text) \leftarrow \text{text}(E\_Text, E\_List), \text{sentence}(E\_Syntax, E\_List, []), \\
\text{syntax2semantics}(E\_Syntax, Semantics), \text{i\_semantics2syntax}(Semantics, I\_Syntax), \text{i\_sentence}(I\_Syntax, I\_List, []), \text{text}(I\_Text, I\_List).
\]

Similar clauses can be written for Italian-English and also Italian-Italian and English-English translations.

The file containing all four predicates (\text{e2i}, \text{i2e}, \text{i2i}, \text{e2e}) is called \text{translator.pl}, which takes also care of consulting all the modules corresponding to the pieces of program described in this chapter.

5.7.2 **Adaptation to ObjectLog2**

In a few cases, the code shown in this chapter needed to be modified, so that the translator could be compiled as an \text{ObjectLog2} program. In particular, in one case (see end of paragraph 5.5.1) there was a sum term with variables, which was used to elegantly formulate concept refinement. Such term was replaced by a simpler term, and the clause it was written into was split in two clauses that made explicit the disjunction expressed by the original term.
In other points of the program there were sums used in is-a clauses for composing a disjunction of constants, like in

\[ \text{isa(Int, truth + schedule + frequency), ...} \]

As explained in section 4.3, this is accepted because we have + only in the outmost level of a ground expression and therefore no rewritings are required.

Particular care was needed when writing the DCG rules, because the OBJECTLOG predicates translating them were supposed to be totally reversible. Therefore, instead of writing a rule for noun phrases like

\[
\text{np(D × N × CL) → det(D), n(N), n_compl_list(CL).}
\]

where the formation of the attribution structure for \(D, N\) and \(CL\) takes place when calling the rules \(\text{det, n}\) and, respectively, \(\text{n_compl_list}\), we had to make such structure explicit for at least two variables out of three. The reason for doing this is that, when using the DCG rules for transforming a syntactic structure into a text, none of the variables are instantiated and therefore the constraint about the number of uninstantiated variables in unification is broken. This rule was therefore rewritten as

\[
\text{np(det(D) × N × ncl(CL)) → det(D), n(N), n_compl_list(CL).}
\]

### 5.7.3 Commentaries and Criticisms

In this chapter, several styles for dealing with concepts in OBJECTLOG are proposed and discussed. In particular, the intensional approach of the scheme concerning the semantics of sentences ends up with a very elegant and compact formulation. On the other hand, in some cases the high level of abstraction typical for this programming language was not exploitable in a convenient way. For instance, when restating a syntactic concept as its semantics, in spite of the restricted and simplified referring model, many details concerning the syntax needed to be examined. This resulted in a number of clauses that could have been written in plain Prolog with approximately the same payoff. Still, peculiarities of OBJECTLOG, like concept refinements, inclusion relationships and concept unification, were used in a sensible way in order to express such clauses in a more readable form.

One problem concerning concept refinement was found in section 5.5.1 when the \(\text{min/3}\) predicate was introduced as a substitute for attribution intersection. Let us represent such an operator - not available in OBJECTLOG - with the symbol \(\hat{\circ}\). The following expression explains with an example what is intuitively wanted:

\[(a(d) + b(d)) \hat{\circ} (b(d) + c(d)) = b(d)\]

The crux operator works, however, in a different way:

\[
(a(d) + b(d)) \times (b(d) + c(d)) = a(d) \times b(d) + a(d) \times c(d) + b(d) \times b(d) + b(d) \times c(d) = b(d)
\]

\[
\text{min/3 results, in this case, in } \bot, \text{ as neither concept is included in the other.}
\]

This operator would have been useful, for instance, for translating the Italian sentence: "da dove parte il treno?", where the interest related to the interrogative is \(\text{source + from_point(nat)}\) and the interest connected to the action (partire) is \(\text{from_point(nat) + destination}\). Neither \(\times\) nor \(\text{min/3}\), if applied to these two concepts, give the desired result.
from_point(nat). Italian sentences with these interrogative and action are therefore out of the scope of the translator.

One of the clear results obtained from the execution of the program is that the process of unifying the complex terms described in this example (and complex terms in general) is highly time-consuming. Therefore, it turns out that the translator is a quite unpractical tool: the time required for the execution of the goal

\[ \text{i2e('il treno proveniente da parma va a milano.', T).} \]

was 5 min 30 sec on a Pentium 200. On the other hand, the 80% of the time was spent in the execution of the DCG rules, which are expressed by particularly general (and reversible) predicates that backtrack many times before the right rule is “fired” with the right arguments. The syntax-to-semantics and semantics-to-syntax interfaces were programmed with ad hoc (and non-reversible) predicates that highly improved the performances with (relatively) short execution times.

Whether writing this program in OBJECTLOG was worth the effort is, therefore, a question without an immediate answer. Problems, difficulties and advantages were, however, pointed out in the course of the chapter, which was our main objective.
Conclusion

In this thesis we showed that OBJECTLOG is an extremely powerful programming language that reveals its full potentiality especially when taxonomies need to be defined. In the past years, an algebraic subset of the language allowing a rich set of relations between concepts, like "isa" (i.e. is a kind of) and "apo" (i.e. is part of), was used to thoroughly describe a medical domain (see e.g. [20]). We think that OBJECTLOG is a particularly suited language for these purposes. Smaller examples of classifications were also given in this document: in the first chapter (in section 1.6), and in the fifth chapter, where schemes for the representation of the semantics and the syntax of sentences were proposed.

The peculiarities of object oriented programming, such as inheritance, are an essential means for dealing with knowledge specification. We are convinced that the algebraic approach of OBJECTLOG is a convenient way for handling complex objects, which, furthermore, allows us avoiding to go into the field of constraint logic programming. Relevant considerations in this respect are found in chapter 3.

On the other hand, when programming with OBJECTLOG, we had, in some circumstances, the impression of carrying an enormous structure for handling ordinary cases. This feeling of heaviness was made concrete especially when writing very detailed portions of program. In other words, a high level programming language such as OBJECTLOG perfectly suits descriptions of models of knowledge, but, perhaps, it is inadequate for defining clauses closer to the machine level than to the level of conceptualization.

Concerning complex objects, we believe that attributes are both the main advantage and the principal source of problems. The benefits of attributes are that they hide the complexity of the binary relational (Peirce) product and provide a way for forming tuples with extreme flexibility. By contrast, particular care has to be taken with the following aspects of attributes:

- the lack of a level for handling attribute names. Variables can range over complex objects, but not over attribute names. It is, therefore, impossible to establish that in the following term
  \[ a(c) \]

  \( a \) is the attribute name and \( c \) is the value, unless we know it already. This means that an equality like
  \[ a(c) = A(C) \]

  cannot be written for retrieving
  \[ A = a, C = c \]

  This kind of meta-level for attributions could have been advantageously used in several circumstances, for instance for defining more conveniently the projection operator of relational algebra.
the \times operator, which behaves as intersection on denotation of concepts, does not have a counterpart on attributions, namely the ◊ operator, as suggested in paragraph 5.7.3.

nested attributes, as they do not have a clear topological interpretation. They were, though, intensely used in the examples shown in this document.

One of the main issues of this project was to identify an algorithm for unification of \textsc{Objectlog} terms fulfilling the algebraic laws (axioms) shown in the first chapter. The realization of such an algorithm has been done for a subset of the language that does not allow sums in the program, but whose complex objects are definable through type definitions. This was sufficient for pointing out that unification is, in general, very complicated and measures of various kinds need to be taken. First of all, it was necessary to make clear what kind of results were desired as instantiations of variables after unification, since there is, in general, more than one unifier complying with the algebraic axioms. For this purpose, we had to define ourselves an idea of "most general", which regarded the following aspects:

- more general (lattice-wise) solutions for outer variables have higher priority than for inner variables;
- when including type definitions, the most general unifier of interest can be only expressed in terms of the concepts defining the constants involved in unification, with no sums. This is a mechanism for avoiding introduction of negation and set difference, which are facilities not comprised in \textsc{Objectlog};
- uninstantiated variables are considered more general than the top element of the lattice.

Such assumptions do not lend themselves easily to the treatment of the general case:

- when unifying terms, we supposed that one of them was either completely ground or a variable. If both terms could have more than one variable, we would not have just one outmost variable and therefore a higher priority should probably be assigned to one term with respect to the other one;
- if sums in the program are allowed, then it does make sense to return, as solution, instantiations of variables which have sums;
- once we have sums in the program, probably the solution we are looking for is not always the most general one. Let us consider a case like

\[ a + X = b + Y \]

where \( a \) and \( b \) may have some implications. \( X = Y = T \) is a solution, but \{ \( X = b, Y = a \) \} is a solution as well. If we have the following type definitions:

\[ a = c + d \]
\[ b = d + e \]
then \( \{ X = c, Y = c \} \) is probably the kind of solution we expect, which is, lattice-wise, less general than the others, both for \( X \) and for \( Y \).

Implementation-wise, unification was made in an efficient way for the subset OBJECTLOG\(_1\), which did non include type definitions, as it was possible to exploit the order set and maintained on the canonic form of frame terms in a convenient way. On the contrary, in OBJECTLOG\(_2\), unification was based on repeated use of the test for inclusion of concepts, which provoked a major increase of the execution time. The crucial point in the algorithm is the simplification of common parts in the two terms, which largely uses is-a clauses. We believe that the complexity of the algorithm of unification, as it is now, is exponential with the size of the two terms, although we did not give a proof for this. The simplification mentioned above can be implemented more efficiently, for instance by programming an accessory is-a check, not as general as the one available to the OBJECTLOG\(_2\) user, but quicker and sufficient for this purpose.

We think, however, that unification of unrestricted OBJECTLOG terms would probably be unmanageable and, altogether, not necessary for most practical purposes. The most sensible choice is, perhaps, to still keeping some constraints on the terms and providing further built-in predicates that give rise to instantiation of variables. For instance, it could be very useful to have a special is-a predicate for the table look-up described in paragraph 1.5.5, which includes sums in the second argument and whose semantics is clear. Also, it should be possible to have an is-a predicate used for retrieving “instances” of a type, which only returns atomic concepts (against the idea of always returning the most general concept) and backtracks for enumerating all the possible solutions.

Another improvement in the efficiency of the execution of programs could perhaps be made by individuating cases where the adjustment of terms is not necessary, and therefore removing the clauses for flattening, sorting and simplification of terms, which are highly time-consuming.

When OBJECTLOG was compared to similar proposals, we saw that other paradigms exploit types for defining signatures of functions and checking well-typedness of programs. In the same way, it should be possible to associate types to arguments of OBJECTLOG predicates and allow only terms that are found below those types in the lattice. Furthermore, we could also think of enforcing the pure/ground constraint by adorning predicate names with input/output modes, so that unification can be replaced by matching and the pure/ground constraint can be checked statically.

We conclude this report with the conviction that this thesis has achieved the purposes it was written for and that its results, combined with the theoretical papers it was inspired from, can be taken advantage of for further research on this subject.
Appendix A Source code of the compiler

OBJECTLOG1

/*

ObjectLog 1

no type definitions allowed
no + allowed
only one (recursively) outmost variable allowed in non-ground terms
pure/ground constraint

version augmented with partial relaxation of the pure/ground constraint in unification
and working with isa clauses (with pure/ground constraint)

*/

% we take advantage of some facilities for list operations
:- use_module(library(lists)).
:- use_module(library(terms)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for unification of frame terms %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
unify(X,Y) :- (ground(X), ground(Y)),!,X=Y. % if both ground, they must be equal
unify(X,Y) :- ground(X),!,uni(X,Y).
unify(X,Y) :- ground(Y),!,uni(Y,X).

%%%%% clauses for relaxation of pure/ground constraint:
%% if they are both a variable (even the same) we equal them
unify([X],[Y]) :- var(X),var(Y),!,X=Y.
%% if one of them is a var not appearing in the other
%% then we just use prolog unification
unify([X],Y) :- var(X),not_in(X,Y),!,X=Y.
unify(X,[Y]) :- var(Y),not_in(Y,X),!,Y=X.
%% variable appearing in the other but without occur check
unify([X],Y) :- var(X),subsumes_chk([X|_],Y),Y=[_|T],not_in(X,T),!,X=T.
unify(X,[Y]) :- var(Y),subsumes_chk([Y|_],X),X=[_|T],not_in(Y,T),!,Y=T.
%% occur check
unify([X],Y) :- var(X),remove_outmost_variable(Y,T),in(X,T),!,X=[bottom].
unify(X,[Y]) :- var(Y),remove_outmost_variable(X,T),in(Y,T),!,Y=[bottom].

union(Y,[X]) :- write('*** Pure/ground constraint violated for '),
print(X), write(' and '),print(Y),nl,fail.

%% uni(ground term, non-ground term)
%% bottom: a variable must be equalled to bottom
uni([bottom],F) :- member(X,F), var(X), X=[bottom].
uni([bottom],F) :- member(X,F), compound(X), X=..[_|F2], unify([bottom],F2).
uni([bottom],_) :- !,fail.

%% the terms are different and there is a variable outside
uni(F1,F2) :-
separate(F1,_,C1,A1), separate(F2,V2,C2,A2),
V2=[Var],
compare_const(C1,C2,C12),
compare_attr(A1,A2,A12),
append(C12,A12,Var).

%% the terms are different and there is a variable inside
uni(F1,F2) :-
% we must have the same constants!
separate(F1,_,C1,A1), separate(F2,V2,C2,A2),
V2=[|],C1=C2,
compare_attr(A1,A2,[]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for set operations in unification %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% separate(+Frame,-Variables,-Constants,-Attributions)
separate([],[],[],[]).
separate([V|T],[],[V|TV],A) :- var(V),!,separate(T,TV,A).
separate([C|T],[],[C|TC],A) :- simple(C),!,separate(T,[],[C|A]).
separate([A|T],[],[],[A|T]) :- compound(A),!.

%% separate(+Frame,-Variables,-Constants,-Attributions)
separate([],[],[],[]).
separate([V|T],[],[V|TV],A) :- var(V),!,separate(T,TV,A).
separate([C|T],[],[C|TC],A) :- simple(C),!,separate(T,[],[C|A]).
separate([A|T],[],[],[A|T]) :- compound(A),!.

%% compare_const(C1, C2, C12).
/* C12 is the list of constants in C1 which are not in C2. If there
are any constants in C2 which are not in C1, compare_const fails. */
compare_const([],[],[]).
compare_const([H|T],[],CL) :- compare_const(T,CL,CL2),
append([H],CL2,CL).
compare_const([H|T1],[H|T2],CL) :- compare_const([H|T1],T2,CL),
append([H],T2,CL).
compare_attr([A|T1],[A|T2],CL) :- compare_attr(T1,T2,CL).
append([A],CL,CL).

%% compare_attr(A1, A2, A12).
/* A12 is the list of attributions in A1 which are not in A2. If there
are any attributions in A2 which are not in A1, compare_attr fails*/
compare_attr([],[],[]).
compare_attr([H|T],[],CL) :- compare_attr(T,CL,CL2),
append([H],CL2,CL).
compare_attr([A1|T1],[A2|T2],CL) :- %% equal attributes: we unify what is inside
A1=..[N1,F1],A2=..[N2,F2],!
( unify(F1,F2),!,compare_attr(T1,T2,CL) ;
remove(F1,F2,F),ARES=..[N,F],compare_attr(T1,T2,CL0),CL=[ARES|CL0] ).

%% clauses for isa %%%
isa(X,Y) :- (ground(X), ground(Y)),!,isa_ground(X,Y). % if both ground
isa(X,Y) :- ground(X),!,isa_right(X,Y).
isa(X,Y) :- ground(Y),!,isa_left(X,Y).
isa(X,Y) :- write('**** Pure/ground constraint violated for '), print(X), write(' and'), print(Y), !.

%% clauses for isa %%%
isa_ground([bottom],[ ]) :- !. % bottom is less or equal than anything
isa_ground(F1,F2) :- separate(F1,_,C1,A1), separate(F2,_,C2,A2),
compare_attr(C1,C2,A1), % the first has at least all the constant as the second
compare_attr(A1,A2,A1), % and at least all the attributions (and so on recursively)
isa_left(F, [bottom]) :- term_variables(F,V), any_to_bottom(V).
isa_left(_, [bottom]) :- !,fail.

%%%% clauses for isa %%%
isa_left(F, F) :- % the variable is outside in the left frame
separate(F1,V1,C1,A1), separate(F2,_,C2,A2),
\ V1=\ [Var],
isa_compare_const(C2,C1,C21),
isa_compare_attr(A2,A1,A21),
append(C21,A21,V1),
\ V == [] -> true,! ; Var=V .
term_variables(A1,V1), all_to_top(V1). % unbound variables are set to top

%%% the terms are different and there is a variable inside
isa_left(F1, F2) :- % the variable is inside in the left frame
    separate(F1, V1, C1, A1), separate(F2, C2, A2),
    V1=[], compare_const(C1, C2),
    isa_strict_compare_attr(A1, A2), %
    term_variables(A1, VL1), all_to_top(VL1). % unbound variables are set to top

isa_right(F1, F2) :- % there are variables in the right frame
    remove_variables(F2, G2),
    isa_ground(F1, G2),
    term_variables(F2, V2), all_to_top(V2). % unbound variables are set to top
canonic_form(X,Y) :- sort(X,Z), transform(Z,Y).

%% top
transform([],[]).

%% bounds and annihilation
transform(List,[bottom]) :- member(B,List), nonvar(B), B=bottom, ! ;
    member(A,List), compound(A), A=..[_,F],
    canonic_form(F,BOTTOM), BOTTOM==[bottom], !.
    %% annihilation and bounds recursively applied
transform([H|T], [H|TT]) :-
    ( simple(H) ; var(H) ) -> transform(T,TT).

transform([Att],[TAtt]) :-
    %% last attribution in the frame
    compound(Att), !, Att=..[Name,F],
    canonic_form(F,TF), TAtt=..[Name,TF].

transform([Att1, Att2], [TAtt]) :-
    %% two last and equally-named attributions: we merge them
    compound(Att1), Att1=..[Name,F1], Att2=..[Name,F2], !,
    append(F1,F2,FF), canonic_form(FF,TF), TAtt=..[Name,TF].

transform([Att1, Att2|T], TT) :-
    %% two equally-named attributions: we merge them
    compound(Att1), Att1=..[Name,F1], Att2=..[Name,F2], !,
    append(F1,F2,FF), canonic_form(FF,TF),
    TAtt=..[Name,TF], transform([TAtt|T],TT).

%% last case: attribution followed by different attribution
transform([Att|T], [TAtt|TT]) :-
    compound(Att), Att=..[Name,F],
    canonic_form(F,TF), TAtt=..[Name,TF], transform(T,TT).

%%% clauses for the ObjectLog1 compiler %%%
%%% clauses for ObjectLog <-> Prolog term conversion %%%
:op(400, xfy, *).
:op(1100, fx, [?]).
%%% general error handler
translate(X, fail) :- var(X), !, error('Expected clause, found variable'), nl.

%%% for prolog metautilities: we don't translate
translate(Y,X) :- subsumes.chk([?_],Y), Y=[?X], !.

%%% clauses
translate([H :- B], [TH :- Body]) :- !,
    translate_head(H,TH,U_clauses,N_clause),
    translate_formula(B,BT),
    append(U_clauses,BT,UTB),
    append(UTB,N_clause,Body), list2comma(UTB,Body).
%%% goals
translate((:- G),(:- TG)) :- !, translate_formula(G,LG), list2comma(LG,TG).

%%% facts
translate(F, (Head :- Body)) :- !, translate_head(F,Head,U_clauses,N_clause),
append(U_clauses,N_clause,LBody),
list2comma(LBody, Body).

%%% failing translation
translate(X, fail) :- !, error('Cannot compile '), write(X), nl.

%%% error handler for heads of clauses
translate_head(H, _, _, _) :- var(H), !, error('Expected head, found variable'), nl.
translate_head(H, _, _, _) :- compound(H), H=..'[','|_], !,
error('Expected head, found formula: '),
write(H), nl.

%%% 0-arity heads
translate_head(H,H,[],[]) :- simple(H), !.

%%% n-ary heads
translate_head(H,TH,U_clauses,[nonnull(FL)]) :- compound(H), !, H=..'[N|AL],
length(AL,Len), fresh_list(Len, FL), TH=..'[N|FL], generate_u_clauses(AL,FL,U_clauses).

%%% error handler for atoms
translate_atom(A, [fail]) :- var(A), !, error('Expected atom, found variable'), nl.
translate_atom(A, [fail]) :- compound(A), A=..'[','|_], !,
error('Expected atom, found formula: '),
write(A), nl.

%%% 0-arity atoms
translate_atom(X,[X]) :- simple(X), !.

%%% n-arity atoms
translate_atom(A,Clauses) :- compound(A), !, A=..'[N|AL],
length(AL,Len), fresh_list(Len,FL), TA=..'[N|FL],
generate_list(AL,FL,TAL), adjust_clauses(TAL,FL,AC), append(AC,[TA],Clauses).

%%% a formula is either a conjunction (atom,formula) or an atom
translate_formula(F,TF) :- compound(F), F=..'[F1|[F2]], !,
translate_atom(F1,TF1), translate_formula(F2,TF2), append(TF1,TF2,TF).
translate_formula(F,Cl) :- !, translate_atom(F,Cl).

%%% Term T is represented in ObjectLog and brought to canonic form
represent(T,CF) :- crux2list(T,L), canonic_form(L,CF).
represent_list([],[]).
represent_list([H|T],SH * ST) :-
represent(H,SH),
represent_list(T,ST).
crux2list(X,[],SH,SL) :-
(var2(X),!; X='$VAR'(_),!).
list2crux([],SH), SL=SH.
crux2list(X,[LH|T],SH,SL) :-
var2(X),!
list2crux([LH|T],SH,SL).
crux2list(X,Y) :-
crux2list(X,Y).
comma2list(',','(H,CT),..(H,LT)):- comma2list(CT,LT).
comma2list(X,[X]).

error(X) :- write('**** ObjectLog error ****'), nl,
        write(' --> '), write(X).

%%% Generates clauses for unification between formal and actual parameters in heads of clauses
generate_u_clauses([],[],[]).
generate_u_clauses([CA|CA], [FCA|FCA], [FL|FL], [U|U], [TU|TU]) :-
  represent(A,CA), FL=adjust(CA,FCA),
  U=unify(F,FCA), generate_u_clauses(TA, TF, TU, CA, FCA, FL, U, TU).

%%% Generates clauses for dynamic adjustment of terms
adjust_clauses([],[],[]).
adjust_clauses([T|T], [AT|AT], [TAT], [TT], [TC]) :-
  C=adjust(T, AT),
  adjust_clauses(TT, AT, TAT, TC).

%%% Generates a list of N fresh variables
fresh_list(0,[]). fresh_list(N,[N|T]) :- M is N-1, fresh_list(M,T).

%%% dynamic adjustment of terms
adjust(X,Y) :- flatten(X,FX), canonic_form(FX,Y).

%%% Flatten lists against instantiation of variables
flatten([],[]).
flatten([H|T],FHT) :-
  is_list(H), !,
  append(H,T,HT),
  flatten(HT,FHT).

%%% Bottom as failure
nonnull([],[]).
nnonnull([H|T]) :- adjust(H,FH), FH \== [bottom],
  nnonnull(T).

% % % clauses for the ObjectLog interpreter % % %
% ?- represents goals.
?- dynamic objectlog/1.
prolog :- retract(objectlog(_)),
         assert(objectlog(on)).
objectlog :- retract(objectlog(_)),
            assert(objectlog(on)).
?- assert(objectlog(on)).
% Backtranslation of Prolog terms
portray(X) :- objectlog(on),
             (is_list(X)  -> AX=X, list2crux(AV,Y), write(Y)),
             !.
% error handler
term_expansion( ?-(Var), ?-(true)) :- var(Var), !,
        error('Expected goal, found variable'), nl.
% Quoted goals are Prolog goals
term_expansion( ?-(?(Goal)), ?-(Goal)) :- !,
        write('OBJECTLOG: Executing a Prolog goal'), nl.

% Consulting a program
term_expansion( ?-(Consult), ?-(Consult)) :-
        is_list(Consult), !,
        write('OBJECTLOG: Consulting a Prolog program'), nl.
% Compiling a program
term_expansion( ?-(compile(X)), ?-(obj_compile(X))) :- !,
        write('OBJECTLOG: Compiling file '), writeq(X), nl.

% Switch for activation of backtranslation of terms
term_expansion( ?-(prolog), ?-(prolog)) :- !,
        write('OBJECTLOG: Answers are now in Prolog'), nl.

% Goals are treated as definite clause bodies
term_expansion( ?-(Goal), ?-(TGoal)) :-
        translate_formula(Goal,LGoal),
        list2comma(LGoal,TGoal), !,
        write('OBJECTLOG: Internal form is: '), write(TGoal), write(''), nl.

% Switch for activation of backtranslation of terms
term_expansion( ?-(Goal), ?-(Goal)) :- error('expansion failed'), nl,write(Goal), nl.
OBJELECTLOG2

objectlog2.pl

佔洲

ObjectLog 2

type equations allowed (with +)
  + allowed in the program only for ground terms and in the outmost level
  only one (recursively) outmost variable allowed in non-ground terms
  partially relaxed pure/ground constraint

% we take advantage of some facilities for list operations
:- use_module(library(lists)).
:- use_module(library(terms)).
:- [unification].
:- [representation].
:- [interpreter].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for the ObjectLog1 compiler %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% obj_compile(File) :- name(File,FileName),
   append(FileName,".ol",InputName), name(Input,InputName),
   append(FileName,".pl",OutputName), name(Output,OutputName),
   seeing(OldInput),
   see(Input),
   telling(OldOutput),
   tell(Output),
   portray_clause((:- multifile seq/2, p eq/2, a eq/2, leq/2, def/1)),
   repeat,
   read_term(Term,[syntax_errors(dec10)]),
   write_file(Term),
   Term == end_of_file,
   !,
   told,
   tell(OldOutput),
   seen,
   see(OldInput),
   retractall(def_/_),retractall(dep_/_). % deletion of accessory clauses

%%% write_file outputs a term to a file and adds a period
%%% it also takes care not to print the 'end_of_file' term
write_file(end_of_file) :- !.
write_file(X) :- translate(X,TX), portray_clause(TX), nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for ObjectLog <-> Prolog clause translation %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- op(400, xfy, *).
:- op(500, xfy, +).
:- dynamic dep_/2. % type dependencies
:- dynamic def_/1. % true for every defined type

%%% general error handler
translate(X, fail) :- var(X), !, error('Expected clause, found variable').

%%% for prolog metatilities: we don’t translate
translate([X],X) :- !.

%%% type definitions
translate((X = Y),T) :- simple(X), ground([X,Y]), represent(X,RX), represent_sum(Y,RY),
  { sum_type(X,Y) -> T=seq(RX,RY) ;
    product_type(X,Y) -> T=p eq(RX,RY) ;
    attribute_type(RX,RY) -> T=a eq(RX,RY) 
  },
  assert(def_(RX)), portray_clause(def_(RX)),!.
translate((X = Y),_ :- !, error('Incorrect type definition',(X=Y)), fail.
%%% clauses
translate((H :- B), (TH :- Body)) :- translate_head(H, TH, U_clauses, N_clause),
    translate_formula(B, TB), append(U_clauses, TB, UTB), append(UTB, N_clause, LBody),
    list2comma(LBody, Body), !.

%%% Consulting of files
translate((:- C), (:- C)) :- is_list(C), !.

%%% goals
translate((:- G), (:- TG)) :- translate_formula(G, LG), list2comma(LG, TG), !.

%%% DCG rules
translate((H --> B), DCGRule) :- expand_term((H --> B), ExpRule), translate(ExpRule, DCGRule), !.

%%% facts
translate(F, (Head :- Body)) :- translate_head(F, Head, U_clauses, N_clause),
    append(U_clauses, N_clause, LBody),
    list2comma(LBody, Body), !.

%%% failing translation
translate(X, fail) :- !, error('Cannot compile', X).

%%% error handler for heads of clauses
translate_head(H, _, _, _) :- var(H), !, error('Expected head, found variable').
translate_head(H, _, _, _) :- compound(H), H=..[','|_], !, error('Expected head, found formula', H).

%%% 0-arity heads
translate_head(H, H, [], []) :- simple(H), !.

%%% n-ary heads
translate_head(H, TH, U_clauses, [nonnull(FL)]) :- compound(H), !, H=..[N|AL],
    length(AL, Len), fresh_list(Len, FL),
    TH=..[N|FL], generate_u_clauses(AL, FL, U_clauses).

%%% error handler for atoms
translate_atom(A, [fail]) :- var(A), !, error('Expected atom, found variable').
translate_atom(A, [fail]) :- compound(A), A=..[','|_], !, error('Expected atom, found formula', A).

%%% 0-arity atoms
translate_atom(X, [X]) :- simple(X), !.

%%% DCG - terminals
translate_atom('C'(C1, C2, C3), [TA]) :- !, represent_sum(C2, R2), TA=..'C2'(C1, R2, C3).

%%% Unification
translate_atom(=(E1, E2), [TA]) :- !, TA=..[unify, E1, E2],
    represent_sum_list([E1, E2], [F1, F2]).

%%% n-ary atoms
translate_atom(A, Clauses) :- compound(A), !, A=..[N|AL], length(AL, Len), fresh_list(Len, FL),
    TA=..[N|FL],
    represent_sum_list(AL, TAL),
    adjust_clauses(TAL, FL, AC), append(AC, [TA], Clauses).

%%% a formula is either a conjunction (atom, formula) or an atom
translate_formula(F, TF) :- compound(F), F=..[','|[F1|[F2]]], !,
    translate_atom(F1, TF1), translate_formula(F2, TF2), append(TF1, TF2, TF).
translate_formula(F, Cl) :- !, translate_atom(F, Cl).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Utilities for type definitions %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
check_type_consistency(X, Y, LY) :-
    { defined_([X]) -> error('Duplicate type definition', (X=Y)) ;
      direct_recursion([X], LY) -> error('Type definition with direct recursion', (X=Y)) ;
      check_dependencies(X, LY) -> assert(dep_(X, LY)) ;
      error('Type definition with indirect recursion', (X=Y))
    }.

sum_type(X, Y) :- plus2list(Y, LY), simple_list(LY), check_type_consistency(X, Y, LY),
    assert_leq(LY, X).
product_type(X, Y) :- crux2list(Y, LY), simple_list(LY), check_type_consistency(X, Y, LY),
    assert_geq(LY, X).
attribute_type([X], [Y]) :- Y=..[A, V], dif(A, *), dif(A, +),
( defined_(V) -> true ;
  error('Undefined type',V,(X=Y)) ).

simple_list([],).

simple_list([H|T]) :- simple(H), simple_list(T).

%%% type consistency check

direct_recursion(X,[X|_]) :- !.
direct_recursion(X,[_|T]) :- direct_recursion(X,T).

check_dependencies(_, []) :- !.
check_dependencies(X,[H|T]) :- not_depend(H,X), check_dependencies(X,T).

depend_list([H|T],D) :- depend(H,D),! ; depend_list(T,D).
depend(X,D) :- dep_(X,Dep), (member(D,Dep),! ; depend_list(Dep,D)).
not_depend(X,H) :- \+ depend(X,H).

assert_leq([],_). assert_leq([H|T],X) :- portray_clause(leq([H],[X])), assert_leq(T,X).

assert_geq([],_). assert_geq([H|T],X) :- portray_clause(leq([X],[H])), assert_geq(T,X).

eq(X,Y) :- seq(X,Y),! ; peq(X,Y), ! ; aeq(X,Y),!.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Utilities for translation of clauses %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Generates clauses for unification between formal and actual parameters in heads of clauses
generate_u_clauses([],[],[]).
generate_u_clauses([A|TA],[F|TF],[FL|[U|TU]]) :- represent_sum(A,CA), FL=adjust_sum(CA,FCA),
  U=unify(F,FCA), generate_u_clauses(TA,TF,TU).

%% Generates clauses for dynamic adjustment of terms
adjust_clauses([],[],[]).
adjust_clauses([T|TT],[AT|TAT],[C|TC]) :- C=adjust_sum(T,AT), adjust_clauses(TT,TAT,TC).

%% generates a list of N fresh variables
fresh_list(0,[]). fresh_list(N,[_|T]) :- M is N-1, fresh_list(M,T).

%%% Bottom as failure
nonnull([],).
nonnull([H|T]) :- adjust_sum(H,FH), FH \== [bottom], nonnull(T).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% definition of atomicity and built-in concepts %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

atomic_(X) :- ground(X),!, atomic_(X).
atom_([bottom]) :- !. % bottom is atomic

atom_(X) <- simple(X), !, defined(X) -> fail ;

atom_([]) :- basic([]), !, atom_(X). % a constant is atomic if its definition

atom_(X) :- X=..[_,A], !, atom_(A). % a product is atomic if the factors are

non_atomic_ X :- \+ atomic(X).

basic([]):-!.

basic([bottom]):-!.

basic([nat]):-!.

basic([real]):-!.

defined(X) :- basic(X),! ; def(X).
defined_(X) :- basic(X),! ; def_(X).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Other utilities %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transformation of objectlog list to prolog list
olist2list([nil],[]).
olist2list([h(H),t(OT)],[H|T]):- olist2list(OT,T).

% standard error messages
error(X,Y,Z) :- error(X), print(user_error,Y), write(user_error,' in '), print(user_error,Z), nl(user_error).
error(X,Y) :- error(X), write(user_error,' --> '), print(user_error,Y), nl(user_error).
error(X) :- nl(user_error), write(user_error,'*** ObjectLog error: '), write(user_error,X), nl(user_error).

% objectlog representation of texts
text([T],OL) :-
ground(T) -> text2list(T,L), [OL2]=OL, olist2list(OL2,L) ;
ground(OL) -> olist2list(OL,L), list2text(L,T) ;
error('Wrong text or list').

text2list(T,L) :-
t2l([],[],WL,WL) :- t2l([?],CW,CWL,WL).
t2l([63|Spaces],CW,CWL,WL) :- name(W,CW), append(CWL, [W], WL),
delete(Spaces,32,''), !.
t2l([46|Spaces],CW,CWL,WL) :- name(W,CW), append(CWL, [W], WL), append(WL1, [?], WL),
delete(Spaces,32,''), !.
t2l([H|T],CW,CWL,WL) :- dif(H,32),!, append(CW, [H], CW2),
t2l(T,CW2,CWL,WL).
t2l([32|T],CW,CWL,WL) :- !, t2l(T,CW,CWL,WL) ;
t2l([32|T],[],CW,WL) :- !, t2l(T,CW,CWL,WL). % skip trailing spaces

t2l([32|T],[],CW,WL) :- !, name(W,CW), append(CWL, [W], CWL2),
t2l(T,CWL2,WL), !.
t2l([32|T],[],CW,WL) :- !, name(W,CW), append(CWL, [W], CWL2),
t2l(T,CWL2,WL).
list2text(L,T) :- l2t(L,NT), name(T,NT).
l2t([Word],Name) :- !, name(Word,Name).
l2t([H|T],Text) :- name(H,NH), append(NH," ", NHS),
l2t(T,NT), append(NHS,NT,Text).

% predicate for connection of DCG terminals
'C2'(C1,C2,C3) :-
{ (+var(C2) -> [L2] = C2 ; true),
  var(C1) -> olist2list(C3,L3), 'C'(L1,L2,L3), olist2list(C1,L1) ;
  var(C3) -> olist2list(C1,L1), 'C'(L1,L2,L3), olist2list(C3,L3) ;
  olist2list(C1,L1), olist2list(C3,L3), 'C'(L1,L2,L3) },
(var(C2) -> C2 = [L2] ; true).

-%- interpreter.pl

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for the ObjectLog1 interpreter %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ?- represents goals.
:- dynamic objectlog/1.
prolog :- retract(objectlog(_)), assert(objectlog(off)).
objectlog :- retract(objectlog(_)), assert(objectlog(on)).
:- assert(objectlog(on)).
% Backtranslation of Prolog terms
%%% bisogna convertire back considerando anche il +
portray(X) :- objectlog(on), !, (is_list(X) -> (adjust_sum(X,AX),!;AX=X), list2crux(AX,Y), write(Y)).

% error handler
term_expansion( ?-{Var}, ?-(true)) :- var(Var), !, error('Expected goal, found variable').
% Quoted goals are Prolog goals
term_expansion( ?-{Goal}, ?-(Goal)) :- !, write('OBJECTLOG: Executing a Prolog goal'), nl.
% Consulting a program
term_expansion( ?-{Consult}, ?-(Consult)) :- is_list(Consult), !,
write('OBJECTLOG: Consulting a Prolog program'), nl.
% Compiling a program
term_expansion( ?-{compile(X)}, ?-(obj_compile(X))) :- !, write('OBJECTLOG: Compiling file '), writeq(X), nl.
% Switch for activation of backtranslation of terms
term_expansion( ?-(prolog), ?-(prolog)) :- !, write('OBJECTLOG: Answers are now in Prolog'), nl.
term_expansion( ?-(objectlog), ?-(objectlog)) :- !, write('OBJECTLOG: Answers are now in ObjectLog'), nl.

% Goals are treated as definite clause bodies
term_expansion( ?-(Goal), ?-(TGoal)) :- translate_formula(Goal,LGoal), list2comma(LGoal,TGoal), !.
term_expansion( ?-(Goal), ?-(Goal)) :- error('expansion failed',Goal).

representation.pl

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% clauses for canonic representation of frame terms %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sum_canonic_form(X,Y) :-
  subsumes_chk(A+B,X) -> X=A+B, canonic_form(A,RA), sum_canonic_form(B,RB), Y=RA+RB, ! ;
  canonic_form(X,Y).
canonic_form(X,Y) :- sort(X,Z), transform(Z,Y).

%% top
transform([],[[]]).

%% bounds and annihilation
transform(List,[bottom]) :-
  member(B,List), nonvar(B), B=bottom, ! ;
  member(Att,List), compound(Att), Att=..[_,F],
  sum_canonic_form(F,BOTTOM), BOTTOM=[bottom], !.
  % annihilation and bounds recursively applied
  transform([H|T], [H|TT]) :-
    % no transformation on constants or variables
    ( simple(H) ; var(H) ) -> transform(T,TT).
 .
  transform([Att], [TAtt]) :- % last attribution in the frame
    compound(Att), !, Att=..[Name,F],
    sum_canonic_form(F,TT), TAtt=..[Name,TF].
  transform([Att1,Att2], [TAtt]) :- % two last and equally-named attributions: we merge them
    compound(Att1), Att1=..[Name,F1], Att2=..[Name,F2], !,
    append(F1,F2,FF), sum_canonic_form(FF,TT),
    TAtt=..[Name,TF].
  transform([Att1,Att2|T], TT) :- % two equally-named attributions: we merge them
    compound(Att1), Att1=..[Name,F1], Att2=..[Name,F2], !,
    append(F1,F2,FF), sum_canonic_form(FF,TT),
    TAtt=..[Name,TF], transform([TAtt|T],TT).
  % last case: attribution followed by different attribution
  transform([Att|T], [TAtt|TT]) :-
    compound(Att), Att=..[Name,F],
    sum_canonic_form(F,TT), TAtt=..[Name,TF], transform(T,TT).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Conversion and representation of terms %%%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%% ça c'est pour objectlog2
represent_sum(X,Y) :-
  subsumes_chk(A+B,X) -> X=A+B, represent(A,RA), represent_sum(B,RB), Y=RA+RB, ! ;
  represent(X,Y).
adjust_sum(X,Y) :-
  (subsumes_chk(A+B,X) -> X=A+B, adjust(A,RA), adjust_sum(B,RB), Y=RA+RB ;
  adjust(X,Y)), !.

%%% dynamic adjustment of terms
adjust(X,Y) :- flatten(X,FX), canonic_form(FX,Y), simplify(CX,SX), canonic_form(SX,Y).

%%% Flatten lists against instantiation of variables
flatten([],[[]]).
flatten([H|T], FHT) :- is_list(H), !, append(H,T,HT), flatten(HT,FHT).
flatten([H|T], [HH|FT]) :- !, is_list(H), !,
  ( simple(H) -> HH = H, !;
  H=..[N,L], flatten(L,FL), HH=..[N,FL] ),
  !, flatten(T,FT), append(H,TT,HT), flatten(HT,FHT), flatten(FT,FT).
flatten(T,FT).

%%% Term T is represented in ObjectLog and brought to canonic form
represent(T,CF) :- crux2list(T,L), canonic_form(L,CF).

represent_sum_list([],[]).
represent_sum_list([H|T],[H|TT]) :- represent_sum(H,TH),represent_sum_list(T,TT).

% var2 holds if either X is a var or X is a name of a var
var2(X) :- (var(X),!; X='$VAR'(_),! ).

list2crux([H],SH) :- !,
  ((simple(H) ; var2(H)) -> SH = H ; H=..[N,F],list2crux(F,SF),SH=..[N,SF] ).
list2crux([],top):-!.
list2crux([H|T],SH * ST) :-
  list2crux([H],SH),
  list2crux(T,ST).

crux2list(X,[]) :- X=top,!.

crux2list(H,[LH]) :-
  simple(H) -> LH = H ; H=..[N,F],list2crux(F,SF),LH=..[N,SF].
crux2list(X,L) :- subsumes_chk(top * T,X),X=top*T,crux2list(T,L).
crux2list(X,[LH|LT]) :- subsumes_chk(H * T,X),!,X= H*T, simple(H) -> LH = H ;
H=..[N,F],dif(N,*),represent_sum(F,LF), LH=..[N,LF] ),
crux2list(T,LT).

list2plus([H],SH) :- !,
  ( (simple(H) ; var2(H)) -> SH = H ; H=..[N,F],list2plus(F,SF),SH=..[N,SF] ).
list2plus([],top):-!.
list2plus([H|T],SH + ST) :-
  list2plus([H],SH),
  list2plus(T,ST).

plus2list(X,[]) :- X=top,!.

plus2list(H,[LH]) :-
  simple(H) -> LH = H ; H=..[N,F],dif(N,+),plus2list(F,LF), LH=..[N,LF].
plus2list(X,[LH|LT]) :- subsumes_chk(H + T,X),!,X= H+T, simple(H) -> LH = H ;
H=..[N,F],dif(N,+),plus2list(F,LF), LH=..[N,LF] ),
plus2list(T,LT).

%%% Converts a list into a comma structure
list2comma([],( true ) ) :- !.
list2comma([X],X):-!.
list2comma(.(H,LT),',(H,CT)):- list2comma(LT,CT).

%%% Converts a comma structure into a list
comma2list('',(H,CT),.(H,LT)):- comma2list(CT,LT).
comma2list(X,[X]).

simplify([],[]):-!.
simplify(X,Y) :- expand(X,EX),EX=..[H,T],simplify([],H,T,Y).

simp(Prec,Curr,Full,NewPrec) :- append(Prec,Full,Partial),remove_variables(Partial,Part),
  ( var(Curr) -> append(Prec,[Curr],NewPrec) ; isa(Part,[Curr])-> NewPrec=Prec ;
  compound(Curr) -> ( Curr=..[N,F], simplify(F,SF),NewCurr=..[N,SF],
  append(Prec,[NewCurr],NewPrec) ) ;
  disjoint(Part,[Curr]) -> NewPrec=[bottom] ;
  append(Prec,[Curr],NewPrec) ).

simplify(Prec,Curr,[],Simple) :- !,simp(Prec,Curr,[],Simple).
simplify(Prec,Curr,Full,Simple) :- !,simp(Prec,Curr,Full,NewPrec),

disjoint(X,_) :- \ + ground(X),!,false.
disjoint(_,Y) :- \ + ground(Y),!,false.
disjoint([X],[Y]) :- simple(X),simple(Y),atomic_([X]),atomic_{[Y]},!,dif(X,Y).
unification.pl

% if both ground it is isa both ways
unify(X,Y) :- (ground(X), ground(Y)),!, (isa_ground(X,Y) -> isa_ground(Y,X)).

% clauses for relaxation of pure/ground constraint
% % variable appearing in the other but without occur check
unify(X,Y) :- var(X),subsumes_chk([X|_],Y),Y=[_|T],not_in(X,T),!,X=T.

% ordinary unification with types
unify(X,Y) :- ground(X),!, type_uni(X,Y),!.
unify(X,Y) :- ground(Y),!, type_uni(Y,X),!.

% other clauses for relaxation of pure/ground constraint
only_var_atts(N,YY) :-
    separate(N,V,_,A), (ground(A) -> VA = []; only_var_atts(A,VA)),
    append(V,VA,YY).

remove_useless_atts(N,XX,YY) :-
    remove_useless_atts(N,XX,Y,Z), append(Y,Z,YY),
    only_var_atts(N,Y,Z).

% removes invariable parts of a frame
only_var_atts([],[],_A) :- separate([A],V,_,A), (ground(A) -> VA = []; only_var_atts(A,VA)),
append(V,VA,Y).

remove_useless_atts([],[],A) :- !.
remove_useless_atts([X|T],Y,Z) :- !,
    remove_useless_atts([X|T],Y,Z),
    remove_useless_atts([X|T],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).

remove_useless_atts([X],[Y|Z]) :- !,
    remove_useless_atts([X],Y,Z),
    remove_useless_atts([X],Y,Z).
aeq([X],XX), remove_useless_atts(N,XX,Y,Z) ;
X=..[N2,F], !, remove_useless_atts([N2(N)],F,Y,Z) ;
Z=X2).

remove_useless_atts(N, [X|T], Y, Z) :- !,
remove_useless_atts(N, [X], Y, Z1), remove_useless_atts(N, T, Y, Z2), append(Z1, Z2, Z).

make_att([], X, X) :- !.
make_att([H|T], X, Z) :- !, Y=..[H,X], make_att(T, Y, Z).

%% uni(ground term, non-ground term)
%% bottom: a variable must be equalled to bottom
uni(bottom, F) :- member(X, F), var(X), X=..[bottom].
uni(bottom, F) :- member(X, F), compound(X), X=..[_,F2], unify([bottom], F2).
uni(bottom, _) :- !, fail.

uni(F1, F2) :-
separate(F1, _, C1, A1), separate(F2, V2, _, A2),
\n\n\n\nremove_outmost_variable(VF, F) :- VF = [V|F], var(V), !.
\n\n\nremove_outmost_variable(VF, F) :-
separate(F1, _, C1, A1), separate(F2, V2, _, A2),
V2=[],
\n\n\n\nisa_left(F, [bottom]) :- term_variables(F, V), any_to_bottom(V).
isa_left(F, [bottom]) :- !, fail.
isa_left(X, Y) :- error('isa with variables in the left argument', isa(X, Y)), fail.
isa_right(F1,F2) :- % there are variables in the right frame
    remove_sum_variables(F2,G2),
    isa_ground(F1,G2), !,
    term_variables(F2,V2), all_to_top(V2). % unbound variables are set to top

remove_sum_variables(X,Y) :-
    subsumes_chk(A+B,X) -> X=A+B, remove_variables(A,RA),
    remove_sum_variables(B,RB), Y=RA+RB, ! ;
    remove_variables(X,Y).

remove_variables([],[]) :- !.
remove_variables(VF,F) :- separate(VF,_,C,A), remove_var_in_att(A,GA),
    append(C,GA,F).
remove_var_in_att([],[]) :- !.
remove_var_in_att([A|T], [GA|GT]) :-
    A=..[N,F], remove_variables(F,G), GA=..[N,G], remove_var_in_att(T,GT).

all_to_top([]).
all_to_top([[|T]]) :- all_to_top(T).
any_to_bottom([]) :- !, fail.
any_to_bottom([H|T]) :- H=[bottom] ; any_to_bottom(T).

isa_ground(_,[|]) :- !.
isa_ground([bottom],_:_:-).
isa_ground(X,X) :- !.
isa_ground([N],[nat]) :- !, integer(N), N>=0.
isa_ground([R],[real]) :- !, float(R).
isa_ground(X1*X2,Y) :- !, isa_ground(X1,Y), isa_ground(X2,Y).
isa_ground(X,Y1+Y2) :- (isa_ground(X,Y1) ; isa_ground(X,Y2)).
isa_ground(X,[Y|T]) :- dif(T,[]), !, isa_ground(X,[Y]), isa_ground(X,T).
isa_ground([X],[Y]) :- compound(X), compound(Y), !, X=..[N,FX], Y=..[N,FY], isa_ground(FX,FY).

isa_ground(X,Y) :- leql(X,Y).
isa_ground(X,Y) :- eq(X,Z), isa_ground(Z,Y).
isa_ground(X,Y) :- eq(Y,Z), isa_ground(X,Z).
leql(X,X).
leql(X,Y) :- leq(X,Z), leql(Z,Y).

expand_atts(X,Y) :- subsumes_chk(A+B,X), !, X=A+B,
    expand_atts(A,RA), expand_atts(B,RB), Y=RA+RB.
expand_atts(X,Y) :- exp_atts(X,Z), adjust_sum(Z,Y).
exp_atts([],[]) :- !.
exp_atts([X],X2) :- !, (aeq([X],X2), ! ; X2=[X]).
exp_atts([X|T], [X2|T2]) :- !, exp_atts([X], [X2]), exp_atts(T, T2).
Appendix B English-Italian translator

translator.ol

%%% consulting the modules
:- [semantics].
:- [english_syntax].
:- [italian_syntax].
:- [english_dcg].
:- [italian_dcg].
:- [semantics2syntax].
:- [syntax2semantics].
:- [i_semantics2syntax].

%%% English to English
e2e(T1,T2) :-
text(T1,RT1), % from text to text representation (ObjectLog list)
sentence(Syn1,RT1,nil), % to syntax
print('English syntactic representation: '),print(Syn1),nl,
syntax2semantics(Syn1,Sem), % to semantics
print('Semantic representation: '),print(Sem),nl,
semantics2syntax(Sem,Syn2), % back to syntax
print('New English syntactic representation: '),print(Syn2),nl,
sentence(Syn2,RT2,nil), % back to text representation
text(T2,RT2).

%%% Italian to English
i2e(T1,T2) :-
text(T1,RT1), % from text to text representation (ObjectLog list)
i_sentence(Syn1,RT1,nil), % to syntax
print('Italian syntactic representation: '),print(Syn1),nl,
i_syntax2semantics(Syn1,Sem), % to semantics
print('Semantic representation: '),print(Sem),nl,
semantics2syntax(Sem,Syn2), % back to syntax
print('English syntactic representation: '),print(Syn2),nl,
sentence(Syn2,RT2,nil), % back to text representation
text(T2,RT2).

%%% English to Italian
e2i(T1,T2) :-
text(T1,RT1), % from text to text representation (ObjectLog list)
sentence(Syn1,RT1,nil), % to syntax
print('English syntactic representation: '),print(Syn1),nl,
syntax2semantics(Syn1,Sem), % to semantics
print('Semantic representation: '),print(Sem),nl,
i_semantics2syntax(Sem,Syn2), % back to syntax
print('Italian syntactic representation: '),print(Syn2),nl,
i_sentence(Syn2,RT2,nil), % back to text representation
text(T2,RT2).

%%% Italian to Italian
i2i(T1,T2) :-
i_sentence(Syn1,RT1,nil), % to syntax
print('Italian syntactic representation: '),print(Syn1),nl,
i_syntax2semantics(Syn1,Sem), % to semantics
print('Semantic representation: '),print(Sem),nl,
i_semantics2syntax(Sem,Syn2), % back to syntax
print('New Italian syntactic representation: '),print(Syn2),nl,
i_sentence(Syn2,RT2,nil), % back to text representation
text(T2,RT2).

semantics.ol

%%%% Scheme of the semantics

interest = truth + source + destination + schedule + point + frequency + duration.
event = vehicle * source * destination * schedule * point * frequency * duration.

semantics_of_assertion = designated(event).
sogAsked = asked(interest).

123
soq_context = context(event).
semantics_of_question = soq_asked * soq_context.

object = train + airplane.
city = bologna + genoa + milan + parma + turin.
time_hour = hour(nat).
time_minute = minute(nat).
time = time_hour * time_minute.
period = minute + hour + day + week.
vehicle = by(object).
source = from(city).
destination = to(city).
schedule = at(time).
p_from_point = from_point(nat).
p_to_point = to_point(nat).
point = p_from_point + p_to_point.
f_times_per = f_times * f_per.
f_times = times(nat).
f_per = per(period).
f_every = every(time).
frequency = f_times_per + f_every.
duration = for(time).

english_syntax.ol

%%% Syntactic representation of English sentences


verb_dictionary = leave + depart + arrive + go + come + take.
noun_dictionary = train + flight + track + gate + city + time + minute + hour + day.
present_participle_dictionary = coming.
pronoun_dictionary = it + this.
preposition_dictionary = from + to + for + at.
proper_noun_dictionary = milan + turin + genoa + bologna + parma.
determiner_dictionary = a + the + this.
interrogative_adjective_dictionary = which + what.
interrogative_adverb_dictionary = how + when + where.
modifying_adverb_dictionary = often + long.
quantifier_dictionary = every.

sentence = assertion + question.


sentence = assertion + question.
nc_list = nil + nc_ht.
nc_ht = nc_head * nc_tail.
nc_head = h(noun_complement).
nc_tail = t(nc_list).

vc_list = nil + vc_ht.
vc_ht = vc_head * vc_tail.
vc_head = h(verb_complement).
vc_tail = t(vc_list).

noun_complement_list = ncl(nc_list).
verb_complement_list = vcl(vc_list).

plus_np_empty = noun_phrase + empty.
crux_vpp = preposition * plus_np_empty.
verb_prepositional_phrase = pp(crux_vpp).
crux_preposition_np = preposition * noun_phrase.
prepositional_phrase = pp(crux_preposition_np).

verb = v(verb_dictionary).
noun_n = n(noun_dictionary).
noun_plur_n = plur_n(noun_dictionary).
noun = noun_n + noun_plur_n.
pronoun = p(pronoun_dictionary).
preposition = prep(preposition_dictionary).
proper_noun = pn(proper_noun_dictionary).
determiner = det(determiner_dictionary).
interrogative_adjective = int_adj(interrogative_adjective_dictionary).
interrogative_adverb = int_adv(interrogative_adverb_dictionary).
modifying_adverb = mod_adv(modifying_adverb_dictionary).

plus_adjective = present_participle + numeral + quantifier.
adjective = adj(plus_adjective).
sub_clause = sc(verb).
plus_numeral_empty = numeral + empty.
crux_quantified_clause = quantifier * plus_numeral_empty * noun.
quantified_clause = qc(crux_quantified_clause).

present_participle = pres_part(present_participle_dictionary).
numeral = num(nat).
quantifier = quant(quantifier_dictionary).

void = emptiness.
empty = no_structure(void).

italian_syntax.ol

%%% Syntactic representation of Italian sentences

***********************************************
*********** Dictionaries ******
***********************************************
i_verb_dictionary = partire + arrivare + andare + venire + impiegare + essere.
i_noun_dictionary = treno + aereo + volo + binario + uscita + città + giorno + ora + minuto.
i_present_participle_dictionary = proveniente.
i_contracted_preposition_dictionary = da + a + ad.
i_simple_preposition_dictionary = per + dal + 'dall'''+ + al + 'all'''+ + alle.
i_proper_noun_dictionary = milan + turin + genoa + bologna + parma.
i_determiner_dictionary = un + uno + una + 'un''' + il + 'l''' + questo + questa + 'quest'''.
i_interrogative_adjective_dictionary = quale + che.
i_interrogative_adverb_dictionary = quando + dove + 'dov'''+ + quanto.
i_quantifier_dictionary = ogni.
i_sentence = i Assertion + i_question.
i_np_vp = i noun_phrase * i verb_phrase.
i Assertion = a(i[np_vp]).
i_plus_int_empty = i interrogative + empty.
i_pie_np_vp = i_plus_int_empty * i noun_phrase * i verb_phrase.
i Question = q(i_pie_np_vp).

i_determiner * i_noun * i_complement_list.
i_plus_determiner + i_noun + i_complement_list.
i verb_cl = i verb * i_complement_list.
i noun_phrase = np(i_plus_determiner).
i_verb_phrase = vp( i_verb_cl ).
i_complement = i_prepositional_phrase + i_adjective + i_quantified_clause + i_sub_clause.
i_c_list = nil + i_c_ht.
i_c_ht = i_c_head * i_c_tail.
i_c_head = h(i_complement).
i_c_tail = t(i_c_list).
i_complement_list = cl( i_c_list ).
i_prep_np = i_preposition * i_noun_phrase.
i_prepositional_phrase = pp( i_prep_np ).
i_preposition = i_simple_preposition + i_contracted_preposition.
i_present_participle = pres_part( i_present_participle_dictionary ).
i_numerical = num( nat ).
i_quantifier = quant( i_quantifier_dictionary ).
i_verb = v(i_verb_dictionary).
i_noun = n(i_noun_dictionary).
i_noun_plur_n = plur_n(i_noun_dictionary).
i_noun = i_noun_n + i_noun_plur_n.
i_simple_preposition = prep(i_simple_preposition_dictionary).
i_contracted_preposition = prep(i_contracted_preposition_dictionary).
i_proper_noun = pn(i_proper_noun_dictionary).
i_determiner = det(i_determiner_dictionary).
i_interrogative_adjective = int_adj(i_interrogative_adjective_dictionary).
i_interrogative_adverb = int_adv(i_interrogative_adverb_dictionary).
i_int1234 = i_int1 + i_int2 + i_int3 + i_int4.
i_int2t = i_preposition * i_interrogative_adverb.
i_int3t = i_quantifier * i_interrogative_adverb.
i_int4t = i_preposition * i_interrogative_adjective * i_noun.
i_interrogative = int( i_int1234 ).
i_int1 = int1(i_interrogative_adverb).
i_int2 = int2(i_int2t).
i_int3 = int3(i_int3t).
i_int4 = int4(i_int4t).
i_plus_pp_num_quant = i_present_participle + i_numerical + i_quantifier.
i_adjective = adj( i_plus_pp_num_quant ).
i_sub_clause = sc( i_verb ).
i_plus_num_empty = i_numerical + empty.
i_quant_ne_n = i_quantifier * i_plus_num_empty * i_noun.
i_quantified_clause = qc( i_quantified_ne_n ).

english_dcg.ol

sentence(S) --> assertion(S).
sentence(S) --> question(S).
assertion(a(np(NP)*avp(AVP))) --> np(NP), agr_vp(AVP), [ ].
assertion(a(np(NP)*ivp(IPV))) --> np(NP), [ is ], ing_vp(IPV), [ ].
question(q(OI*np(NP)*pvvp(PVP))) --> opt_int(OI), [ does ], np(NP), plain_vp(PVP), [ ? ].
question(q(OI*np(NP)*ivp(IPV))) --> opt_int(OI), [ is ], np(NP), ing_vp(IPV), [ ? ].
question(q(OI*np(NP))) --> opt_int(OI), [ is ], np(NP), [ ? ].
np(det(D)*N*ncl(CL)) --> det(D), n(N), n_compl_list(CL).
np(empty*N*ncl(CL)) --> n(N), n_compl_list(CL).
np(F) --> p(F).
np(FN) --> pn(FN).
plain_vp(v(IPV)*vcl(CL)) --> plain_v(IPV), v_compl_list(CL).
ing_vp(v(IPV)*vcl(CL)) --> ing_v(IPV), v_compl_list(CL).
agr_vp(v(AIV)*vcl(CL)) --> agr_v(AIV), v_compl_list(CL).
int(int_adj(IA)*N)) --> int_adj(IA), n(N).
int(int_adv(IA)) --> int_adv(IA).
int(int_adv(IA)*mod_adv(MA)) --> int_adv(IA), mod_adv(MA).
v_compl_list(nil) --> [].
v_compl_list(h(C)*t(CL)) --> v_compl(C), v_compl_list(CL).
n_compl_list(h(C)*t(CL)) --> n_compl(C), n_compl_list(CL),!.
n_compl_list(nil) --> [].
v_compl(C) --> v_pp(C).
v_compl(C) --> adjective(C).
v_compl(C) --> sub_clause(C).
v_compl(C) --> quantified_clause(C).
n_compl(C) --> pp(C).
n_compl(C) --> adjective(C).
n_compl(C) --> quantified_clause(C).

v_pp(pp(P*np(NP))) --&gt; prep(P), opt_np(NP).
pp(pp(P*np(NP))) --&gt; prep(P), np(NP).
adjective(adj(A)) --&gt; present_participle(A).
adjective(adj(A)) --&gt; numeral(A).
adjective(adj(A)) --&gt; quantifier(A).

sub_clause(sc(V)) --&gt; [to], plain_v(V).
quantified_clause(qc(quant(Q)*num(Num)*N)) --&gt; quantifier(Q), numeral(Num), n(N).
quantified_clause(qc(quant(Q)*N)) --&gt; quantifier(Q), n(N).

%%% terminals
plain_v(Verb) --&gt; [Verb], {v(Verb)}.
ing_v(Verb) --&gt; [verb], {v(Verb), ing_form(Verb,IVerb)}.
agr_v(Verb) --&gt; [AVerb], {v(Verb), agreement(Verb,AVerb)}.
present_participle(pres_part(come)) --&gt; [coming].
numeral(X) --&gt; [X], { is_int(X) }.
quantifier(Q) --&gt; [Q], { quant(Q) }.
n(n(N)) --&gt; [N], { n(N) }.
n(plur_n(SN)) --&gt; [N], { n(SN), agreement(SN,N) }.
p(p(P)) --&gt; [P], {p(P)}.
prep(preP(P)) --&gt; [P], {prep(P)}.
np(np(PN)) --&gt; [PN], {np(PN)}.
det(D) --&gt; [D], {det(D)}.
int_adj(IA) --&gt; [IA], {int_adj(IA)}.
int_adv(IA) --&gt; [IA], {int_adv(IA)}.
mod_adv(MA) --&gt; [MA], {mod_adv(MA)}.

%%% optionals
opt_int(empty) --&gt; [].
opt_int(I) --&gt; int(I).
optional_numeral(empty) --&gt; [].
optional_numeral(N) --&gt; numeral(N).

%%% dictionary
v(leave).
v(depart).
v(arrive).
v(go).
v(come).
v(take).
n(train).
n(flight).
n(track).
n(gate).
n(city).
n(time).
n(minute).
n(hour).
n(day).
p(it).
p(this).
p(from).
p(to).
p(for).
p(at).
n(milan).
n(turin).
n(genoa).
n(bologna).
n(parma).
det(a).
det(the).
det(this).
int_adj(which).
int_adj(what).
int_adv(how).
int_adv(when).
int_adv(where).
mod_adv(often).
mod_adv(long).
quant(every).
ing_form(leave,leaving).
ing_form(depart,departing).
ing_form(arrive,arriving).
ing_form(go,going).
ing_form(come,coming).
ing_form(take,taking).
agreement(leave,leaves).
agreement(depart,departs).
agreement(arrive,arrives).
agreement(go,goes).
agreement(come,comes).
agreement(take,takes).
agreement(train,train).
agreement(flight,flights).
agreement(track,tracks).
agreement(gate,gares).
agreement(city,cities).
agreement(time,times).
agreement(minute,minutes).
agreement(hour,hours).
agreement(day,days).

italian_dcg.ol

i_sentence(S) --> i_assertion(S).
i_sentence(S) --> i_question(S).
i_assertion(a(np(NP)*vp(VP))) --~ i_np(NP), i_vp(VP), [?].
i_question(q(OI*vp(VP)*np(NP))) --~ i_opt_int(OI), i_vp(VP), i_np(NP), [?].
i_question(q(empty*np(NP)*vp(VP))) --~ i_np(NP), i_vp(VP), [?].
i_int(int1(int_adv(IA))) --~ i_int_adv(IA).
i_int(int2(prep(P)*int_adv(IA))) --~ i_prep(P), i_int_adv(IA).
i_int(int3(quant(Q)*int_adv(IA))) --~ i_quantifier(Q), i_int_adv(IA).
i_int(int4(prep(P)*int_adj(IA)*N)) --~ i_prep(P), i_int_adj(IA), i_n(N).
i_np(det(D)*N*cl(CL)) --~ i_det(D), i_n(N), i_compl_list(CL).
i_np(empty*N*cl(CL)) --~ i_n(N), i_compl_list(CL).
i_np(PN) --~ i_pn(PN).
i_vp(IV*cl(CL)) --~ i_v(IV), i_compl_list(CL).
i_compl_list(nil) --~ [].
i_compl_list(h(C)*t(CL)) --~ i_compl(C), i_compl_list(CL).
i_compl(C) --~ i_pp(C).
i_compl(C) --~ i_adjective(C).
i_compl(C) --~ i_sub_clause(C).
i_compl(C) --~ i_quantified_clause(C).
i_pp(pp(prep(P)*np(NP))) --~ i_prep(P), i_np(NP).
i_adjective(adj(A)) --~ i_present_participle(A).
i_adjective(adj(num(A))) --~ i_numeral(A).
i_adjective(adj(quant(A))) --~ i_quantifier(A).
i_sub_clause(sc(V)) --~ [per], i_inf_v(V).
i_quantified_clause(qc(quant(Q)*num(Num)*N)) --~ i_quantifier(Q), i_numeral(Num), i_n(N).
i_quantified_clause(qc(quant(Q)*N)) --~ i_quantifier(Q), i_n(N).
i_prep(P) --~ i_simp_prep(P).
i_prep(P) --~ i_contr_prep(P).

%%% terminals
i_inf_v(v(Verb)) --~ [Verb], {i_inf_v(Verb)}.
i_v(v(Verb)) --~ [Verb], {i_inf_v(Verb), i_agreement(Verb,Verb)}.
i_present_participle(Verb) --~ [Verb], {i_pres_part(Verb)}.
i_numeral(X) --~ [X], {i_numeral(X)}.
i_quantifier(Q) --~ [Q], {i_quant(Q)}.
i_n(N) --~ [N], {i_n(N)}.
i_n_plural(SN) --~ [N], {i_n(SN), i_plural(SN, N)}.
i_pn(PN) --~ [PN], {i_pn(PN)}.
i_det(D) --~ [D], {i_det(D)}.
i_int_adv(IA) --~ [IA], {i_int_adv(IA)}.
i_int_adj(IA) --~ [IA], {i_int_adj(IA)}.
i_simp_prep(P) --> [P], {i_simp_prep(P)}.
i_contr_prep(P) --> [P], {i_contr_prep(P)}.

%%% optionals
i_opt_int(I) --> i_int(I).
i_opt_int(empty) --> [].

%%% dictionary
i_int_adv(quando).
i_int_adv(dove).
i_int_adv('dov''').
i_int_adv(quanto).
i_int_adj(che).
i_simp_prep(da).
i_simp_prep(a).
i_simp_prep(ad).
i_simp_prep(per).
i_contr_prep(dal).
i_contr_prep('dall''').
i_contr_prep('all''').
i_contr_prep(alle).
i_pn(milano).
i_pn(torino).
i_pn(parma).
i_pn(bologna).
i_pn(genova).
i_n(treno).
i_n(aereo).
i_n(volo).
i_n(binario).
i_n(uscita).
i_n(città).
i_n(giorno).
i_n(ora).
i_n(minuto).
i_inf_v(partire).
i_inf_v(arrivare).
i_inf_v(essere).
i_det(un).
i_det(uno).
i_det('un''').
i_det(il).
i_det('l''').
i_det(questo).
i_det(questa).
i_det('quest''').
i_pres_part(proveniente).
i_quant(ogni).
i_agreement(essere,'è').
i_agreement(andare,va).
i_agreement(venire,viene).
i_agreement(partire,parte).
i_agreement(arrivare,arriva).
i_agreement(impiegare,impiega).
i_plural{treno,treni}.
i_plural{aereo,aerei}.
i_plural{volo,voli}.
i_plural{binario,binari}.
i_plural{uscita,uscite}.
i_plural{giorno,giorni}.
i_plural{ora,ore}.
i_plural{minuto,minuti}.
i_plural('città','città').

---
syntax2semantics.ol

%%% Clauses for transforming from English syntax to semantics

syntax2semantics(a(A), designated(event * E)) :-
assertion2designated_event(A,E).
syntax2semantics(q(Q), asked(I) * context(event * C * by(Object))) :-
  question2interest(Q,I,Object),
  question2context(Q,I,C).

assertion2designated_event(np(NP) * ivp(VP), by(O) * CNP * CVP) :-
  np2object(NP,O), np2context(NP,CNP), vp2context_assertion(VP,CVP).

assertion2designated_event(np(NP) * avp(VP), by(O) * CNP * CVP) :-
  np2object(NP,O), np2context(NP,CNP), vp2context_assertion(VP,CVP).

np2object(n(O) * _, O).
min(A,B,B) :- isa(B,A).
min(A,B,A) :- isa(A,B).
min(A,B,bottom).

vp2context_assertion(VP,C1 * C2) :- get_action_assertion(VP,ACTION,ARG),
  refine_from_action_assertion(ARG,ACTION,C1),
  get_quantified_clause(VP,C2).

vp2context_assertion(det(_) * n(N) * ncl(h(pp(prep(PREP)) * np(pn(CITY))))*t(nil)), C) :-
  isa(N, train + flight),
  vp2context_assertion(PREP,CITY,C).

vp2context_assertion(det(_)*n(N) * ncl(h(adj(pres_part(coming)))*)t(h(pp(prep(from) *
np(pn(CITY))))*t(nil))), from(CITY),
  isa(N, train + flight).

vp2context_assertion(n(N) * ncl(h(adj(num(NUM)))*t(nil)), from_point(NUM) * to_point(NUM)) :-
  isa(N, track + gate).

np2context(n(_) * ncl(nil), top).

get_action_assertion(v(Verb) * vcl(h(pp(prep(Prep)) * np(pn(ARG))))*t(_)), Action, ARG) :-
  compose_words(Verb, Prep, Action).

refine_from_action_assertion(N,arrive_to, to_point(N)).
refine_from_action_assertion(C,arrive_from, from(C)).

refine_from_action_assertion(C,leave_for, to(C)).
refine_from_action_assertion(C,leave_from, from(C)). %%% ambiguity arbitrarily
resolved!!!

question2interest(int(Interrogative) * Rest, Interest, object * O) :-
  get_interest_and_object(Interrogative, Int, O),
  refine_interest(Int, Rest, Ref_Int),
  min(Int,Ref_Int,Interest).

question2interest(empty * np(_) * VP, truth, object * O) :-
  isa(VP, plain_verb_phrase + ing_verb_phrase + empty).

%%% Interest contained in the interrogative
/*1*/ get_interest_and_object(Interrogative, point, train) :-
  isa(Interrogative, n(track)).
/*2*/ get_interest_and_object(Interrogative, point, airplane) :-
  isa(Interrogative, n(gate)).
/*3*/ get_interest_and_object(Interrogative, source + destination, _) :-
  isa(Interrogative, n(city)).
/*4*/ get_interest_and_object(int_adj(what) * n(time), schedule, _). 
/*5*/ get_interest_and_object(int_adj(where) * empty, source + destination + point, _).
/*6*/ get_interest_and_object(int_adj(where) * mod_adv(long), duration, _).
/*7*/ get_interest_and_object(int_adj(where) * mod_adv(often), frequency, _).

refine_interest(Int, _, Int) :- atomic_(Int). %%% the interest is already fully
specified
%%% this is used for 1, 2, 3, 6
refine_interest(_, Rest, Ref_Int) :- get_main_action(Rest, Action),
  refine_from_action(Action, Ref_Int).

/*
main_action because there might be more than one:
where is the train coming from Parma going to?
in this case go_to is the main action in that it is linked to the interrogative,
whereas coming_from (and then Parma) are just attributes of "train". This means that we are looking at the verb phrase only */

get_main_action(np(_ * vcl(h(pp(prep(from) * _)))) * empty, be_from).
get_main_action(np(_ * vcl(h(pp(prep(Prep) * _)))) * empty, be_to) :- isa(Prep, to + for).
get_main_action(np(_ * empty, be).
get_main_action(np(_ * avp(VP), Action) :- get_action(VP, Action).
get_main_action(np(_ * ivp(VP), Action) :- get_action(VP, Action).

get_action(v(Verb) * vcl(h(pp(prep(Prep) * empty))*t(nil)), Action) :- compose_words(Verb, Prep, Action).
get_action(v(leave) * vcl(VCL), leave_for) :- isa(VCL, nil + h(quantified_clause)*t(top)).
get_action(v(go) * vcl(VCL), go_to) :- isa(VCL, nil + h(quantified_clause)*t(top)).

compose_words(arrive,to,arrive_to).
compose_words(arrive,from,arrive_from).
compose_words(go,to,go_to).
compose_words(leave,for,leave_for).
compose_words(leave,from,leave_from).
compose_words(depart,from,depart_from).

refine_from_action(arrive_to, to_point(nat)).
refine_from_action(arrive_from, source).
refine_from_action(go_to, destination).
refine_from_action(leave_for, destination).  %%% ambiguity arbitrarily resolved!!!!
refine_from_action(depart_from, from_point(nat)).
refine_from_action(leave_from, source).
refine_from_action(be_from, to_point(nat)).
refine_from_action(be_to, from_point(nat)).
refine_from_action(be, point).

%%%%%%%%%%%%%%%%%%%%% question to context

question2context(_, _ * np(NP) * empty, I, CNP) :-
    isa(I, point),
    np2context(NP, CNP).

question2context(_, _ * np(p(it)) * pvp(v(take)*pp(prep(for)*np(NP))*vcl(h(sc(v(V)))*t(h(pp(prep(PREP)*np(pn(CITY))))*t(nil)))), duration, CNP * C * to(CITY)) :-
    np2context(NP, CNP), compose_words(V, PREP, ACTION),
    refine_from_action(ACTION,C).

question2context(_, _ * np(p(it)) * pvp(v(take)*pp(prep(for)*np(NP))*vcl(h(sc(v(V)))*t(h(pp(prep(PREP)*np(pn(CITY))))*t(nil)))), duration, CNP * C * from(CITY)) :-
    np2context(NP, CNP), compose_words(V, PREP, ACTION),
    refine_from_action(ACTION,C).

question2context(_, _ * np(NP) * ivp(VP), _, CNP * CVP) :-
    np2context(NP, CNP), vp2context(VP, CVP).

question2context(_, _ * np(NP) * avp(VP), _, CNP * CVP) :-
    np2context(NP, CNP), vp2context(VP, CVP).

np2context(det(_)* n(N) * ncl(nil(nil)), top).
np2context(det(_)* n(N) * ncl(h(pp(prep(PREP) * np(pn(CITY))))*t(nil)), C) :-
    isa(N, train + flight),
    prep2context(PREP, CITY, C).
np2context(det(_)* n(N) * ncl(h(adj(pres_part(coming)))*t(h(pp(prep(from) * np(pn(CITY)))))*t(nil))), from(CITY)) :-
    isa(N, train + flight).
np2context(n(N) * ncl(h(adj(num(NUM)))*t(nil)), point * from_point(NUM) * to_point(NUM)) :-
    isa(N, track + gate).
np2context(n(_)* ncl([]), top).

vp2context(VP, C1 * C2) :- get_action(VP, ACTION, ARG),
    refine_from_action(ARG, ACTION, C1),
    get_quantified_clause(VP, C2).
get_quantified_clause(_ * vcl(nil), top).
get_quantified_clause(_ * vcl(h(qc(QC))*t(_)), C2) :- get_from_qc(QC,C2).
get_quantified_clause(_ * vcl(h(_)*t(TAIL)), C2) :- get_quantified_clause(vcl(TAIL), C2).

get_from_qc(quant(_)* empty * n(N), times(1) * period(N)).
get_from_qc(quant(_)* num(1) * n(hour), every(hour(1)*minute(0))).
get_from_qc(quant(_)* num(1) * n(minute), every(hour(0)*minute(1))).
get_from_qc(quant(_)* num(NUM) * plur_n(hour), every(hour(NUM)*minute(0))).
get_from_qc(quant(_)* num(NUM) * plur_n(minute), every(hour(0)*minute(NUM))).

prep2context(for, CITY, to(CITY)).
prep2context(to, CITY, to(CITY)).
prep2context(from, CITY, from(CITY)).

i_syntax2semantics.ol
%%% Clauses for transforming from Italian syntax to semantics
i_syntax2semantics(a(A), designated(event*E)) :-
i_assertion2designated_event(A,E).
i_assertion2designated_event(np(NP) * vp(VP), by(O) * CNP * CVP) :-
i_np2object(NP,O),i_np2context(NP,CNP), i_vp2context_assertion(VP,CVP).
i_np2object(n(IO) _ o) :- eng2ita(O,IO).
i_np2context(det(_)* n(N) * cl(h(pp(prep(PREP) * np(pn(CITY))))*t(nil), C) :-
isa(N, treno + aereo + volo),
i_prep2context(PREP,CITY,C).
i_np2context(det(_)* n(N) * cl(h(adj(pres_part(proveniente)))*t(h(pp(prep(da) * np(pn(CITY)))))*t(nil))), from(CITY)) :-
isa(N, treno + aereo + volo).
i_np2context(det(_)* n(N) * cl(h(adj(num(NUM)))*t(nil)), from_point(NUM) * to_point(NUM)) :-
isa(N, binario + uscita).
i_np2context(det(_)* n(_)* cl(nil), top).
i_vp2context(v(ACTION) * cl(CL), C1 * C2) :-
i_get_quantified_clause(CL,C2).
i_vp2context_assertion(VP,C1 * C2) :- i_get_action(VP,ACTION,ARG),
i_refine_from_action(ARG,ACTION,C1),
i_get_quantified_clause(VP,C2).
i_get_action(v(Action) * cl(h(pp(prep(PREP) * np(pn(IARG))))*t(_)), Action, ARG) :-
eng2ita(ARG, IARG).
i_refine_from_action(N, partire, from_point(N)).
i_refine_from_action(C, partire, to(C)).
i_refine_from_action(N, arrivare, to_point(N)).
i_refine_from_action(C, andare, to(C)).
i_refine_from_action(C, venire, from(C)).
i_refine_from_action(_, essere, point).
i_refine_from_action(N, essere_da, to_point(N)).
i_refine_from_action(N, essere_per, from_point(N)).
i_get_quantified_clause(_ * cl(nil), top).
i_get_quantified_clause(_ * cl(h(qc(QC))*t(_)), C2) :- get_from_qc(QC,C2).
i_get_quantified_clause(_ * cl(h(_)*t(TAIL)), C2) :- get_quantified_clause(vcl(TAIL), C2).
i_get_from_qc(quant(_) * empty * n(N), times(1) * period(N)) :- eng2ita(N,IN).
i_get_from_qc(quant(_) * num(1) * n(hour), every(hour(1)*minute(0))).
i_get_from_qc(quant(_) * num(1) * n(minute), every(hour(0)*minute(1))).
i_get_from_qc(quant(_) * num(NUM) * plur_n(hour), every(hour(NUM)*minute(0))).
i_get_from_qc(quant(_) * num(NUM) * plur_n(minute), every(hour(0)*minute(NUM))).
i_prep2context(per, CITY, to(CITY)).
i_prep2context(a, CITY, to(CITY)).
i_prep2context(da, CITY, from(CITY)).

eng2ita(train, treno).
eng2ita(airplane, volo).
eng2ita(hour, ora).
eng2ita(minute, minuto).
eng2ita(day, giorno).
eng2ita(week, settimana).
eng2ita(bologna, bologna).
eng2ita(genoa, genova).
eng2ita(milan, Milano).
eng2ita(parma, Parma).
eng2ita(turin, Torino).

%%% questions

%%% the interest is already fully specified
i_refine_interest(Int, _, Int) :- atomic_(Int).

%%% this is used for 1, 2, 3, 6
i_refine_interest(_, Rest, Ref_Int) :- i_get_main_action(Rest, Action),
i_refine_from_action(Action, Ref_Int).

i_get_main_action(vp(v(essere)) * np(_, cl(h(adj(pres_part(proveniente))))*t(h(pp(prep(da) * _)))*t(nil))), essere_da).
i_get_main_action(vp(v(essere)) * np(_, cl(h(pp(prep(per) * _)))), essere_per).
i_get_main_action(vp(v(essere)) * np(_), essere).
i_get_main_action(np(_), vp(v(Verb) * _), Verb).

i_syntax2semantics(q(Q), asked(I) * context(C * by(Object))) :-
i_question2interest(Q, I, Object),
i_question2context(Q, I, C).

i_question2interest(int(Interrogative) * Rest, Interest, object * O) :-
i_get_interest_and_object(Interrogative, Int, O),
i_refine_interest(Int, Rest, Ref_Int),
min(Int, Ref_Int, Interest).

i_question2interest(empty * np(_), vp(_), truth, object * O).

%%% Interest contained in the interrogative
/*1*/ i_get_interest_and_object(int1(int_adv(dove)), destination + to_point(nat), _).
/*2*/ i_get_interest_and_object(int2(prep(da) * int_adv(dove)), source + from_point(nat), _).
/*3*/ i_get_interest_and_object(int4(Interrogative), from_point(nat), train) :-
isa(Interrogative, prep(a) * n(binario)).
/*4*/ i_get_interest_and_object(int4(Interrogative), from_point(nat), airplane):-
isa(Interrogative, prep(a) * n(uscita)).
/*5*/ i_get_interest_and_object(int4(Interrogative), to_point(nat), train) :-
isa(Interrogative, prep(da) * n(binario)).
/*6*/ i_get_interest_and_object(int4(Interrogative), to_point(nat), airplane) :-
isa(Interrogative, prep(da) * n(uscita)).
/*7*/ i_get_interest_and_object(int4(Interrogative), destination, _) :-
isa(Interrogative, prep(a) * n(città)).
/*8*/ i_get_interest_and_object(int4(Interrogative), source, _) :-
isa(Interrogative, prep(da) * n(città)).
/*9*/ i_get_interest_and_object(int4(Interrogative), schedule, _) :-
isa(Interrogative, prep(a) * n(ora)).
/*10*/ i_get_interest_and_object(int1(int_adv(quando)), schedule, _).
/*11*/ i_get_interest_and_object(int1(int_adv(quanto)), duration, _).
/*12*/ i_get_interest_and_object(int3(quant(ogni) * int_adv(quanto)), frequency, _).

%%% question to context
i_question2context(_, * np(NP) * vp(v(essere))), I, CNP) :-
isa(I, point),
i_np2context(NP, CNP).

i_question2context(_, * np(NP) * vp(VP), _, CNP * CVP) :-
i_np2context(NP, CNP), i_vp2context(VP, CVP).

semantics2syntax.ol

semantics2syntax(designated(Event * by(Object)), a(Sentence)) :-
express_event_assertion(truth, none, Event, Object, Sentence).

semantics2syntax(asked(Interest) * context(Event * by(Object)), q(Interrogative * Sentence)) :-
express_interrogative(Interest, Object, Interrogative, Action),
express_event(Interest, Action, Event, Object, Sentence).

133
express_interrogative(truth,_,empty,none).
express_interrogative(source,_,int(int_adj(which) * n(city)),come_from).
express_interrogative(destination,_,int(int_adj(which) * n(city)),go_to).
express_interrogative(schedule,_,int(int_adj(what) * n(time)),none).
express_interrogative(point,_,int(int_adv(where) * empty),be).
express_interrogative(from_point(nat),train,int(int_adj(which) * n(track)),depart_from).
express_interrogative(from_point(nat),airplane,int(int_adj(which) * n(gate)),depart_from).
express_interrogative(to_point(nat),train,int(int_adj(which) * n(track)),arrive_to).
express_interrogative(to_point(nat),airplane,int(int_adj(which) * n(gate)),arrive_to).
express_interrogative(frequency,_,int(int_adv(what) * mod_adv(often)),none).
express_interrogative(duration,_,int(int_adv(how) * mod_adv(long)),none).
express_event_assertion(Int,none,Event,Object,np(det(the) * n(Object) * ncl(nil)) * avp(v(V) * vcl(VCL))) :-
  isa(Int, truth + schedule + frequency),
  take_vcl_and_action(Event,Object,VCL,A),
  action2verb(A,V,_).
express_event(duration,none,Event,Object,np(p(it)) * pvp(v(take) * vcl(pp(prep(for) * np(det(the) * n(Object) * ncl(nil))) * vcl(sc(go) * vcl(VCL)))) :-
  take_vcl_and_action(Event,Object,VCL,_).
express_event(point,be,Event,Object,np(det(the) * n(Object) * ncl(NCL)) * empty) :-
  take_ncl(Event,Object,NCL,_).
express_event(_,Action,Event,Object,np(det(the) * n(Object) * ncl(NCL)) * pv(p(V) * vcl(VCL))) :-
  take_ncl(Event,Object,NCL,A),
  action2verb(Action * A,V,Prep).
plus_cag = come_from + arrive_to + go_to.
plus_gd = go_to + depart_from.
take_ncl(from(City) * Rest,O,h(pp(prep(from) * np(pn(City))))*t(E),plus_cag * A) :-
  atomic_(from(City)),take_ncl(Rest,O,E,A).
take_ncl(to(City) * Rest,O,h(pp(prep(to) * np(pn(City))))*t(E),plus_gd * A) :-
  atomic_(to(City)),take_ncl(Rest,O,E,A).
take_ncl(_,_,nil,_).
take_vcl_and_action(from(City) * Rest,O,h(pp(prep(from) * np(pn(City))))*t(E),plus_cag * A) :-
  atomic_(from(City)),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(to(City) * Rest,O,h(pp(prep(to) * np(pn(City))))*t(E),plus_gd * A) :-
  atomic_(to(City)),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(from_point(Num)*Rest,O,h(pp(prep(from) * np(empty * n(track) *
  adj(num(Num)))))*t(E),depart_from) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(from_point(Num)*Rest,airplane,h(pp(prep(from) * np(empty * n(gate)
  * adj(num(Num)))))*t(E),depart_from) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(to_point(Num)*Rest,train,h(pp(prep(to) * np(empty * n(track) *
  adj(num(Num)))))*t(E),arrive_to) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(to_point(Num)*Rest,airplane,h(pp(prep(to) * np(empty * n(gate) *
  adj(num(Num)))))*t(E),arrive_to) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(times(1) * per(P) * Rest,O,h(qc(qant(every) * empty * n(P)))*t(E),A) :-
  atomic_(P),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(hour(1) * minute(0)) * Rest,O,h(qc(qant(every) * num(1) *
  n(hour)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(hour(N) * minute(0)) * Rest,O,h(qc(qant(every) * num(N) *
  plur_n(minute)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(hour(0) * minute(1)) * Rest,O,h(qc(qant(every) * num(1) *
  n(minute)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(hour(0) * minute(N)) * Rest,O,h(qc(qant(every) * num(N) *
  plur_n(minute)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(time(1) * minute(0)) * Rest,O,h(qc(qant(every) * num(1) *
  n(time)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(time(N) * minute(0)) * Rest,O,h(qc(qant(every) * num(N) *
  plur_n(time)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(time(0) * minute(N)) * Rest,O,h(qc(qant(every) * num(1) *
  n(minute)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(every(time(0) * minute(N)) * Rest,O,h(qc(qant(every) * num(N) *
  plur_n(minute)))*t(E),A) :-
  atomic_(Num),take_vcl_and_action(Rest,O,E,A).
take_vcl_and_action(_,_,nil,_).
action2verb(come_from,come,from).
action2verb(go_to, go, to).
action2verb(arrive_to, arrive, to).
action2verb(depart_from, depart, from).
action2verb(come_from + arrive_to + go_to, come, from).
action2verb(go_to + depart_from, go, to).

i_syntax2semantics.ol

i_semantics2syntax{asked(Interest) * context(Event * by(Object)), q(Interrogative * Sentence)} :-
    i_express_interrogative(Interest, Object, Interrogative, Action),
    i_express_event(Interest, Action, Event, Object, Sentence).

i_semantics2syntax{designated(Event * by(Object)), a(Sentence)) :-
    i_express_event(truth, none, Event, Object, Sentence).

i_express_interrogative(destination, _, int1(int_adv(dove)), andare).

i_express_interrogative(schedule, _, int4(prep(a) * int_adv(ora)), none).

i_express_interrogative(point, _, int1(int_adv(dove)), essere).

i_express_interrogative(from_point(nat), train, int4(prep(da) * int_adv(quale) * n(binario)), partire).

i_express_interrogative(to_point(nat), train, int4(prep(a) * int_adv(qua)le) * n(binario)), arrivare).

i_express_interrogative(point, essere, Event, Object, np(det(i) * n(TObject) * cl(nil)) * vp(v(essere) * cl(nil))):-
    i_take_cl_adj(Event, Object, CL, A),
    eng2ita(Object, TObject).

i_express_event(_, Action, Event, Object, np(det(i) * n(TObject) * cl(nil)) * vp(v(A) * cl(CL))):
    isa(Int, truth + schedule + frequency),
    i_take_cl_adj(Event, Object, CL, A),
    eng2ita(Object, TObject).

v_a_a = venire + arrivare + andare.

a_p = andare + partire.
i_take_cl(Rest,O,E,\).  
i_take_cl(to_point(Num),airplane,h(pp{prep{'all'''} * np{empty * n{uscita} * 
adj{num{Num}}})\])*t(E),arrivare)  
i_take_cl(Rest,O,E,\).  
i_take_cl(times(1) * per(P) * Rest,O,h(qc{quant{ogni} * empty * n{TP}})*t(E),A)  
i_take_cl(Rest,O,E,A),  
ei2(P,TP).  
i_take_cl(every(hour(N) * minute(0)) * Rest,O,h(qc{quant{ogni} * num{N} * 
plur_n{ora}})*t(E),A)  
i_take_cl(Rest,O,E,A).  
i_take_cl(every(hour(0) * minute(N)) * Rest,O,h(qc{quant{ogni} * num{N} * 
plur_n{minuto}})*t(E),A)  
i_take_cl(Rest,O,E,A).  
i_action2verb(v_a_p,venire).  
i_action2verb(a_p,andare).  
i_action2verb(X,X).
Appendix C Bibliography


