The expressive power of LISA-D

Afstudeerscriptie 314,

ter verkrijging van de graad doctorandus
aan de Katholieke Universiteit Nijmegen,
Faculteit der Wiskunde en Informatica,
Afdeling Informatiesystemen

door:

R. Migchelsen

geboren 09-07-71 te Nijmegen

begeleider(s):

Dr. A.H.M. ter Hofstede

June 24, 1994

Contents

1 Introduction

There are a lot of languages that can be used to manipulate the data in a database. It is not very useful, however to have a database with a lot of data in it and you cannot get proper access to the information in the database. So it is necessary to have a good language with which you can formulate queries that can be asked to the database and that result in the right answers. In other words, it is useful to have a query-language that has enough *expressive power* to formulate a reasonable number of queries with it.

In this thesis, we are going to examine the expressive power of one of those languages. This language is called LISA-D (Language for Information Structure and Access Descriptions). LISA-D has two great advantages that most other query-languages do not have:

- It has a complete formal specification (see [H93]). This makes it possible to prove issues about this language (like the expressive power) in a formal way.
- Queries (and constraints), formulated in LISA-D, look very much like natural language, so it is easier to use than most other languages.

Those were the most important points of interest, when LISA-D was developed.

The expressive power of LISA-D has not been examined thoroughly, yet. There are some theories that can be used to check if the expressive power of query-languages is sufficient to express all the queries in certain classes. A lot of research in this area is done by A.K. Chandra (see [C88]) and by Chandra and Harel (e.g. [CH82]). These articles describe a hierarchy of classes of queries and we used the hierarchy described in [C88] to compare the expressive power of LISA-D with. In [SW91] a method is described to perform this comparison. We will use this method throughout the whole article. The main reason for that is that it is a method that is generally usable (the type of the languages involved is not important) and it has a solid theoretical background.

In figure 1 a graphical representation of the hierarchy is shown. A more detailed description will be given here:

QF: This is the class called *quantifier-free first order logic*. It consists of first order formulas with only $\wedge$ and $\neg$ as operations. We use this class of queries, because we need the translation of the queries in this class to LISA-D, to prove that LISA-D is more expressive than first order logic (with quantifiers).

FO: This class is called *first order logic*. It consists of all first order formulas (with the quantifiers $\exists$ and $\forall$). It is clear that this class has a bigger expressive power than the class of quantifier-free first order logic.

HC: This is the class of the Horn clause programs (or DATALOG-programs). This class consists of all logic programs without negation. A typical query in this class is the *transitive closure*. This query shows us immediately that the Horn clause programs are no subclass of first order logic, because the transitive closure is not expressible as a formula in first order logic (see e.g. [AU79]). First order logic is not a subclass of the Horn clause programs either, because a query like *ForAll* (see [SW91]) is not expressible as a Horn clause program.

SL: This is the class of *stratified logic programs* (see e.g. [K91]). This class consists of logic programs with negation, but in such a way that recursion is not allowed "through" negation. A stratified logic program can be divided into one or more *strata*, such that a negated formula is always defined in a lower stratum. It is clear that this class has at least the same amount of expresive power as the class of Horn clause programs (a Horn clause program can in fact be seen as a stratified logic program with only one stratum). The expressive power is even bigger, because for example the query *NotTC* (see [SW91]), the negation of the transitive closure, cannot be expressed as a Horn clause program. As clauses in a stratified logic program can be seen as formulas in first order logic (with $\wedge$, $\neg$ and $\exists$), the expressive power of this class of queries is also bigger than that of first order logic.

FP: This class contains all *fixed-point queries*. These are all queries that can be formed by augmenting first order logic with a fixed-point operator. In [K91] is proved that this class of queries is bigger than the class of stratified logic programs. An example of a query that is expressible as a fixed-point query, but not as a stratified logic program is the query called *Game*. This query can informally be described as (see e.g. [SW91]): there is a rooted tree G and two players. The leaves of the tree are either black or white. The game starts at the root of the tree. Player 1 starts to move down the tree a step and then player 2 continues. This goes on until a player reaches a leaf. Player 1 wins if this leaf is black and player 2 if it is white. The query will be: *Can player 1 win, independently from the moves by player 2?*

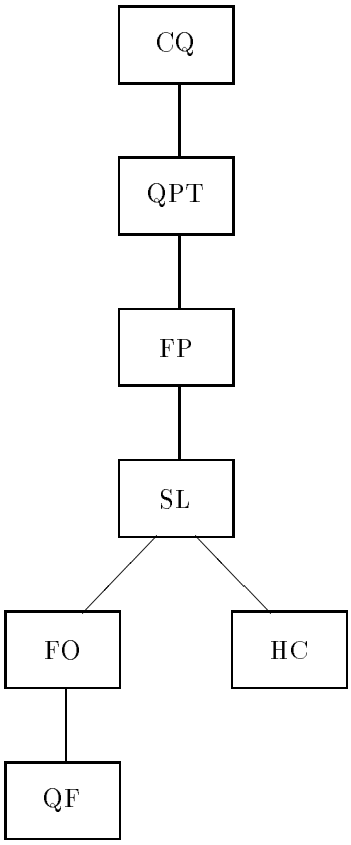

Figure 1: Hierarchy of query-classes

QPT: This class is called QPTIME and consists of all queries that are computable in polynomial time. In [CH82] it is proved that every fixed-point query is computable in polynomial time, but there are queries in QPTIME that cannot be expressed as a fixed-point query. An example of these queries is the query called *Even* (see [SW91]) This query checks if the number of tuples in the population of a relation is even.

CQ: This class contains all computable queries. This is the biggest class, you can wish your query-language to be able to express all queries from. Of course all queries in QPTIME are computable, but there are computable queries that cannot be evaluated in polynomial time. An example is the query that results in all subsets of the population of a given relation. As there are exponential many subsets, this query cannot be evaluated in polynomial time.

We will distinguish two aspects of the language LISA-D, because there are different kinds of

problems when we compare these two aspects of LISA-D with the classes described before. First, we will handle the expressive power of LISA-D seen as a language to formulate constraints in. When you see LISA-D as a language to formulate queries in, you have to deal with the answers of the queries, which consist of n-ary relations and it proves to be difficult to handle these relations. But as will be seen in the remainder of this article, this does not effect the place that LISA-D occupies in the hierarchy of classes.

So we concentrate ourselves in the first section only on LISA-D seen as a constraint-language. To be able to do this, we will give some general definitions in which we define the method we will use throughout the rest of that section to compare the expressive power of LISA-D and the classes of constraint-languages. After that, we will prove (using the defined method) that the constraint-language of LISA-D is more expressive than that of first order logic. To do this we will also present a transformation-process of formulas in first order logic. In the last subsection, we will prove that LISA-D is not at least as expressive as the class of Horn clause programs. We will use an example Horn clause program and show that it cannot be translated to a corresponding LISA-D constraint. We will also show that the notion at least as expressive as does not hold either, when we switch the order of the two constraint-languages (the Horn clause programs are not at least as expressive as LISA-D).

In the next section, we will deal with the expressive power of LISA-D seen as a query-language. This aspect of LISA-D proves to be more expressive than the class of quantifier-free first order logic. The formulas in the class of quantifier-free first order logic will also be transformed first. After we had proved this, we discovered that LISA-D as a query-language was more expressive than first order logic, but the translation process is quite complex. Finally, we will say something about the expressive power of LISA-D compared with the class of Horn clause programs. This will only be mentioned very shortly, because this problem is very much related to that in the last subsection of the section about LISA-D as a constraint-language.

In the next section, we will say something about the expressive power of LISA-D, enriched with the definition of *macros*. This enrichment of LISA-D is described in [P94]. In the first subsection, we will give a definition of these macros. Using this macro-mechanism, a very natural looking translation of stratified logic programs to LISA-D can be made. So in the next subsection, we will prove that LISA-D with macros has a bigger amount of expressive power than the class of stratified logic programs. In the last subsection, we will show that LISA-D with macros is even more expressive than the class of fixed-point queries.

We will end this thesis with some conclusions, that can be drawn after we have made this thesis. We will also mention some points that have to be examined in the future.

2 LISA-D as a constraint-language

In this section we examine the expressive power of LISA-D, as a constraint-language. As we mentioned in the introduction, we will use the method described in [SW91] to do this. In the first subsection, we will describe this method. We have to change some things, because the writers of that article deal with query-languages instead of constraint-languages.

In the next subsection, we will prove that LISA-D is more expressive than first order logic. To do that, we will present a transformation-process of first order formulas. These formulas will be transformed in such a way that they can be translated to LISA-D constraints in a more straightforward way.

In the final subsection, we will show that LISA-D is not at least as expressive as the class of Horn clause programs and also that the class of Horn clause programs is not at least as expressive as LISA-D. The first of these two will be proved by using an example Horn clause program (related to the well-known $a^n b^n$ problem in the theory of regular languages. We will show that this Horn clause program cannot be translated to a LISA-D constraint. The other comparison will be shown, using the constraint NotTC (negation of the transitive closure).

2.1 Basic definitions

In this subsection, we are going to give the definitions, we will use in the rest of this section. The main purpose for this subsection is to define the method, we are going to use to compare the expressive power of LISA-D with that of other languages. Most of these definitions are taken from [SW91], but as we use constraint languages, instead of query languages, we have to make some changes to the original definitions.

First we will define what we mean by a *constraint information system* in general (or CIS for short), followed by a definition of the constraint information system of LISA-D. After that, the notion *at least as expressive as* is defined. Then we can define *equally expressive* (by using the definition of at least as expressive as) and finally these two definitions are combined to give the most important definition, that of the notion *more expressive than*.

2.1.1 Definition of a constraint information system

First, we will define what we mean by a constraint information system:

Definition 2.1 *A constraint information system is represented by the following structure:*

$$CIS := < DBS, CL, \alpha >$$

The three components of this structure have the following definitions:

- *DBS (DataBase System) is the set of all possible instances of the database.*
- *CL (Constraint Language) is the set of all possible constraints.*
- *α is the constraint evaluation function. It is a partial function, mapping $DBS \times CL$ to $BOOL$.*

Using this definition, we can define the constraint information system of LISA-D. Throughout this section, we will use LISA-D as represented by this system:

Definition 2.2 *Let $CIS_{LD} := < DBS_{LD}, CL_{LD}, \alpha_{LD} >$, where the three components of this structure have the following definitions:*

- *DBS_{LD} consists of (at least) all instances $(D, \overline{R})$, represented by a domain D and a tuple of relations with various nonnegative arities $\overline{R} = (R_1, ..., R_m)$, with $R_i \subseteq D^{n_i}$ for all $1 \leq i \leq m$ and all $n_i > 0$. Furthermore, it contains the values of the variables involved in the information descriptor. It will contain some other things, but these are not important in this section.*

- *CL_{LD} consists of all the constraints that can be formed in the following way:*
  - *An information descriptor P can be seen as a constraint.*
  - *If C_1 and C_2 are constraints, then C_1 AND C_2 and C_1 OR C_2 are constraints.*
  - *If C is a constraint, then NO C is a constraint.*
  - *If C is a constraint, x a variable and P an information descriptor, then FOR_EACH x IN P HOLDS C and FOR_SOME x IN P HOLDS C are constraints.*
- *α_{LD} is the evaluation function that determines for every instance of the database and for every constraint an answer. This function is recursively defined as follows, where DB stands for an instance of DBS_{LD}:*

  $\alpha_{LD}(DB, P) = \mu[\mathbf{D}[P](e)](Pop) \neq \emptyset$ *(for a definition of the functions μ and $\mathbf{D}$, see [H93]; e contains the current values of the variables and Pop contains $(D, \overline{R})$).*
  $\alpha_{LD}(DB, C_1\ AND\ C_2) = \alpha_{LD}(DB, C_1) \wedge \alpha_{LD}(DB, C_2)$
  $\alpha_{LD}(DB, C_1\ OR\ C_2) = \alpha_{LD}(DB, C_1) \vee \alpha_{LD}(DB, C_2)$
  $\alpha_{LD}(DB, NO\ C) = \neg\alpha_{LD}(DB, C)$
  $\alpha_{LD}(DB, FOR\_EACH\ x\ IN\ P\ HOLDS\ C) = \forall_{x \in \pi_1 \cdot \mu[\mathbf{D}[P](e)](Pop)} \alpha_{LD}(DB', C)$, *where DB' is equal to DB together with the value of x.*
  $\alpha_{LD}(DB, FOR\_SOME\ x\ IN\ P\ HOLDS\ C) = \alpha_{LD}(DB, NO\ FOR\_EACH\ x\ IN\ P\ HOLDS\ NO\ C)$

2.1.2 Definition of "at least as expressive as"

In this part, we will define when a constraint language CL_2 is *at least as expressive as* a constraint language CL_1. This relationship will depend on:

- the constraint information systems to which the languages belong
- the correspondences between the instances of the database systems

The correspondences between the instances of the databases are given by the function f and between the constraints in the constraint-languages by h.

Definition 2.3 *Let $CIS_1 := < DBS_1, CL_1, \alpha_1 >$ and $CIS_2 := < DBS_2, CL_2, \alpha_2 >$ be two constraint information systems. Let $f : DBS_1 \rightarrow DBS_2$ be a function which determines for every instance $db_1 \in DBS_1$ a corresponding instance $f(db_1) \in DBS_2$. The constraint language CL_2 of CIS_2 is called* **at least as expressive as** *the constraint language CL_1 of CIS_1 with respect to the correspondence given by f iff a function $h : CL_1 \rightarrow CL_2$ exists such that*

$$\forall_{(db_1, c_1) \in dom(\alpha_1)}\ \alpha_1(db_1, c_1) = \alpha_2(f(db_1), h(c_1)).$$

To denote that CL_2 of CIS_2 is at least as expressive as CL_1 of CIS_1 with respect to the correspondence given by f, we will write:

$$CL_1(CIS_1) \leq_f CL_2(CIS_2).$$

2.1.3 Definition of "equally expressive"

In order to define the notion *equally expressive*, an equivalence relation between the instances of the database and between the constraints is necessary. These relations are defined as:

$$db \approx db' \Longleftrightarrow \forall_{c \in CL}\ \alpha(db, c) = \alpha(db', c)$$

$$c \approx c' \Longleftrightarrow \forall_{db \in DBS}\ \alpha(db, c) = \alpha(db, c')$$

Of course both the relation $\approx$ on DBS and the relation $\approx$ on CL are equivalence relations. The equivalence class of db is denoted by $[db]$ and of c by $[c]$. The set of all equivalence classes of instances in DBS is denoted by $DBS^{\approx}$ and of CL by $CL^{\approx}$. It is clear that every evaluation function α determines a function $\alpha^{\approx} : DBS^{\approx} \times CL^{\approx} \rightarrow BOOL$, with

$$\alpha^{\approx}([db],[c]) = \alpha(db,c).$$

Every function $f : DBS_1 \rightarrow DBS_2$ satisfying

$$db_1 \approx db_1' \Longrightarrow f(db_1) \approx f(db_1')$$

induces a function $f^{\approx} : DBS_1^{\approx} \rightarrow DBS_2^{\approx}$ with

$$f^{\approx}([db_1]) = [f(db_1)].$$

At this time the notion *equally expressive* can be defined in a very straightforward way. It will be defined in terms of at least as expressive as.

Definition 2.4 *Let CIS_1 and CIS_2 be constraint information systems, just as in definition 2.3. Let the function f be the same function, too. The constraint languages CL_1 of CIS_1 and CL_2 of CIS_2 are called* **equally expressive** *iff there is a function $f^* : DBS_2 \rightarrow DBS_1$, compatible with f ($\forall_{db_2 \in DBS_2} f(f^*(db_2)) \approx db_2$ and $\forall_{db_1 \in DBS_1} f^*(f(db_1)) \approx db_1$), such that $CL_1(CIS_1) \leq_f CL_2(CIS_2) \wedge CL_2(CIS_2) \leq_{f^*} CL_1(CIS_1)$.*

This is denoted by

$$CL_1(CIS_1) =_f CL_2(CIS_2).$$

Some remarks about this definition:

1. The function f does not have to be bijective, because only a bijective relation on the level of equivalence classes is required. Several equivalent instances of DBS_1 may correspond to a single instance in DBS_2 for example.

2. This definition implies the existence of a function $h : CL_1 \rightarrow CL_2$ (see definition 2.3) and a function $h^* : CL_2 \rightarrow CL_1$, compatible with h. So the function $h^{\approx}$ will be bijective (just as $f^{\approx}$ in the previous remark).

2.1.4 Definition of "more expressive than"

After the previous two definitions, the notion *more expressive than* can be easily defined as:

Definition 2.5 *Let CIS_1, CIS_2 and f be defined as in definition 2.3. CL_2 of CIS_2 is called* **more expressive than** *CL_1 of CIS_1 with respect to f iff $CL_1(CIS_1) \leq_f CL_2(CIS_2) \wedge CL_1(CIS_1) \neq_f CL_2(CIS_2)$.*

This is denoted by

$$CL_1(CIS_1) <_f CL_2(CIS_2).$$

2.2 First order logic

In this subsection we will prove that the expressive power of the constraint language of LISA-D is greater than that of first order logic.

In the first part, we will define the constraint information system of first order logic.

In the next part, we will present some transformations of first order formulas. They will be transformed to a format from which they can be translated to LISA-D more easily.

In the last part, we will prove that the constraint language of LISA-D is *at least as expressive as* the constraint language of first order logic. This will be done by presenting a way to translate transformed first order formulas to corresponding LISA-D constraints. Further we will prove that first order logic and LISA-D are not *equally expressive*. Consequently, LISA-D is *more expressive than* first order logic.

2.2.1 Definition of the constraint information system

The constraint information system of first order logic is defined as:

Definition 2.6 *Let* $CIS_{FO} := < DBS_{FO}, CL_{FO}, \alpha_{FO} >$, *where the three components of this structure have the following definitions:*

- DBS_{FO} *consists of all instances* $(D, \overline{R})$, *represented by a domain* D *and a tuple of relations with various nonnegative arities* $\overline{R} = (R_1, ..., R_m)$, *with* $R_i \subseteq D^{n_i}$ *for all* $1 \leq i \leq m$ *and all* $n_i > 0$.
- CL_{FO} *consists of all first order formulas. These first order formulas are defined as follows:*
  - $R(x_1, ..., x_n)$ *is a first order formula.*
  - *If* φ_1 *and* φ_2 *are first order formulas, then* $\varphi_1 \wedge \varphi_2$ *is a first order formula.*
  - *If* φ *is a first order formula, then* $\neg\varphi$ *and* $\exists_x[\varphi]$ *are first order formulas.*

  Remark: *We do not define* $\varphi_1 \vee \varphi_2$ *and* $\forall_x[\varphi]$, *because we want to use a minimal set of first order logic.*
- α_{FO} *is the evaluation function, which determines for every instance* $(D, \overline{R}) \in DBS_{FO}$ *and every constraint* $c \in CL_{FO}$ *an answer* $\alpha_{FO}((D, \overline{R}), c)$. *The result of this function will be* **true** *iff a mapping exists from all the variables* $(x_1, ..., x_n)$ *of the first order formula to corresponding values* $< c_1, ..., c_n >$, *such that the first order formula itself results in* **true**. *Otherwise the function* α_{FO} *results in* **false**.

  This function is recursively defined as follows:
  - $\alpha_{FO}((D, \overline{R}), R(x_1, ..., x_n)) = < c_1, ..., c_n > \in R \wedge R \in \overline{R}$
  - $\alpha_{FO}((D, \overline{R}), \varphi_1 \wedge \varphi_2) = \alpha_{FO}((D, \overline{R}), \varphi_1) \wedge \alpha_{FO}((D, \overline{R}), \varphi_2)$
  - $\alpha_{FO}((D, \overline{R}), \neg\varphi) = \neg(\alpha_{FO}((D, \overline{R}), \varphi))$
  - $\alpha_{FO}((D, \overline{R}), \exists_x[\varphi]) = \exists_x[\alpha_{FO}((D, \overline{R}), \varphi)]$

2.2.2 Transformations of first order logic

In this part, we will define some transformations that can be applied to formulas in first order logic. These transformations will result in other first order formulas that can be translated to LISA-D in a more straightforward way.

In the remainder of this section, we will only use **closed** first order formulas. These are formulas without any free variables (all variables are bound by some quantifier). First order formulas that are not closed, can simply be made closed, by adding existential quantifiers until all variables are bound. These quantifiers must be added in such a way that they reach the whole formula.

We have to mention here, that the use of constants can be simulated with formulas, satisfying our definition. This can be done by using a unary relation with a population of just one element. This is stated in the following lemma:

Lemma 2.1 $R(c, x_1, ..., x_n)] \equiv \exists_y[R'(y) \wedge R(y, x_1, ..., x_n)]$ *where the population of* R' *consists of* c *only.*

Proof: Suppose $R(c, x_1, ..., x_n)$.

We can assume $R'(c)$, because the population of R' contains c.

Then also $R'(c) \wedge R(c, x_1, ..., x_n)$.

This includes $\exists_y[R'(y) \wedge R(y, x_1, ..., x_n)]$ (with $y = c$).

Suppose $\exists_y[R'(y) \wedge R(y, x_1, ..., x_n)$.

This leads to $R'(c) \wedge R(c, x_1, ..., x_n)$, because c is the only element in the population of R'.

This implies directly $R(c, x_1, ..., x_n)$.

So the lemma holds.

The next step in the transformation-process, will be the splitting of n-ary relations. We will replace every relation $R_i(x_1, ..., x_n)$ by $\exists_{y_i}[R_{i,1}(x_1, y_i) \wedge ... \wedge R_{i,n}(x_n, y_i)]$, where y_i is a new variable and all relations $R_{i,j}$ $(1 \leq j \leq n)$ are new relations. The populations of these relations are defined as follows: a tuple $< c_j, d >$ is in the populations of all relations $R_{i,j}$ iff the tuple $< c_1, ..., c_n >$ is in the population of R_i, and d is a new value. Informally, you can say that the value d determines whether the values c_j formed a tuple in the population of the original relation R_i or not.

We will give an example of this transformation:

Example 2.1 *Consider the relation $R_2(x_1, x_2, x_4)$, with a population consisting of the following tuples: $< a_1, b_1, d_1 >$, $< a_1, b_1, d_2 >$ and $< a_2, b_2, d_2 >$. This relation will be split in the three relations $R_{2,1}(x_1, y)$ (with $< a_1, 1 >$, $< a_1, 2 >$ and $< a_2, 3 >$ in its population), $R_{2,2}(x_2, y)$ (with $< b_1, 1 >$, $< b_1, 2 >$ and $< b_2, 3 >$ in its population) and $R_{2,4}(x_4, y)$ (with $< d_1, 1 >$, $< d_2, 2 >$ and $< d_2, 3 >$ in its population).*

The correctness of the transformation will be proved in the next lemma:

Lemma 2.2 *$R_i(x_1, ..., x_n) \equiv \exists_{y_i}[R_{i,1}(x_1, y_i) \wedge ... \wedge R_{i,n}(x_n, y_i)]$, where all $R_{i,j}$ and y_i are defined as described above.*

Proof: Suppose $R_i(x_1, ..., x_n)$.

Then there is a tuple $< c_1, ..., c_n >$ which is an element of the population of R_i.

If we construct the relations $R_{i,j}$ as described above, then obviously holds (from the definition): $\exists_{y_i}[R_{i,1}(x_1, y_i) \wedge ... \wedge R_{i,n}(x_n, y_i)]$ (for $x_j = c_j$ for all $1 \leq j \leq n$ and $y_i = d$ for some d).

Suppose $\neg(R_i(x_1, ..., x_n))$

Then there is no tuple $< c_1, ..., c_n >$ which is an element of the population of R_i.

If we construct the formulas $R_{i,j}$ as described above, then obviously holds (from the definition): $\neg(\exists_{y_i}[R_{i,1}(x_1, y_i) \wedge ... \wedge R_{i,n}(x_n, y_i)])$ (all relations would have an empty population).

So the lemma holds.

At this moment we have reached the right format to translate these first order formulas to LISA-D constraints. This translation will be described in the next part.

2.2.3 LISA-D compared with first order logic

To prove that LISA-D is more expressive than first order logic, we will first have to prove that LISA-D is at least as expressive as first order logic. This is stated in the next theorem. It will be proved, using structural induction.

We will assume that every relation $R(x, y)$ has a predicator p from the object type of x to R. The function $RNm(p)$ describes for a binary relation $R(x, y)$ a path from x to y.

Theorem 2.1 $CL_{FO}(CIS_{FO}) \leq_f CL_{LD}(CIS_{LD})$

Proof: We define the following functions:

- $f : DBS_{FO} \rightarrow DBS_{LD}$
  $f(R(x, y)) = x\ RNm(p)\ y$.

- $h : CL_{FO} \rightarrow CL_{LD}$
  1. $h(R(x,y)) = x\ RNm(p)\ y$
  2. $h(\varphi_1 \wedge \varphi_2) = h(\varphi_1)\ AND\ h(\varphi_2)$
  3. $h(\neg\varphi) = NO\ h(\varphi)$
  4. $h(\exists_x[\varphi]) = FOR\_SOME\ x\ IN\ X\ HOLDS\ h(\varphi)$, where X is the name of a universal domain, which contains at least all objects that are part of the populations of all object types.

To be able to prove this theorem, the following equation must hold:

$$\forall_{(db,c)\in\alpha_{FO}}\, \alpha_{FO}(db,c) = \alpha_{LD}(f(db), h(c)).$$

To prove that this equation holds, we will use a step from the database of LISA-D (called DB) to the result from the applying of the function f to the database of first order logic (called $(D,\overline{R})$). It is easy to see that the database of first order logic is a subset of that of LISA-D. So the function f will only reach a subset of the database of LISA-D. In this subsection, we do not need more than that subset, though.

- $\alpha_{FO}((D,\overline{R}), R(x,y)) =$ (definition of α_{FO})
  $< c,d > \in R(x,y) \wedge R \in \overline{R} =$ (definition of f)
  $x\ RNm(p)\ y =$ (definition of α_{LD})
  $\alpha_{LD}(DB, x\ RNm(p)\ y) =$ (remark made before)
  $\alpha_{LD}(f((D,\overline{R})), x\ RNm(p)\ y) =$ (definition of h)
  $\alpha_{LD}(f((D,\overline{R})), h(R(x,y)))$
- Suppose $\alpha_{FO}((D,\overline{R}),\varphi_1) = \alpha_{LD}(f((D,\overline{R})), h(\varphi_1))$ holds and the same holds for φ_2. Then we can see:

  $\alpha_{FO}((D,\overline{R}), \varphi_1 \wedge \varphi_2) =$ (definition of α_{FO})
  $\alpha_{FO}((D,\overline{R}),\varphi_1) \wedge \alpha_{FO}((D,\overline{R}),\varphi_2) =$ (induction hypothesis)
  $\alpha_{LD}(f((D,\overline{R})), h(\varphi_1)) \wedge \alpha_{LD}(f((D,\overline{R})), h(\varphi_2)) =$ (definition of α_{LD})
  $\alpha_{LD}(f((D,\overline{R})), h(\varphi_1)\ AND\ h(\varphi_2)) =$ (definition of h)
  $\alpha_{LD}(f((D,\overline{R})), h(\varphi_1 \wedge \varphi_2))$

  $\alpha_{FO}((D,\overline{R}), \neg\varphi_1) =$ (definition of α_{FO})
  $\neg(\alpha_{FO}((D,\overline{R}),\varphi_1)) =$ (induction hypothesis)
  $\neg(\alpha_{LD}(f((D,\overline{R})), h(\varphi_1))) =$ (definition of α_{LD})
  $\alpha_{LD}(f((D,\overline{R})), NO\ h(\varphi_1)) =$ (definition of h)
  $\alpha_{LD}(f((D,\overline{R})), h(\neg\varphi_1))$

  $\alpha_{FO}((D,\overline{R}), \exists_x[\varphi_1]) =$ (definition of α_{FO})
  $\exists_x[\alpha_{FO}((D,\overline{R}),\varphi_1)] =$ (induction hypothesis)
  $\exists_x[\alpha_{LD}(f((D,\overline{R})), h(\varphi_1))] =$ (definition of α_{LD}, using the elementary rule in logic, $\exists_x[\varphi] \equiv \neg(\forall_x[\neg\varphi])$)
  $\alpha_{LD}(f((D,\overline{R})), FOR\_SOME\ x\ IN\ X\ HOLDS\ h(\varphi_1)) =$ (definition of h)
  $\alpha_{LD}(f((D,\overline{R})), h(\exists_x[\varphi_1]))$

  By structural induction, we can say that the equation holds. Due to definition 2.3 the theorem holds.

The next condition, to prove that LISA-D is more expressive than first order logic, is that LISA-D and first order logic must not be equally expressive. This is stated in the following theorem, which is proved by contradiction:

Theorem 2.2 $CL_{FO}(CIS_{FO}) \neq_f CL_{LD}(CIS_{LD})$

Proof: Suppose the opposite holds ($CL_{FO}(CIS_{FO}) =_f CL_{LD}(CIS_{LD})$).

Then $CL_{LD}(CIS_{LD}) \leq_{f^*} CL_{FO}(CIS_{FO})$ (see definition 2.4).

Then a function $h^{\approx} : CL^{\approx}_{FO} \rightarrow CL^{\approx}_{LD}$, as in definition 2.4 exists (remark 2).

This function is bijective, so a function $h^{\approx -1} : CL^{\approx}_{LD} \rightarrow CL^{\approx}_{FO}$ exists. This leads to the existence of a constraint $c \in CL^{\approx}_{FO}$ for which the following equation holds:

$$h^{\approx -1}(ANY\_REPETITION\_OF\ P) = c.$$

But the transitive closure of a given set is not expressible in first order logic (see e.g. [AU79]).

Contradiction!

So the theorem is correct.

After we have proved the previous two theorems, we can prove the main theorem of this section (LISA-D is more expressive than first order logic):

Theorem 2.3 $CL_{FO}(CIS_{FO}) <_f CL_{LD}(CIS_{LD})$

Proof: This follows directly from theorems 2.1 and 2.2 and from definition 2.5.

2.3 Horn clause programs

In this subsection we will compare the expressive power of LISA-D, with the expressive power of the class of Horn clause programs. This class of constraints is also known as the class of DATALOG-constraints.

To be able to do that, we will first give a definition of Horn clause programs. After that, we will define the constraint information system of the class of Horn clause programs.

In the next part, we will show that LISA-D is not at least as expressive as the class of Horn clause programs. We will do this by using an example Horn clause program and showing that that Horn clause program cannot be translated to a corresponding LISA-D constraint. Finally it is also shown that the class of Horn clause programs is not at least as expressive as LISA-D either.

2.3.1 Definition of the constraint information system

In this subsection we will define the constraint information system of Horn clause programs. To be able to define this constraint information system, we have to define what we mean by a Horn clause program first. This definition is taken from [CH85].

Definition 2.7 *A* **constant** *will be defined to be an element of the domain D. Furthermore we assume to have unlimited many* **terminal relation symbols** *R, R_0, R_1, R_2, ... and* **nonterminal relation symbols** *S, S_0, S_1, S_2, ... of various nonnegative arities, and we will have* **variables** *x, y, z, x_1, x_2, ... A* **term** *is defined to be either a constant or a variable. If R and S are n-ary relation symbols and t_1, t_2, ..., t_n are terms, then $R(t_1, ..., t_n)$ and $S(t_1, ..., t_n)$ are* **atomic formulas**. *The atomic formula $S(t_1, ..., t_n)$ is called a* **nonterminal atomic formula** *and $R(t_1, ..., t_n)$ is called a* **terminal atomic formula**.

A **clause** *will be defined as an expression of the form:*

$$A \leftarrow B_1, ..., B_n.$$

where $n \geq 0$, A (the **conclusion***) is a nonterminal atomic formula and B_1, ..., B_n (the* **premises***) are atomic formulas.*

A **Horn clause program** *is a finite nonempty set of clauses in which there occur no constants.*

Two other terms will be used in the next subsections. An **alternative** *of a nonterminal atomic formula is a clause with that nonterminal atomic formula as the conclusion. The* **root** *will be that nonterminal atomic formula, whose result will be seen as the result of the whole Horn clause program. This nonterminal atomic formula will normally be denoted by S_0 or S.*

Remark: In the definition of Horn clause programs in [CH85], $t_1 = t_2$ and $t_1 \neq t_2$ were also defined to be atomic formulas. We have chosen not to define these two formulas, because we do not need them in this subsection.

The constraint information system of the class of Horn clause programs will be defined as:

Definition 2.8 *Let $CIS_{HC} := < DBS_{HC}, CL_{HC}, \alpha_{HC} >$, where the three components of this structure have the following definitions:*

- *DBS_{HC} will be the same as DBS_{FO}, defined in definition 2.6.*
- *CL_{HC} consists of all possible Horn clause programs.*
- *α_{HC} is the evaluation function, which determines for every instance of the database and every constraint an answer (of type BOOL). The result of this function will be* **true** *iff a mapping exists from all the variables $(x_1, ..., x_n)$ in the root of the Horn clause program to corresponding values $< c_1, ..., c_n >$, such that the Horn clause program itself results in* **true**. *Otherwise the function results in* **false**. *This function will be defined as:*
  - $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow .\}) = < c_1, ..., c_n >$, *where c_i is part of the domain of x_i for all $1 \leq i \leq n$.*
  - $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow R(x_{i_1}, ..., x_{i_m}).\}) = < c_1, ..., c_n >$, *where c_{i_j} is part of the population of $R(x_{i_1}, ..., x_{i_m})$ for all $1 \leq j \leq m$ with $1 \leq i_j \leq n$ and c_k is part of the domain of x_k for all $k \neq i_j$ $(1 \leq j \leq m)$.*
  - $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow S'(x_{i_1}, ..., x_{i_m}).\}) = \alpha_{HC}((D, \overline{R}), \{S'(x_{i_1}, ..., x_{i_m}) \leftarrow C_1.,$ $..., S'(x_{i_1}, ..., x_{i_m}) \leftarrow C_k.\}$, *where C_i $(1 \leq i \leq k)$ are all the premises of an alternative of S'. It is not necessary that $S' \neq S$. Recursion is permitted.*
  - $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow B_1, ..., B_m.\}) = \alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow B_1.\}) \wedge ... \wedge$ $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow B_m.\})$
  - $\alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow C_1., ..., S(x_1, ..., x_n) \leftarrow C_m.\}) = \alpha_{HC}((D, \overline{R}), \{S(x_1, ...,$ $x_n) \leftarrow C_1.\}) \vee ... \vee \alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow C_m.\})$
  - $\alpha_{HC}((D, \overline{R}), HCP) = \alpha_{HC}((D, \overline{R}), \{S(x_1, ..., x_n) \leftarrow C_1., ..., S(x_1, ..., x_n) \leftarrow C_m.\})$, *where S is the root of the Horn clause program HCP and $S(x_1, ..., x_n) \leftarrow C_i$. (for all $1 \leq i \leq m$) are all alternatives of $S(x_1, ..., x_n)$.*

  In this definition every term B_i stands for an atomic formula and C_i stands for a sequence of zero or more atomic formulas.

2.3.2 LISA-D compared with Horn clause programs

In this part, we will show that LISA-D is not at least as expressive as the class of Horn clause programs. We will show that there is a Horn clause program that cannot be translated to a corresponding LISA-D constraint. We will use the following Horn clause program for this purpose:

(1) $S(x_1, x_2) \leftarrow R_1(x_1, x_2)$.
(2) $S(x_1, x_2) \leftarrow R_2(x_1, x_3), S(x_3, x_4), R_3(x_4, x_2)$.

This Horn clause program results in **true** iff there is a tuple $< c_1, c_2 >$, that is a part of the population of the following path:

$$R_2^n \circ R_1 \circ R_3^n$$

for some $n \geq 0$.

The proof that this Horn clause program cannot be translated to LISA-D will be given in a few steps:

1. Translation of the part R_2^n would at least require the use of the operation ANY_REPETITION_OF.

2. It is necessary to use more than one seperate calls of ANY_REPETITION_OF in the translation of this Horn clause program.

3. It is not possible to determine afterwards, how many times the inside part of ANY_REPETITION_OF is repeated. This step will be split in two smaller steps:

   (a) This number cannot be packed in the result of the operation.

   (b) It cannot be recovered by the usage of other LISA-D operations.

After that, it will be shown that these steps lead to the wanted result.

Before we are going to prove the validity of these steps, we need the following result. We will use the term *lexicon* to refer to all predefined names in the language LISA-D.

Lemma 2.3 *Every possible translation of the described Horn clause program can be transformed into a translation without the use of the lexicon.*

Proof: All LISA-D keywords (WITH, IS_NAME_OF, INVOLVED_IN, ASSOCIATED_WITH, OF, CONTAINING, IN, PART_OF, COMPRISING, INDICES, AT_POSITION, SEQUENCES, OCCURRING_IN, ELEMENTS and HAVING) are abbreviations of path expressions, so they can be replaced by the corresponding path. The naming functions (ONm, PNm and RNm) are used for a similar purpose. They can also be transformed into calls of the path expression or object type they are the name of.

After this, we are able to start with the first step:

Lemma 2.4 *Translation of the path R_2^n, is not possible without the usage of the LISA-D operation ANY_REPETITION_OF.*

This lemma will be proved by contradiction:

Proof: Suppose it is possible to give a translation without the operation ANY_REPETITION_OF.

The resulting constraint can only consist of a finite number of LISA-D operations. As every available operation only performs one transaction on its parameters, only a finite number of transactions on the paramters of the constraint can be performed.

This means that there is an N, such that for all $n > N$ holds that the tuples $< c_1, c_2 >$ that are part of the population of R_2^n are not calculated. This means that it is possible that the LISA-D constraint results in **false**, if the population of R_2^n is not empty.

Contradiction!

So the lemma holds.

We will prove the following step, next:

Lemma 2.5 *It is necessary to use more than one seperate calls of ANY_REPETITION_OF in the translation of the described Horn clause program.*

Proof: Suppose it would be possible to translate this Horn clause program with only one call of ANY_REPETITION_OF (with only possible further calls of ANY_REPETITION_OF inside the reach of this one call). Then the resulting constraint would be of the form P_1 (ANY_REPETITION_OF P_2) P_3, with P_1, P_2 and P_3 information descriptors. P_1 and P_3 must not contain any call of ANY_REPETITION_OF.

Suppose P_2 does not contain a call of R_2. Then R_2^n has to be calculated without the use of ANY_REPETITION_OF. As we saw in lemma 2.4, this was not possible. The same holds of course for the relation R_3.

So P_2 must contain at least both the relations R_2 and R_3. But then the entire expression has to be calculated within the call of *ANY_REPETITION_OF* P_2 and even in one step,

because P_2^2 would contain a path like $...R_2...R_3...R_2...$ and that is not a part of the result of the described Horn clause program.

But a call of ANY_REPETITION_OF of which the inside is always executed once, can be eliminated. So we will have the same result when we look at the expression $P_1\ P_2\ P_3$. This expression has three possible forms:

1. It contains no calls of ANY_REPETITION_OF anymore.
2. It contains more than one seperate calls of ANY_REPETITION_OF.
3. It is of the same form as in the beginning of this lemma.

The first alternative is impossible, because of lemma 2.4; the second alternative is unnecessary, because of our assumption at the beginning of this proof. So we can assume it has the same form as in the beginning of this lemma.

But at this time, we can repeat the described process. This repetition can be continued until all calls of ANY_REPETITION_OF are removed.

Contradiction!

So the lemma holds.

Finally, we will prove the third step.

Lemma 2.6 *It is not possible to determine afterwards, how many times the inside part of a call of ANY_REPETITION_OF is repeated.*

Proof: Suppose that it is possible to determine this number. This number can be made clear outside the call of ANY_REPETITION_OF by packing the number itself in the result or by recovering it from the result by using other LISA-D operations.

When we want to use the first alternative, we have to pack this number in the result in such a way that it can be used by the other calls of ANY_REPETITION_OF. As the front and the back of the result inside ANY_REPETITION_OF have to be type related, it is not possible to deliver a result in which the desired number is only in the front or the back. There has to be a number in both the front and the back. But also the real result of the call of ANY_REPETITION_OF has to be in both the front and the back.

This type of pairs cannot be formed in LISA-D, because there is no relation consisting of pairs of the type of x_1 or x_2 at the one side and a number at the other side.

So we are not able to pack the number in the result of the call of ANY_REPETITION_OF. Thus we have to use the second alternative: we have to calculate the number by using LISA-D operations on the result of the call of ANY_REPETITION_OF. Of course, we cannot use ANY_REPETITION_OF itself for this purpose, because we do not know at this time, how many times the inside of this operation has to be repeated. But without ANY_REPETITION_OF, we have the same problem as in lemma 2.4. We can only calculate a finite amount of numbers. So this alternative is not usable either.

Contradiction!

So the lemma holds.

After we have proved this three points, we can easily prove that the described Horn clause program cannot be translated to a LISA-D constraint.

Lemma 2.7 *It is not possible to translate the Horn clause program, as described in the beginning of this part, to a LISA-D constraint.*

Proof: Suppose it is possible to translate this Horn clause program to a LISA-D constraint.

This constraint has to contain at least one call of ANY_REPETITION_OF, according to lemma 2.4. According to the next lemma (lemma 2.5), it even must contain at least two seperate calls of ANY_REPETITION_OF.

The expression inside these calls has to be repeated an equal number of times, because otherwise it would be impossible to generate results with equal numbers of R_2 and R_3 only. So it should be possible to determine afterwards how many times an expression inside a call of ANY_REPETITION_OF is repeated. According to lemma 2.6, this is not possible.

Contradiction!

So the lemma holds.

At this time, it is quite easy to prove that LISA-D is not at least as expressive as the class of Horn clause programs.

Theorem 2.4 $CL_{HC}(CIS_{HC}) \leq_f CL_{LD}(CIS_{LD})$ *does* **not** *hold.*

Proof: Suppose the opposite holds ($CL_{HC}(CIS_{HC}) \leq_f CL_{LD}(CIS_{LD})$).

Then a function $h : CL_{HC} \rightarrow CL_{LD}$ should exist, such that $h(hcp) = c$, with $c \in CL_{LD}$ and *hcp* is the Horn clause program described in the beginning of this part.

But as we have seen (lemma 2.7), such a constraint does not exist.

Contradiction !

So the theorem holds.

When we change the order of the two constraint-languages, the notion of at least as expressive as, does not hold either.

Theorem 2.5 $CL_{LD}(CIS_{LD}) \leq_f CL_{HC}(CIS_{HC})$ *does* **not** *hold.*

Proof: Suppose the opposite holds ($CL_{LD}(CIS_{LD}) \leq_f CL_{HC}(CIS_{HC})$).

Then a function $h : CL_{LD} \rightarrow CL_{HC}$ should exist, such that $h(NOT\ ANY\_REPETITION\_OF\ P) = hcp$, with $hcp \in CL_{HC}$ and P some information descriptor.

It is known from the literature (e.g. [SW91]), that this constraint (in that article called **NotTC**) is not expressible as a Horn clause program.

Contradiction !

So the theorem holds.

3 LISA-D as a query-language

In this section, we will focus on the expressive power of LISA-D, as a query-language. To do this, we will have to change the definitions, given in the second section, because we do not deal with constraint-languages anymore. The main difference between the definitions in that section and the definitions here, will be the third component of the information system, the answer system. This system plays an important role in this section.

In the first subsection, we will therefore begin with a definition of an *information system*, followed by a definition of the information system of LISA-D. After that, we will give a new definition of the notions *at least as expressive as*, *equally expressive* and *more expressive than*.

In the next subsection we will prove that the query-language of LISA-D is more expressive than that of the class of *quantifier-free first order logic*, a subclass of first order logic. To do that, we will define some transformations of the formulas in this class.

In the third subsection, we will prove that LISA-D is more expressive than the class of first order logic. We will transform the queries in first order logic to a form that can be translated to LISA-D first. After that, the translation of these queries to LISA-D will be proved using small steps, because this translation is not very straightforward.

In the last subsection, we will say something about the expressive power of LISA-D, related to the class of Horn clause programs. We will only deal with this very shortly, because the results in this subsection, follow almost directly from the final subsection of the previous section (about constraints), because the special constraint-operations of LISA-D could not be used there.

3.1 Basic definitions

In this subsection, we give the definitions, we are going to use in the remainder of this section.

First we will give a general definition of an information system (IS for short), immediately followed by a definition of the information system of LISA-D.

After that the three notions (*at least as expressive as*, *equally expressive* and *more expressive than*) are redefined.

3.1.1 Definition of an information system

First, we will give a definition of an information system. This definition is taken directly from [SW91].

Definition 3.1 *An information system is represented by the following structure:*

$$IS := < DBS, QL, AS, \alpha >$$

The four components of the structure have the following definitions:

- *DBS (DataBase System) is the set of all possible populations of the database.*
- *QL (Query Language) is the set of all possible queries.*
- *AS (Answer System) is the set of all possible answers.*
- *α is the query evaluation function. This is a partial function, mapping $DBS \times QL$ to AS.*

Next, we will define the information system of LISA-D.

Definition 3.2 *$IS_{LD} := < DBS_{LD}, QL_{LD}, AS_{LD}, \alpha_{LD} >$, where the four components have the following definitions:*

- *DBS_{LD} has the same definition as in definition 2.2.*
- *QL_{LD} is defined as follows (see [H93]): a query consists of, either an information descriptor (preceeded by the LISA-D keyword LIST) or the language construction LIST $P_1, ..., P_n|Q$, where all P_i $(1 \leq i \leq n)$ and Q are information descriptors.*

- *AS_{LD} consists of all 2-tuples (with possible n-tuples as first and second parts) that can be formed using the constructions of QL_{LD}.*
- *α_{LD} is defined as:*
  - *$\alpha_{LD}(DB, LIST\ P) = \mu[\mathbf{D}[P](e)](\underline{Pop})$, where e consists of the current values of all variables, Pop is defined to be $(D, \overline{R})$ as in definition 2.2 and $\mathbf{D}$ and μ are defined as in [H93].*
  - *$\alpha_{LD}(DB, LIST\ P_1, ..., P_n|Q) = Nm(\mu[\mathbf{D}[P_1, ..., P_n|Q](e)](Pop))$, where μ, Nm and $\mathbf{D}$ are defined as in [H93] and P_i and Q are information descriptors.*

3.1.2 Definition of "at least as expressive as"

Next, we will define when a query language QL_2 is *at least as expressive as* query language QL_1. This relationship will depend on:

- the information systems to which the languages belong
- the correspondences between the instances of the database systems
- the correspondences between the answers in the answer systems

The correspondences between the instances of the databases are given by the function f, those between the answers by g and between the queries by h.

Definition 3.3 *Let $IS_1 := < DBS_1, QL_1, AS_1, \alpha_1 >$ and $IS_2 := < DBS_2, QL_2, AS_2, \alpha_2 >$ be two information systems. Let $f : DBS_1 \rightarrow DBS_2$ be a function which determines for every instance $db_1 \in DBS_1$ a corresponding instance $f(db_1) \in DBS_2$ and let $g : range(\alpha_1) \rightarrow range(\alpha_2)$ be an injective function which determines for every answer $a_1 \in range(\alpha_1)$ a corresponding answer $g(a_1) \in range(\alpha_2)$. The query language QL_2 of IS_2 is called* **at least as expressive as** *the query language QL_1 of IS_1 with respect to the correspondences given by f and g iff a function $h : CL_1 \rightarrow CL_2$ exists such that*

$$\forall_{(db_1, q_1) \in dom(\alpha_1)}\ g(\alpha_1(db_1, q_1)) = \alpha_2(f(db_1), h(q_1)).$$

To denote that QL_2 of IS_2 is at least as expressive as QL_1 of IS_1 with respect to the correspondences given by f and g, we will write:

$$QL_1(IS_1) \leq_{f,g} QL_2(IS_2)$$

3.1.3 Definition of "equally expressive"

In order to define the notion *equally expressive*, an equivalence relation between the instances of the database and between the queries is necessary. These relations are defined as follows:

$$db \approx db' \Longleftrightarrow \forall_{q \in QL} \alpha(db, q) = \alpha(db', q)$$

$$q \approx q' \Longleftrightarrow \forall_{db \in DBS} \alpha(db, q) = \alpha(db, q')$$

Of course both the relation $\approx$ on DBS and the relation $\approx$ on QL are equivalence relations. The equivalence class of db is denoted by $[db]$ and of q by $[q]$. The set of all equivalence classes of instances in DBS is denoted by $DBS^{\approx}$ and of QL by $QL^{\approx}$. It is clear that every evaluation function α determines a function $\alpha^{\approx} : DBS^{\approx} \times QL^{\approx} \rightarrow AS$, with

$$\alpha^{\approx}([db], [q]) = \alpha(db, q).$$

Every function $f : DBS_1 \rightarrow DBS_2$ satisfying

$$db_1 \approx db_1' \Longrightarrow f(db_1) \approx f(db_1')$$

induces a function $f^{\approx} : DBS_1^{\approx} \rightarrow DBS_2^{\approx}$ with

$$f^{\approx}([db_1]) = [f(db_1)].$$

The notion *equally expressive* will be defined as:

Definition 3.4 *Let IS_1 and IS_2 be two information systems, just as in definition 3.3. Let the functions f and g be the same functions, too. The query languages QL_1 of IS_1 and QL_2 of IS_2 are called* **equally expressive** *iff there is a function $f^* : DBS_2 \rightarrow DBS_1$, compatible with f ($\forall_{db_2 \in DBS_2} f(f^*(db_2)) \approx db_2$ and $\forall_{db_1 \in DBS_1} f^*(f(db_1)) \approx db_1$) and a function g^{-1}, such that $QL_1(IS_1) \leq_{f,g} QL_2(IS_2) \wedge QL_2(IS_2) \leq_{f^*,g^{-1}} QL_1(IS_1)$.*

This is denoted by

$$QL_1(IS_1) =_{f,g} QL_2(IS_2).$$

Some remarks about this definition:

1. The function f does not have to be bijective, because only a bijective relation on the level of equivalence classes is required. Several equivalent instances of DBS_1 may correspond to a single instance in DBS_2 for example.

2. This definition implies the existence of a function $h : QL_1 \rightarrow QL_2$ (see definition 2.3) and a function $h^* : QL_2 \rightarrow QL_1$, compatible with h. So the function $h^{\approx}$ will be bijective (just as $f^{\approx}$ in the previous remark).

3.1.4 Definition of "more expressive than"

At this time the notion of *more expressive than* can be easily defined as:

Definition 3.5 *Let IS_1, IS_2, f and g be defined as in definition 3.3. QL_2 of IS_2 is called* **more expressive than** *QL_1 of IS_1 with respect to f and g iff $QL_1(IS_1) \leq_{f,g} QL_2(IS_2) \wedge QL_1(IS_1) \neq_{f,g} QL_2(IS_2)$.*

This is denoted by

$$QL_1(IS_1) <_{f,g} QL_2(IS_2).$$

3.2 Quantifier-free first order logic

In this subsection, we will prove that LISA-D is more expressive than the class of quantifier-free first order logic. Quantifier-free first order logic is a subset of first order logic, in such a way that formulas in this class contain only $\wedge$ and $\neg$.

In the first part of this subsection, we will define the information system of the class of quantifier-free first order logic. In the next part we will transform the queries in this class to a form that can be translated to LISA-D in a more straightforward way. In the last part of this subsection, we will prove that LISA-D is more expressive than this class of queries by proving the correctness of the given translation and by proving that LISA-D and the quantifier-free first order logic are not equally expressive.

3.2.1 Definition of the information system

The definition of the information system of quantifier-free first order logic will be:

Definition 3.6 *Let $IS_{QF} := < DBS_{QF}, QL_{QF}, AS_{QF}, \alpha_{QF} >$, where the four components of this structure have the following definitions:*

- *DBS_{QF} consists of the same instances as DBS_{FO} in definition 2.6.*
- *QL_{QF} consists of all quantifier-free first order formulas. These are defined as:*

- $R(x_1, ..., x_n)$ *is a first order formula.*
- *If* φ_1 *and* φ_2 *are first order formulas, then* $\varphi_1 \wedge \varphi_2$ *is a first order formula.*
- *If* φ *is a first order formula, then* $\neg\varphi$ *is a first order formula.*

- AS_{QF} *consists of all relations* $R(x_1, ..., x_n)$*, with* $R \in D^n$*, where* D *is a domain, that contains all values of the domains of the variables involved in the relations in* DBS_{QF}*.*

- α_{QF} *is a partial function, which assigns an answer* $a \in AS_{QF}$ *to every instance* $db \in DBS_{QF}$ *and* $q \in QL_{QF}$*. This answer consists of all tuples of type* $(x_1, ..., x_n)$ *for which the formula results in* **true** *and* x_1*, ...,* x_n *are all variables in the formula.*

  This function is recursively defined as follows:

  - $\alpha_{QF}((D, \overline{R}), R(x_1, ..., x_n)) = < c_1, ..., c_n > \in R \wedge R \in \overline{R}$
  - $\alpha_{QF}((D, \overline{R}), \varphi_1 \wedge \varphi_2) = \alpha_{QF}((D, \overline{R}), \varphi_1) \wedge \alpha_{QF}((D, \overline{R}), \varphi_2)$
  - $\alpha_{QF}((D, \overline{R}), \neg\varphi) = \neg(\alpha_{QF}((D, \overline{R}), \varphi))$

3.2.2 Transformations of quantifier-free first order logic

In this part, we will define some transformations that can be applied to formulas in the class of quantifier-free first order logic. These transformations will result in other formulas in quantifier-free first order logic that can be translated to LISA-D in a more straightforward way.

Before we start with the transformations, we have to mention, that the use of constants can be simulated with formulas, satisfying our definition. This can be done by using a unary relation with a population of just one element. This is stated in the following lemma:

Lemma 3.1 $R(c, x_1, ..., x_n) \equiv R'(y) \wedge R(y, x_1, ..., x_n)$ *where the population of* R' *consists of* c *only.*

Proof: Suppose $< c_1, ..., c_n >$ is part of the answer to $R(c, x_1, ..., x_n)$.

Then also $< c_1, ..., c_n >$ will be part of the answer to $R'(c) \wedge R(c, x_1, ..., x_n)$, because the population of R' contains c.

This means directly that $< c, c_1, ..., c_n >$ is part of the answer to $R'(y) \wedge R(y, x_1, ..., x_n)$, because this formula results in **true** for $y = c$. These answers are equivalent, because c is the only value that can be filled in for y.

Suppose $< c, c_1, ..., c_n >$ is part of the answer to $R'(y) \wedge R(y, x_1, ..., x_n)$.

Then of course $< c_1, ..., c_n >$ is part of the answer to $R'(c) \wedge R(c, x_1, ..., x_n)$, because c is part of the population of $R'(y)$.

This means directly that $< c_1, ..., c_n >$ is part of the answer to $R(c, x_1, ..., x_n)$.

So the lemma holds.

We will restrict ourselves to formulas without any variables with empty domains. Such variables can easily be removed. Every relation R, containing such a variable can be replaced by **false** and the formula can be further reduced by applying rules like $\neg$**false** = **true**, $\neg$**true** = **false**, **false** $\wedge \varphi$ = **false**, **true** $\wedge \varphi = \varphi$. By using this process, all occurrences of **true** and **false** can be removed unless the complete formula results in **true** or **false**. In that case, we would have a trivial formula. The formula consisting only of **false** can be simulated by a relation $R(x)$ with an empty population. The formula consisting only of **true** can be simulated by the same relation $R(x)$, but with every value in the universal domain in its population.

We also want to restrict ourselves to formulas with relations that do not contain more than one occurrence of the same variable (like e.g. $R(x, x)$). These relations can be transformed by the introduction of a new relation with only tuples with equal values on all positions in its population, as is stated in the next lemma. In this lemma the notation z does not stand for a real variable-name, but it is a notation for a certain other variable-name.

Lemma 3.2 *$R(z_1, ..., z_p) \equiv R'(x_1, ..., x_n) \wedge R(z'_1, ..., z'_p)$, with n occurences of variable x between the variables z_i $(1 \leq i \leq p)$. All x_k $(1 \leq k \leq n)$ are new variable-names and the population of R' consists of the tuples $< c, ..., c >$ for all c in the domain of x. $z'_i = z_i$ if $z_i \neq x$ and $z'_i = x_k$ if z_i is the k'th occurrence of x.*

Proof: Suppose $< d_1, ..., d_p >$ is part of the answer to $R(z_1, ..., z_p)$.

Then also $< d_1, ..., d_p >$ is part of the answer to $R(z'_1, ..., z'_p)$.

But $< d_1, ..., d_p >$ contains equal values on all positions of the x_k's in $R(z'_1, ..., z'_p)$.

So $< d_1, ..., d_p >$ is part of the answer to $R'(x_1, ..., x_n) \wedge R(z'_1, ..., z'_p)$.

Suppose $< d_1, ..., d_p >$ is part of the answer to $R'(x_1, ..., x_n) \wedge R(z'_1, ..., z'_p)$.

This tuple contains n equal values (because of R'). Variables that always have the same value can be replaced by a single variable.

So $< d_1, ..., d_p >$ is part of the answer to $R'(x, ..., x) \wedge R(z_1, ..., z_p)$.

As the population of R' contains all tuples $< c, ..., c >$ with c in the domain of x, this relation always results in **true**. As $\mathbf{true} \wedge \varphi \equiv \varphi$, we can omit this relation.

So $< d_1, ..., d_p >$ is a part of the answer to $R(z_1, ..., z_p)$.

So the lemma holds.

So in the remainder of this section, if we write $R(x_1, ..., x_n)$, we can assume that all these variables x_i $(1 \leq i \leq n)$ are different.

The final step in our transformation will be an extension of the relations in the first order formula. We define the extension of a relation $R(x_1, ..., x_n)$ to be a relation $R'(x_1, ..., x_n, y_1, ..., y_m)$, where y_1, ..., y_m are all variables that are part of the formula, but not part of R. The way in which these variables are ordered is not important. All relations with the same variables, but with a different ordering of these variables are considered to be equivalent.

The extension of a formula φ is defined as φ', with $R'_i(x_1, ..., x_n, y_1, ..., y_m)$ in φ' if $R_i(x_1, ..., x_n)$ is part of φ, where $x_1, ..., x_n, y_1, ..., y_m$ are all variables in φ. The population of R'_i consists of tuples $< c_1, ..., c_n, d_1, ..., d_m >$, where every element $< c_1, ..., c_n >$ that is an element of the population of R_i is combined with all possible combinations of the values in the domain of the other variables. All these domains were considered not to be empty.

This extension is well defined, because for every relation holds:

Lemma 3.3 *$\varphi' \equiv \varphi$, where φ' is constructed using the procedure described before this lemma.*

Proof: From the construction-procedure follows directly that for all relations $R(x_1, ..., x_n)$ holds that $< c_1, ..., c_n >$ is part of the population of R iff $< c_1, ..., c_n, d_1, ..., d_m >$ is part of the population of $R'(x_1, ..., x_n, y_1, ..., y_m)$. So the restriction to the original values in R remains the same. Because this holds for all relations, it holds for the complete formula as well.

At this moment we have reached the right format to translate these quantifier-free first order formulas to LISA-D queries. This translation will be described in the next subsection.

3.2.3 LISA-D compared with quantifier-free first order logic

To prove that LISA-D is more expressive than the class of quantifier-free first order logic, we have to prove first that LISA-D is at least as expressive as this class. This is stated in the next theorem. It will be proved, using structural induction.

Theorem 3.1 $QL_{QF}(IS_{QF}) \leq_{f,g} QL_{LD}(IS_{LD})$

Proof: We define the following functions:

- $f : DBS_{QF} \rightarrow DBS_{LD}$
  $f(R(x_1, ..., x_n)) = R.$
- $g : range(\alpha_{QF}) \rightarrow range(\alpha_{LD})$
  $g((x_1, ..., x_n)) = ((x_1, ..., x_n), (x_1, ..., x_n))$
- $h : QL_{QF} \rightarrow QL_{LD}$
  1. $h(R(x_1, ..., x_n)) = R$
  2. $h(\varphi_1 \wedge \varphi_2) = h(\varphi_1)\ INTERSECTION\ h(\varphi_2)$
  3. $h(\neg\varphi) =\ NOT\ h(\varphi)$

Remark: In the definition of α_{LD}, every query starts with the keyword $LIST$. We ignore this keyword here, but we mention that it is possible to generate it, by using an auxiliary function $h' : QL_{QF} \rightarrow QL_{LD}$ with $h'(q) = LIST\ h(q)$, with h as defined above.

The following equation must hold:

$$\forall_{(db,q) \in \alpha_{QF}}\, g(\alpha_{QF}(db, q)) = \alpha_{LD}(f(db), h(q)).$$

To prove this equation, we will use the step from the database of LISA-D (called DB) to the result from the applying of the function f to the database of first order logic (called $(D, \overline{R})$). We have mentioned earlier that the database of first order logic is a subset of that of LISA-D. So also the function f will reach a part of the database of LISA-D. In this subsection, we do not need more than that part, though.

- $g(\alpha_{QF}((D, \overline{R}), R(x_1, ..., x_n)))$ = (definition of α_{QF})
  $g(< c_1, ..., c_n > \in R(x_1, ..., x_n))$ = (definition of f and g)
  R = (definition of α_{LD})
  $\alpha_{LD}(DB, R)$ = (remark made before)
  $\alpha_{LD}(f((D, \overline{R})), R)$ = (definition of h)
  $\alpha_{LD}(f((D, \overline{R})), h(R(x_1, ..., x_n)))$
- Suppose $g(\alpha_{QF}((D, \overline{R}), \varphi_1)) = \alpha_{LD}(f((D, \overline{R})), h(\varphi_1))$ holds and the same holds for φ_2. Then we can see:

  $g(\alpha_{QF}((D, \overline{R}), \varphi_1 \wedge \varphi_2))$ = (definition of α_{QF})
  $g(\alpha_{QF}((D, \overline{R}), \varphi_1)) \wedge g(\alpha_{QF}((D, \overline{R}), \varphi_2))$ = (induction hypothesis)
  $\alpha_{LD}(f((D, \overline{R})), h(\varphi_1)) \wedge \alpha_{LD}(f((D, \overline{R})), h(\varphi_2))$ = ($\wedge$ and $\cap$ have the same definitions)
  $\alpha_{LD}(f((D, \overline{R})), h(\varphi_1)) \cap \alpha_{LD}(f((D, \overline{R})), h(\varphi_2))$ = (definition of α_{LD})
  $\alpha_{LD}(f((D, \overline{R})), h(\varphi_1)\ INTERSECTION\ h(\varphi_2))$ = (definition of h)
  $\alpha_{LD}(f((D, \overline{R})), h(\varphi_1 \wedge \varphi_2))$

  $g(\alpha_{QF}((D, \overline{R}), \neg\varphi_1))$ = (definition of α_{QF})
  $\neg(g(\alpha_{QF}((D, \overline{R}), \varphi_1)))$ = (induction hypothesis)
  $\neg(\alpha_{LD}(f((D, \overline{R})), h(\varphi_1)))$ = (definition of α_{LD})
  $\alpha_{LD}(f((D, \overline{R})), NOT\ h(\varphi_1))$ = (definition of h)
  $\alpha_{LD}(f((D, \overline{R})), h(\neg\varphi_1))$

  By structural induction, we can say that the equation holds. Due to definition 3.3 the theorem holds.

The next condition, to prove that LISA-D is more expressive than quantifier-free first order logic, is that LISA-D and quantifier-free first order logic must not be equally expressive. This is stated in the following theorem, which is proved by contradiction:

Theorem 3.2 $QL_{QF}(IS_{QF}) \neq_{f,g} QL_{LD}(IS_{LD})$

Proof: Suppose the opposite holds ($QL_{QF}(IS_{QF}) =_{f,g} QL_{LD}(IS_{LD})$).

Then $QL_{LD}(IS_{LD}) \leq_{f^*,g^{-1}} QL_{QF}(IS_{QF})$ (see definition 3.4).

Then a function $h^{\approx} : QL^{\approx}_{QF} \rightarrow QL^{\approx}_{LD}$, as in definition 3.4 exists (see also remark 2 of this definition).

This function is bijective, so a function $h^{\approx -1} : QL^{\approx}_{LD} \rightarrow QL^{\approx}_{QF}$ exists. This leads to the existence of an element $q \in QL^{\approx}_{QF}$ for which the following equation holds:

$$h^{\approx -1}(ANY\_REPETITION\_OF\ P) = q.$$

But the transitive closure of a given set is not expressible in first order logic (see e.g. [AU79]). Certainly the transitive closure is not expressible in a formula in quantifier-free first order logic either, because the class of quantifier-free first order logic is a subset of the class of first order logic.

Contradiction!

So the theorem is correct.

After we have proved the previous two theorems, we can prove the main theorem of this section (LISA-D is more expressive than the class of quantifier-free first order logic):

Theorem 3.3 $QL_{QF}(IS_{QF}) <_{f,g} QL_{LD}(IS_{LD})$

Proof: This follows directly from the theorems 3.1 and 3.2 and from definition 3.5.

3.3 First order logic

In this subsection, we will show that LISA-D is more expressive than first order logic. To be able to do this, we have to define the information system of first order logic first. After that we will define a sequence of transformations, that can be applied to first order formulas and results in formulas that have a format that can be translated to LISA-D. This translation will be shown in the last part of this subsection.

3.3.1 Definition of the information system

The information system of first order logic looks a bit like the constraint information system of first order logic, as it is defined in definition 2.6:

Definition 3.7 *$IS_{FO} := < DBS_{FO}, QL_{FO}, AS_{FO}, \alpha_{FO} >$, where the four components are defined as follows:*

- *DBS_{FO} has the same definition as DBS_{FO} in definition 2.6.*
- *QL_{FO} consists of all first order formulas. These first order formulas are defined as follows:*
  - *$R(x_1, ..., x_n)$ is a first order formula.*
  - *If φ_1 and φ_2 are first order formulas, then $\varphi_1 \wedge \varphi_2$ is a first order formula.*
  - *If φ is a first order formula, then $\neg\varphi$, $\exists_x[\varphi]$ and $\forall_x[\varphi]$ are first order formulas.*

  Remark: *We do not define $\varphi_1 \vee \varphi_2$, because we want to use a minimal set of first order logic. We do need both quantifiers, because of the transformations, described in the next part.*
- *AS_{FO} consists of all relations $R(x_1, ..., x_n)$, with $R \in D^n$, where D is the domain, defined in definition 2.6 (a universal domain, that contains all values of the variables involved in DBS_{FO}).*

- *α_{FO} is a partial function, that assigns an answer $a \in AS_{FO}$ to every instance $db \in DBS_{FO}$ and $q \in QL_{FO}$. This answer consists of all tuples $< c_1, ..., c_n >$, (where all c_i are part of the domain of x_i for all $1 \leq i \leq n$) for which the formula results in* **true**. *The variables $x_1, ..., x_n$ are all free variables in the formula. The bound variables do not appear in the answer.*

3.3.2 Transformations of first order logic

In this part, we will define some transformations that can be applied to formulas in first order logic.indexfirst order logic!transformations (query) These transformations will result in other first order formulas that can be translated to LISA-D in a more straightforward way. When we use the notation Q or Q_y, then Q is some quantifier ($\exists$ or $\forall$). Other notational conventions are that we use $x_1, ..., x_n$ for free variables, $y_1, ..., y_m$ for bound variables and z for a general variable (either bound or free; z is not a real name but stands for either x or y). Some of the following transformation look very much like transformation, described in the previous sections, but as there are some slight changes, we have chosen not to refer to those sections, but to repeat all transformations here.

We will restrict ourselves to formulas without any variables with empty domains. Such variables can easily be removed. Every relation R, containing such a variable can be replaced by **false** (because the population of such a relation would always be empty) and the formula can be further reduced by applying rules like $\neg$**false** = **true**, $\neg$**true** = **false**, **false**$\wedge \varphi$ = **false**, **true**$\wedge \varphi = \varphi$. By using this process, all occurrences of **true** and **false** can be removed unless the complete formula results in **true** or **false**. In that case, we would have a trivial formula. The formula consisting only of **false** can be simulated by a relation $R(x)$ with an empty population. The formula consisting only of **true** can be simulated by the same relation $R(x)$, but with every value in the universal domain in its population.

Before we continue with the transformations, we have to mention, that the use of constants can be simulated with formulas, satisfying our definition. This can be done by using a unary relation with a population of just one element. This is stated in the following lemma:

Lemma 3.4 *$R(c, x_1, ..., x_n, y_1, ..., y_m) \equiv \exists_y [R'(y) \wedge R(y, x_1, ..., x_n, y_1, ..., y_m)$, where the population of R' consists of c only.*

Proof: Suppose $< c_1, ..., c_n >$ is part of the answer to $R(c, x_1, ..., x_n, y_1, ..., y_m)$.

We can assume that $R'(c) \equiv$ **true**, for every result of the query, because the population of R' contains c.

Then also $< c_1, ..., c_n >$ is part of the answer to $R'(c) \wedge R(c, x_1, ..., x_n, y_1, ..., y_m)$.

This includes that this tuple is also part of the answer to $\exists_y [R'(y) \wedge R(y, x_1, ..., x_n, y_1, ..., y_m)]$ (with $y = c$).

Suppose $< c_1, ..., c_n >$ is part of the answer to $\exists_y [R'(y) \wedge R(y, x_1, ..., x_n, y_1, ..., y_m)]$.

This implies directly that this tuple is part of the answer to $R'(c) \wedge R(c, x_1, ..., x_n, y_1, ..., y_m)$, because c is the only element in the population of R'.

This includes that it is part of the answer to $R(c, x_1, ..., x_n, y_1, ..., y_m)$.

So the lemma holds.

We also want to restrict ourselves to formulas with relations that do not contain more than one occurrence of the same variable (like e.g. $R(x, x)$). These relations can be transformed by the introduction of a new relation with only tuples with equal values on all positions in its population, as is stated in the next lemma. In this lemma, we will mention the set I. This is the set of the indices of the variable that occurs more than one times. It is defined as $I \subseteq \{1, ..., p\}$, with $i \in I \Leftrightarrow z_i = x$.

Lemma 3.5 *We separate two cases: the variable that occurs more than once can be free (1) or bound (2).*

1. $R(z_1, ..., z_p)] \equiv R'(z'_1, ..., z'_{p-j}, x)$, *with* j *occurrences of variable* x *between the variables* z_i $(1 \leq i \leq p)$. $z'_i = z_{i+k}$, *if there are* k *elements of* I, *that have a value lower than* $i+k$ *and* $z_{i+k} \neq x$. *The population of* R' *consists of the tuples* $< c'_1, ..., c'_{p-j}, c >$, *where* $c'_i = c_{i+k}$, *just as* z'_i.

2. $R(z_1, ..., z_p)] \equiv R'(z'_1, ..., z'_{p-j}, y)$, *with* j *occurrences of variable* y *between the variables* z_i $(1 \leq i \leq p)$. $z'_i = z_{i+k}$, *if there are* k *elements of* I, *that have a value lower than* $i+k$ *and* $z_{i+k} \neq y$. *The population of* R' *consists of the tuples* $< c'_1, ..., c'_{p-j}, c >$, *where* $c'_i = c_{i+k}$, *just as* z'_i.

Proof: We will prove the first part of this lemma. The second part will not be proved, because this proof follows almost the same procedure.

Suppose $< c, c_1, ..., c_n >$ is part of the answer to $R(z_1, ..., z_p)$.

As the population of $R'(z'_1, ..., z'_{p-j}, x)$ contains the same tuples, except the multiple occurrences of x (which are already filtered out in the answer), this implies directly that $< c, c_1, ..., c_n >$ is also part of the answer to $R'(z'_1, ..., z'_{p-j}, x)$.

Suppose $< c, c_1, ..., c_n >$ is part of the answer to $R'(z'_1, ..., z'_{p-j}, x)$.

As the population of $R(z_1, ..., z_p)$ is equal to that of $R'(z'_1, ..., z'_{p-j}, x)$, except that it contains more copies of the same value, this implies directly that $< c, c_1, ..., c_n >$ is part of the answer to $R(z_1, ..., z_p)$.

So the first part of the lemma holds.

The second part of the lemma can be proved in a quite similar way. We will not show this here.

So we can conclude that the lemma holds.

So in the remainder of this subsection, if we write $R(x_1, ..., x_n, y_1, ..., y_m)$, we can assume that all these x_i and y_j are different.

In the next step, we will transform the formulas in first order logic to *prenex* form. A formula in prenex form has the following format:

$$Q^1_{y_1} ... Q^m_{y_m} [\varphi]$$

where Q^i stands for $\forall$ or $\exists$ for all $1 \leq i \leq m$ and the formula φ is quantifier-free. In [LP81] (chapter 9, section 9.3) a method is described to bring formulas in first order logic to prenex form. In their theorem 9.3.2, they state that every formula in first order logic can be transformed to an equivalent formula in prenex form.

So we can assume in the remainder of this section, that every formula in first order logic has prenex form.

The final step in our transformation will be an extension of the relations in the first order formula. We define the extension of a relation $R(x_1, ..., x_n, y_1, ..., y_m)$ to be a relation $R'(x_1, ..., x_n, y_1, ..., y_m, z_1, ..., z_p)$, where z_1, ..., z_p are all variables that are part of the formula, but not part of R. The way in which these variables are ordered is not important. All relations with the same variables, but with a different ordering of these variables are considered to be equivalent.

The extension of a formula φ is defined as φ', with $R'_i(x_1, ..., x_n, y_1, ..., y_m, z_1, ..., z_p)$ in φ' if $R_i(x_1, ..., x_n, y_1, ..., y_m)$ is part of φ, where $x_1, ..., x_n, y_1, ..., y_m, z_1, ..., z_p$ are all variables in φ. The population of R'_i consists of the tuples $< c_1, ..., c_n, d_1, ..., d_m, e_1, ..., e_p >$, where every element

$< c_1, ..., c_n, d_1, ..., d_m >$ that is an element of the population of R_i is combined with all possible combinations of the values in the domain of the other variables. All these domains were considered to be nonempty.

This extension is well defined, because for every relation holds:

Lemma 3.6 *$\varphi' \equiv \varphi$, where φ' is constructed using the procedure described before this lemma.*

Proof: From the construction-procedure follows that for all relations $R(x_1, ..., x_n, y_1, ..., y_m)$ holds that $< c_1, ..., c_n >$ is part of the answer to R iff $< c_1, ..., c_n, e_{i_1}, ..., e_{i_j} >$ is part of the answer to $R'(x_1, ..., x_n, y_1, ..., y_m, z_1, ..., z_p)$. Here $e_{i_1}, ..., e_{i_j}$ are the values in the domains of all free variables amongst the variables $z_1, ..., z_p$. So the restriction to the original values in R remains the same. Because this holds for all relations, it holds for the complete formula as well.

At last we have reached the right format to translate these first order formulas to LISA-D queries. This translation will be described in the next subsection.

3.3.3 LISA-D compared with first order logic

In this part, we will prove that LISA-D is at least as expressive as first order logic. To do this, we will show that we can translate queries in first order logic, that have the format of the previous part, to LISA-D. This translation is not as straightforward as in the previous sections.

We will asume that every relation $R_i(x_1, ..., x_n, y_1, ..., y_m)$ has predicators $p_{i,j}$ $(1 \leq j \leq n+m)$ from the variables to the relation.

The first step in this proof, will be the translation of the following formula: $\forall_y[R(x, y)]$. The translation will be shown in the next lemma:

Lemma 3.7 *The formula $\forall_y[R(x, y)]$ can be translated to*
$LIST\ INVERT(p')\ x\ BUT\_NOT\ (NOT\ (x\ p_1\ INVERT(p_2)\ y))$,
where $R'(x)$ is a relation with c in its population, iff $< c, d >$ is part of the population of $R(x, y)$ for some d in the domain of y.

Remark: In this lemma, we used the notation $INVERT(p)$, where p is a predicator, resulting in the inverse of this predicator ($p^{\smile}$). This notation is not part of LISA-D, but it is possible to use the inverse of a predicator, because it can be given a name by using the lexicon. We will use the $INVERT$-notation, because it looks more natural in this situation.

Proof: Suppose c is part of the answer to $\forall_y[R(x, y)]$.

Then $< c, d >$ is part of the population of $R(x, y)$, so $< c, d >$ will be part of the answer to $x\ p_1\ INVERT(p_2)\ y$.

As $< c, d >$ is part of the population of $R(x, y)$ for all d in the domain of y, the answer to $NOT\ (x\ RNm(p)\ y)$ will not contain any tuples with c as a first element.

So c will be in the front of the answer to the query
$LIST\ INVERT(p')\ x\ BUT\_NOT\ (NOT\ (x\ p_1\ INVERT(p_2)\ y))$.

Suppose c is in the front of the answer to
$LIST\ INVERT(p')\ x\ BUT\_NOT\ (NOT\ (x\ p_1\ INVERT(p_2)\ y))$.

Then c is part of the population of $R(x, y)$ (because x must be part of the result of $INVERT(p')$.

It also follows that there is no d in the domain of y for which $< c, d >$ is not part of the population of $R(x, y)$, because otherwise c would have been filtered out by the part $BUT\_NOT\ (NOT\ (x\ p_1\ INVERT(p_2)\ y))$.

So $< c, d >$ is part of the population of $R(x, y)$ for all d in the domain of y. This implies directly that c is part of the answer to $\forall_y[R(x, y)]$.

So the lemma holds.

The next formula, we will translate, is stated in the next lemma:

Lemma 3.8 *The first order formula* $\exists_y[R(x,y)]$ *can be translated to* $LIST\ INVERT(p')\ x\ p_1\ INVERT(p_2)\ y$.

Proof: Suppose c is part of the answer to $\exists_y[R(x,y)]$.

Then $< c,d >$ is part of the population of $R(x,y)$ for some d in the domain of y.

This implies directly that c is in the front of the answer to $LIST\ INVERT(p')\ x\ p_1\ INVERT(p_2)\ y$.

Suppose c is in the front of the answer to $LIST\ INVERT(p')\ x\ p_1\ INVERT(p_2)\ y$.

Then $< c,d >$ is part of the population of $R(x,y)$ for some d in the domain of y.

This implies directly that c is part of the answer to $\exists_y[R(x,y)]$.

So the lemma holds.

At this point we can translate all formulas of a more general format. To be able to formulate the resulting LISA-D query in a more readable way, we introduce a notational abbreviation:

$$AND\_ALSO_{i=1}^{n}\ (P_i) := P_1\ AND\_ALSO ... AND\_ALSO\ P_n$$

Lemma 3.9 *A first order formula of the form* $\forall_{y_1}...\forall_{y_m}[R(x,y_1,...,y_m)]$ *can be translated to* $LIST\ INVERT(p')\ x\ BUT\_NOT\ (AND\_ALSO_{j=1}^{m}\ (NOT\ (x\ p_1\ INVERT(p_{1+j})\ y_j)))$

Proof: Suppose c is part of the answer to $\forall_{y_1}...\forall_{y_m}[R(x,y_1,...,y_m)]$.

Then $< c,d_1,...,d_m >$ is part of the population of $R(x,y_1,...,y_m)$ for all d_j in the domain of y_j for all $1 \leq j \leq m$.

Because AND_ALSO results in an intersection and those values c for which c is part of the population of $R(x,y_1,...,y_m)$ for all values in the domain of a certain y_j can be found with the construction in lemma 3.7, c is part of the answer to the LISA-D query, too.

Suppose c is part of the answer to the LISA-D query.

Because of the definition of AND_ALSO and lemma 3.7, it is immediately clear that c is part of the answer to $\forall_{y_1}...\forall_{y_m}[R(x,y_1,...,y_m)]$.

So the lemma holds.

If we perform another generalization, we can translate formulas of the following form:

Lemma 3.10 *A first order formula of the form* $\forall_{y_1}...\forall_{y_m}[R(x_1,...,x_n,y_1,...,y_m)]$ *can be translated to*
$LIST\ AND\_ALSO_{i=1}^{n}(INVERT(p_i')\ x_i\ BUT\_NOT$
$(AND\_ALSO_{j=1}^{m}\ (NOT\ (x_i\ p_i\ INVERT(p_{n+j})\ y_j))))$
where $R'(x_1,...,x_n)$ *is a relation with all tuples* $< c_1,...,c_n >$ *in its population, iff* $< c_1,...,c_n, d_1,...,d_m >$ *is part of the population of* $R(x_1,...,x_n,y_1,...,y_m)$ *with* d_i *in the domain of* y_i *for all* $1 \leq i \leq m$.

Proof: Suppose $< c_1,...,c_n >$ is part of the answer to the first order formula.

Then it is clear that $< c_1,...,c_n >$ is part of the population of $R'(x_1,...,x_n)$. So $< c_1,...,c_n >$ is in the front of the answer to $INVERT(p'i)\ x_i$.

We also know (from lemma 3.9) that for all c_i holds that they are part of the answer to $x_i\ BUT\_NOT\ (AND\_ALSO_{j=1}^{m}\ (NOT\ (x_i\ p_i\ INVERT(p_{n+j})\ y_j)))$.

So $< c_1,...,c_n >$ will be part of the answer to the LISA-D query.

Suppose $< c_1, ..., c_n >$ is part of the answer to the LISA-D query.

From lemma 3.9 we know that all c_i are combined with all combinations of $d_1, ..., d_m$.

As $< c_1, ..., c_n >$ is also part of the population of $R'(x_1, ..., x_n)$, we can conclude that this tuple is part of the answer to the first order formula in the lemma.

So the lemma holds.

A similar generalization, like in the last two steps can be performed on the formula of the form as in lemma 3.8. We will perform this generalization in one step. To be able to formulate the resulting LISA-D query in a more readable way, we introduce another notational abbreviation:

$$CONCATENATE_{i=1}^{n}\ (P_i) := \ P_1\ P_2 ... P_n$$

Lemma 3.11 *A first order formula of the form* $\exists_{y_1} ... \exists_{y_m} [R(x_1, ..., x_n, y_1, ..., y_m)]$ *can be translated to*
$LIST\ AND\_ALSO_{i=1}^{n}(INVERT(p_i')\ x_i\ p_i\ INVERT(p_{n+1})\ y_1$
$CONCATENATE_{j=1}^{m-1}(y_j\ p_{n+j}\ INVERT(p_{n+j+1})\ y_{j+1}))$

Proof: Suppose $< c_1, ..., c_n >$ is part of the answer to the first order formula.

Then $< c_1, ..., c_n >$ is also part of the population of $R'(x_1, ..., x_n)$. So $< c_1, ..., c_n >$ is in the front of the answer to $INVERT(p'i)\ x_i$.

The rest of the LISA-D query
$(x_i\ p_i\ INVERT(p_{n+1})\ y_1\ CONCATENATE_{j=1}^{m-1}(y_j\ p_{n+j}\ INVERT(p_{n+j+1})\ y_{j+1})$ is no real restriction, because we already knew that there were values d_j in the domains of y_j for all $1 \leq j \leq m$.

So $< c_1, ..., c_n >$ is part of the answer to the LISA-D query.

Suppose $< c_1, ..., c_n >$ is part of the answer to the LISA-D query.

Then $< c_1, ..., c_n >$ is part of the population of $R'(x_1, ..., x_n)$.

This implies directly that $< c_1, ..., c_n >$ is part of the answer to the first order formula in the lemma.

So the lemma holds.

We will use a combination of the last two lemmas to translate a more general first order formula. To do this, we have to define the following set I:

$I \subseteq \{0, ..., m\}$
$i \in I \Leftrightarrow Q_{y_i}^{i} = \exists_{y_i}$

We assume that I has got l elements. We define $k_j = p - 1$ (for all $1 \leq j \leq l$) and p is the j'th element of I.

With this notations, we can translate the following first order formulas:

Lemma 3.12 *A first order formula of the form* $Q_{y_1}^{1} ... Q_{y_m}^{m} [R(x_1, ..., x_n, y_1, ..., y_m)]$ *can be translated to*
$LIST\ AND\_ALSO_{i=1}^{n}\ (INVERT\ (p_i')\ x_i$
$(x_i\ BUT\_NOT\ (AND\_ALSO_{j=1}^{k_1}\ (NOT\ (x_i\ p_i\ INVERT(p_{n+j})\ y_j))))$
$x_i\ p_i\ INVERT(p_{n+k_1+1})\ y_{k_1+1}$
$CONCATENATE_{p=1}^{l-1}$
$((y_{k_p+1}\ BUT\_NOT\ (AND\_ALSO_{j=k_p+2}^{k_{p+1}}\ (NOT\ (y_{k_p+1}\ p_{n+k_p+1}\ INVERT(p_{n+j})\ y_j))))$
$y_{k_p+1}\ p_{n+k_p+1}\ INVERT(p_{n+k_{p+1}+1})\ y_{k_{p+1}+1})$
$(y_{k_l+1}\ BUT\_NOT\ (AND\_ALSO_{j=k_l+2}^{m}\ (NOT\ (y_{k_l+1}\ p_{n+k_l+1}\ INVERT(p_{n+j})\ y_j)))))$

Proof: This lemma is only a generalization of the lemmas 3.10 and 3.11. The translation of every sequence of $\forall$'s (zero or more) is followed by the translation of one $\exists$. We assume that P in an operation like $AND\_ALSO_{i=1}^{0}\ P$ will not be executed.

In fact, we follow a path through the existentially quantified variables (just as in lemma 3.11) and combine this path at the right places with the translations of the sequences of universally quantified variables (as in lemma 3.10).

So the lemma holds.

To take the final step towards the translation of a general first order formula (having the format described in the previous part) we only have to change a few parts of the previous LISA-D query. The predicators used in this query have to be replaced by information descriptors that represent the result of (slightly different versions of) the formula. Let the formula to be translated (called φ) be denoted by $Q_{y_1}^1 ... Q_{y_m}^m[\psi]$. We will use the following formulas, derived from this formula:

- Let φ' be ψ (this is equal to φ, without the quantfiers. So the variables $x_1, ..., x_n, y_1, ..., y_m$ are all free and therefore part of the answer to φ'.
- Let φ'' be φ, without all variables y_i. This means that $\varphi'' \equiv \varphi'$, where $R'(x_1, ..., x_n)$ is part of φ'', iff $R(x_1, ..., x_n, y_1, ..., y_m)$ is part of φ'. The population of these relations R' is defined as: $< c_1, ..., c_n >$ is part of the population of R', iff $< c_1, ..., c_n, d_1, ..., d_m >$ is part of the population of R for some $d_1, ..., d_m$.

Next, we will define three types of abbreviations of information descriptors:

- Let $P_{1,i}$ be $h'(\varphi')$, where $h'(\varphi) = h(\varphi)$ as described in the subsection about the translation of quantifier-free first order logic (theorem 3.1), except the first step of this function: $h'(R(x_1, ..., x_n, y_1, ..., y_m)) = INVERT(p_{n+i})$. This abbreviation is defined for all $1 \leq i \leq m$.
- Let $P_{2,i}$ be $h'(\varphi')$ with h' the same as in the previous step, except: $h'(R(x_1, ..., x_n, y_1, ..., y_m)) = p_i$. This abbreviation is defined for all $1 \leq i \leq n + m$.
- Let $P_{3,i}$ be $h'(\varphi'')$ with h' the same as in the previous step, except: $h'(R(x_1, ..., x_n)) = INVERT(p_i)$. This abbreviation is defined for all $1 \leq i \leq n$.

The existence of these three types of abbreviations is proved in the same theorem (theorem 3.1). We will use these abbreviations to formulate the translation of first order formulas (in the format of the previous part) to corresponding LISA-D queries:

Lemma 3.13 *The formula $\varphi = Q_{y_1}^1 ... Q_{y_m}^m[\psi]$, which has the format of the previous part, can be translated to the following LISA-D query:*
$LIST\ AND\_ALSO_{i=1}^{n}\ (P_{3,i}\ x_i$
$(x_i\ BUT\_NOT\ (AND\_ALSO_{j=1}^{k_1}\ (NOT\ (x_i\ P_{2,i}\ P_{1,j}\ y_j))))$
$x_i\ P_{2,i}\ P_{1,k_1+1}\ y_{k_1+1}$
$CONCATENATE_{p=1}^{l-1}$
$((y_{k_p+1}\ BUT\_NOT\ (AND\_ALSO_{j=k_p+2}^{k_{p+1}}\ (NOT\ (y_{k_p+1}\ P_{2,n+k_p+1}\ P_{1,j}\ y_j))))$
$y_{k_p+1}\ P_{2,n+k_p+1}\ P_{1,k_{p+1}+1}\ y_{k_{p+1}+1})$
$(y_{k_l+1}\ BUT\_NOT\ (AND\_ALSO_{j=k_l+2}^{m}\ (NOT\ (y_{k_l+1}\ P_{2,n+k_l+1}\ P_{1,j}\ y_j))))).$

Proof: This lemma follows almost directly from lemma 3.12. The only difference is that we use the answer to φ'', instead of $R'(x_1, ..., x_n)$ and the answer to φ', instead of $R(x_1, ..., x_n, y_1, ..., y_m)$.

The existence of these answers is (as mentioned before) proved in theorem 3.1, because both the formulas φ' and φ'' have no quantifiers, so they are part of the class of quantifier-free first order logic.

So the lemma holds.

After this last lemma, we can easily prove that LISA-D is at least as expressive as first order logic:

Theorem 3.4 $QL_{FO}(IS_{FO}) \leq_{f,g} QL_{LD}(IS_{LD})$

Proof: We define the following functions:

- $f : DBS_{FO} \rightarrow DBS_{LD}$
  $f(R(x_1, ..., x_n, y_1, ..., y_m)) = R$.
- $g : range(\alpha_{FO}) \rightarrow range(\alpha_{LD})$
  $g((x_1, ..., x_n)) = ((x_1, ..., x_n), (x_1, ..., x_n))$
- $h : QL_{FO} \rightarrow QL_{LD}$
  $h(\varphi) = THE\ P$, where P is the LISA-D query of the previous lemma and THE is the operation described in [P94].

The following equation must hold:

$$\forall_{(db,q) \in \alpha_{FO}}\, g(\alpha_{FO}(db, q)) = \alpha_{LD}(f(db), h(q)).$$

The correctness of this equation follows from the result of lemma 3.13. From this lemma follows that the tuples in the front of the result of $h(\varphi)$ are the same tuples as in the result of the first order formula. By using the operation THE (which duplicates the tuples in the front of an information descriptor in such a way that the front and the back become identical), the front and the back of the answer contain the same tuples. Furthermore, we have proved in the previous part of this subsection, that every first order formula can be transformed into the right format.

This means directly that the equation holds.

Due to definition 3.3, the theorem holds.

The next condition, to prove that LISA-D is more expressive than first order logic, is that LISA-D and first order logic must not be equally expressive. This is stated in the following theorem, which is proved by contradiction:

Theorem 3.5 $QL_{FO}(IS_{FO}) \neq_{f,g} QL_{LD}(IS_{LD})$

Proof: Suppose the opposite holds ($QL_{FO}(IS_{FO}) =_{f,g} QL_{LD}(IS_{LD})$).

Then $QL_{LD}(IS_{LD}) \leq_{f^*,g^{-1}} QL_{FO}(IS_{FO})$ (see definition 3.4).

Then a function $h^{\approx} : QL^{\approx}_{FO} \rightarrow QL^{\approx}_{LD}$, as in definition 3.4 exists (remark 2).

This function is bijective, so a function $h^{\approx -1} : QL^{\approx}_{LD} \rightarrow QL^{\approx}_{FO}$ exists. This leads to the existence of an element $q \in QL^{\approx}_{FO}$ for which the following equation holds:

$$h^{\approx -1}(ANY\_REPETITION\_OF\ P) = q.$$

But the transitive closure of a given set is not expressible in first order logic (see e.g. [AU79]).

Contradiction!

So the theorem is correct.

After we have proved the previous two theorems, we can prove the main theorem of this subsection (LISA-D is more expressive than first order logic):

Theorem 3.6 $QL_{FO}(IS_{FO}) <_{f,g} QL_{LD}(IS_{LD})$

Proof: This follows directly from theorems 3.4 and 3.5 and from definition 3.5.

3.4 Horn clause programs

In this subsection, we will show that LISA-D as a query-language is not at least as expressive as the class of Horn clause programs. We will refer to the subsection about Horn clause programs in the section about constraints for the definition of Horn clause programs.

We will start with a definition of the information system of Horn clause programs and in the next part, we will show (with a reference to the subsection mentioned before) that LISA-D is not at least as expressive as Horn clause programs and also that the class of Horn clause programs is not at least as expressive as LISA-D either.

3.4.1 Definition of the information system

The information system of Horn clause programs will be defined as:

Definition 3.8 *Let* $IS_{HC} := < DBS_{HC}, QL_{HC}, AS_{HC}, \alpha_{HC} >$, *where the four components of this structure have the following definitions:*

- DBS_{HC} *will be defined to be the same as* DBS_{FO} *in definition 2.6.*
- QL_{HC} *consists of all possible Horn clause programs (see definition 2.7).*
- AS_{HC} *consists of all combinations of the variables* $(x_1, ..., x_n)$, *where* $x_i \in D$ *for all* $1 \leq i \leq n$ *and* D *is the domain defined in definition 2.6.*
- α_{HC} *is the evaluation function, which determines for every instance of the database and every Horn clause program an answer (in* AS_{HC}*). The result of this function will consist of all tuples* $< c_1, ..., c_n >$, *such that a mapping from* $(x_1, ..., x_n)$ *to the corresponding values in the tuple, makes the Horn clause program result in* **true**. $(x_1, ..., x_n)$ *are the variables in the root of the Horn clause program.*

3.4.2 LISA-D compared with Horn clause programs

To compare LISA-D as a query language with the class of Horn clause programs, we will use the same example of a Horn clause program as in the subsection about Horn clause programs in the section that dealt with constraints:

(1) $S(x_1, x_2) \leftarrow R_1(x_1, x_2)$.
(2) $S(x_1, x_2) \leftarrow R_2(x_1, x_3), S(x_3, x_4), R_3(x_4, x_2)$.

We will show that this program cannot be translated to a LISA-D query either.

Lemma 3.14 *The Horn clause program described above cannot be translated to a corresponding LISA-D query.*

Proof: In lemma 2.7 we have shown that this program cannot be translated to a LISA-D constraint.

The only operation that is added, when we use queries instead of constraints, is confluention. It is quite obvious that this operation is not very useful for this purpose. To calculate the result of each of the involved information descriptors, the complete result has to be calculated and that proved to be impossible.

At this time, we can conclude that LISA-D is not at least as expressive as the class of Horn clause programs.

Theorem 3.7 $QL_{HC}(IS_{HC}) \leq_{f,g} QL_{LD}(IS_{LD})$ *does* **not** *hold.*

Proof: This is almost the same as the prove of theorem 2.4.

When we switch the order of the two information systems involved, we can repeat theorem 2.5 to conclude that the class of Horn clause programs is not at least as expressive as LISA-D either.

Theorem 3.8 $QL_{LD}(IS_{LD}) \leq_{f,g} QL_{HC}(IS_{HC})$ *does* **not** *hold.*

Proof: This is almost the same as the prove of theorem 2.5.

4 LISA-D with macros

In this section, we will examine the expressive power of LISA-D with macros. We only deal with the query-language aspect of LISA-D, because, if we do not use the confluention-operation, then it is true that, if LISA-D as a query-language is more expressive than a certain class of queries, the constraint-part of LISA-D will at least have the same amount of expressive power. In [P94] a method is described to add a macro mechanism to the language LISA-D. We will show that this mechanism improves the expressive power of LISA-D. In the first subsection, we will give a more detailed definition of the macro mechanism.

In the next subsection, we will compare the expressive power of LISA-D with these macros with the class of *stratified logic programs*. LISA-D with macros proves to be more expressive than this class of queries.

In the last subsection, we will show that LISA-D is even more expressive as the bigger class of the *fixed-point queries*. This subsection is a bit less formally written, but the correspondence between the evaluation of recursive macros and the calculation of a (least) fixed-point is quite obvious. It should not be too difficult to give a complete formal proof.

4.1 Definition of macros

In this subsection, we will define the macros and show how they are evaluated. We will also define the information system of LISA-D with these macros.

To define macros in LISA-D, we use some definitions and notations from [P94]. In this book an evolving environment is handled however, so we have to make some changes.

The general format of a macro definition is:

$$LET\ \omega_0\ X_1\ \omega_1\ ...\ X_n\ \omega_n\ BE\ E$$

where $X_1, ..., X_n$ are the names of variables and the string $\omega_0, ..., \omega_n$ is the name of the macro. The expression E (the *body* of the macro) may contain other calls of macros. Macros can even be recursive.

We will first give an informal definition of the evaluation-process of macros. It will be an inductive definition:

- At step 0, we have the call of the macro.
- Step $n+1$ results from step n by replacing all macro-calls in the result of step n by their bodies.

After each step, we check the result of the resulting information descriptor. To be able to do this, the macro-calls have to be eliminated, because these are not recognized by the evaluation function of LISA-D (μ and $\mathbf{D}$). If two steps $m+1$ and m result in the same tuples, the evaluation-process is terminated.

This process can be stated more formally as in the following procedure. It is stated in some sort of pseudo-code:

```
PROCEDURE EvaluateMacro (MACRO m): Result;
BEGIN
    i := 1;
    Result := ∅;
    AuxResult := ∅;
    REPEAT
        Result := Result ∪ μ[D[ε^i_LM(m)](< M, V >)](Pop);
        AuxResult := Result ∪ μ[D[ε^{i+1}_LM(m)](< M, V >)](Pop);
        i := i + 1
    UNTIL Result = AuxResult
END Procedure;
```

where the function ϵ_{LM} is defined as:

$\epsilon_{LM} : QL_{LM} \rightarrow QL_{LM}$
$\epsilon^0_{LM}(\omega_0\ P_1\ \omega_1\ ...\ P_n\ \omega_n) = P_\emptyset$
$\epsilon^{n+1}_{LM}(\omega_0\ P_1\ \omega_1\ ...\ P_n\ \omega_n) =$
$\quad UNION_{M(\omega_0\ X_1\ \omega_1\ ...\ X_n\ \omega_n)\downarrow} M(\omega_0\ X_1\ \omega_1\ ...\ X_n\ \omega_n)[X_i := P_i]^n_{i=1}$
$\quad [\psi_0\ Y_1\ \psi_1\ ...\ Y_m\ \psi_m := \epsilon^n_{LM}(\psi_0\ Y_1\ \psi_1\ ...\ Y_m\ \psi_m)]$
for all macro-calls in $M(\omega_0\ P_1\ \omega_1\ ...\ P_n\ \omega_n)$.

The construction $E[A := B]$ means that every occurrence of A in E is replaced by B; the notation $P_\emptyset$ stand for an information descriptor, which results in the empty set of tuples.

The information system of LISA-D with macros is defined as:

Definition 4.1 *Let $IS_{LM} := < DBS_{LM}, QL_{LM}, AS_{LM}, \alpha_{LM} >$, where all these four components have the same definitions as in definition 3.2. It should be noted that the definition of information descriptors (not given in that definition) is different here, because they can contain calls of macros.*

4.2 Stratified logic programs

In this subsection, we will show that LISA-D with macros is more expressive than the class of stratified logic programs. We will show this for queries.

In the first part we will give a definition of stratified logic programs and after that we are able to give a definition of the information system of this class of queries.

In the next part, we will define a way of translating the clauses of a stratified logic program to corresponding macros and prove that this translation results in the same answer as the stratified logic program. It is quite easy to prove after that that LISA-D with macros is at least as expressive as the class of stratified logic programs. Finally it is shown that the query called *Even*, which (as is shown in many articles) cannot be expressed as a stratified logic program, can be expressed in a LISA-D macro. So LISA-D with macros and the stratified logic programs are not equally expressive. So LISA-D with macros is more expressive than this class of queries.

4.2.1 Definition of the information system

Before we are able to define the infomation system of stratified logic programs, we have to give a definition of this class of programs first. We refer to definition 2.7 for the definitions of some of the used terms. This definition is taken from [K91].

Definition 4.2 *A* **general logic program** *consists of a set of clauses, which have the following form:*

$$A \leftarrow B_1, ..., B_n.$$

where the conclusion A is a nonterminal atomic formula and the premises $B_1, ..., B_n$ are atomic formulas or **negated** *atomic formulas.*

A **stratified logic program** *will be a general logic program in which the clauses can be divided into a partition $P = \cup^l_{i=1} P_l$ such that:*

- *If a nonterminal atomic formula S_k occurs positively amongst the premises of a clause in some P_i, then all clauses having S_k as a conclusion have to be contained in $\cup^i_{j=1} P_j$.*
- *If a nonterminal atomic formula S_k occurs negatively amongst the premises of a clause in some P_i, then all clauses having S_k as a conclusion have to be contained in $\cup^{i-1}_{j=1} P_j$. In fact P_1 will not contain any negated nonterminal atomic formulas.*

Informally, we can say that the program can be divided into *strata* and every nonterminal atomic formula, used in some stratum is defined in the same stratum or in a lower stratum. Further, no recursion is allowed through negation. A Horn clause program (see definition 2.7 can be seen

as a stratified logic program with only one stratum. Stratified logic programs are therefore at least as expressive as the Horn clause programs. In fact they are more expressive than Horn clause programs, because (see e.g. [SW91]) the query NotTC can be expressed as a stratified logic program and not as a Horn clause program.

Example 4.1 *The following set of clauses forms a stratified logic program:*

(1) $S_0(x_1, x_2) \leftarrow R_1(x_1, x_2)$.
(2) $S_0(x_1, x_2) \leftarrow R_2(x_1, x_3), \neg S_1(x_3, x_2)$.
(3) $S_1(x_3, x_2) \leftarrow \neg R_3(x_3, x_2)$.
(4) $S_1(x_3, x_2) \leftarrow R_4(x_3, x_4), S_1(x_4, x_2)$.

The clauses (3) and (4) form the lowest stratum (P_1). This stratum is in fact not a Horn clause program, because the terminal atomic formula R_3 occurs negatively in clause (3). The clauses (1) and (2) form a higher stratum (P_2), because clause (2) contains a negated occurrence of the nonterminal atomic formula S_1.

The information system of stratified logic programs can be defined very easily at this time. It will be defined as:

Definition 4.3 *Let $IS_{SL} := < DBS_{SL}, QL_{SL}, AS_{SL}, \alpha_{SL} >$, where the four components of this structure have the following definitions:*

- *DBS_{SL} will be defined as DBS_{HC} in definition 3.8.*
- *QL_{SL} consists of all stratified logic programs.*
- *AS_{SL} has the same definition as AS_{HC} in definition 3.8.*
- *α_{SL} is the evaluation function, which determines for every instance of the database and every stratified logic program an answer (in AS_{SL}). The result of this function will consist of all tuples $< c_1, ..., c_n >$, such that a mapping from $(x_1, ..., x_n)$ to the corresponding values in the tuple, make the stratified logic program result in* **true**. *$(x_1, ..., x_n)$ are the variables in the conclusion of the root of the stratified logic program.*

The evaluation process of stratified logic programs can be formalized in a quite similar way as with the LISA-D macros:

```
PROCEDURE EvaluateStratifiedProgram (STRATIFIED_PROGRAM slp): Result;
BEGIN
    i := 1;
    Result := ∅;
    AuxResult := ∅;
    REPEAT
        Result := Result ∪ α_SL((D, R̄), ε^i_SL(slp));
        AuxResult := Result ∪ α_SL((D, R̄), ε^{i+1}_SL(slp));
        i := i + 1
    UNTIL Result = AuxResult
END Procedure;
```

In this procedure, we assume that a stratified logic program *slp* is represented by the nonterminal atomic formula that is the root of the program.

The function ϵ_{SL} is defined as:indexstratified logic program!evaluation function

$\epsilon_{SL} : QL_{SL} \rightarrow QL_{SL}$
$\epsilon^0_{SL}(slp) = S_\emptyset$
$\epsilon^{n+1}_{SL}(slp) = \vee^m_{i=1} C_i[S_j(x_{j_1}, ..., x_{j_k}) := \epsilon^n_{SL}(S_j(x_{j_1}, ..., x_{j_k}))]$

In this definition $S_\emptyset$ stands for a stratified logic program, which results in the empty set of tuples.
The function α_{SL} will be defined as:

- $\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow .\}) = < c_1,...,c_n >$, where c_i is part of the domain of x_i for all $1 \leq i \leq n$.
- $\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow R(x_{i_1},...,x_{i_m}).\}) = < c_1,...,c_n >$, where c_{i_j} is part of the population of $R(x_{i_1},...,x_{i_m})$ for all $1 \leq j \leq m$ with $1 \leq i_j \leq n$ and c_k is part of the domain of x_k for all $k \neq i_j$ $(1 \leq j \leq m)$.
- $\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow \neg(B_1).\}) = \neg(\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow B_1.\}))$
- $\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow B_1,...,B_m.\}) = \alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow B_1.\}) \wedge ... \wedge \alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow B_m.\})$
- $\alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow C_1.,...,S(x_1,...,x_n) \leftarrow C_m.\}) = \alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow C_1.\}) \vee ... \vee \alpha_{SL}((D,\overline{R}),\{S(x_1,...,x_n) \leftarrow C_m.\})$

In this definition every term B_i stands for an atomic formula and C_i stands for a sequence of zero or more atomic formulas.

4.2.2 LISA-D compared with stratified logic programs

In this part, we will show that LISA-D with macros is more expressive than the class of stratified logic programs. To do this, we will first describe how a general stratified logic program can be translated to a number of LISA-D macros and we will prove that the information descriptor consisting of a call of the macro with the same name as the root of the stratified logic program, results in the same tuples as the stratified logic program itself. After that, it is easy to show that LISA-D (with macros) is at least as expressive as this class of queries. As both classes of queries prove not to be equally expressive, LISA-D with macros is even more expressive than the stratified logic programs.

In the following definition, we describe a relation between clauses in a stratified logic program and the call of a macro:

Definition 4.4 *We will define a macro for every clause $S_i \leftarrow B_1,...,B_k.$ in the stratified logic program. This macro is defined as:*

$$LET\ M_i\ BE\ E$$

where $E = h'(\varphi)$. φ is the formula in first order logic that is formed by the premises of the clause and h' is the function described in theorem 3.4.

Such a formula φ exists, because these sequences of premises form a subset of first order logic (with only $\wedge$, $\neg$ and $\exists$). It is quite clear that this is correct, because the premises can be seen as relations (with or without negation) separated by conjunctions. The variables in the conclusion of the clause will be handled as free variables. The others are considered to be bound by an existential quantifier. If nonterminal atomic formulas occur amongst the premises, they can be replaced by the calls of their macros, so they are in fact handled in the same way as the terminal atomic formulas.

Definition 4.5 *To translate a general stratified logic program to a LISA-D query, we define the following function:*

$$h : QL_{SL} \rightarrow QL_{LM}$$
$$h(P) = M_0$$

where P is a stratified logic program. M_0 is the call of the macro which is the translation of the premises of those clauses with S_0 as a conclusion. S_0 is supposed to be the root of P.

Example 4.2 *The stratified logic program of example 4.1 will be translated to the following macros:*

$LET\ M_0\ BE\ E_1$
$LET\ M_0\ BE\ E_2$

$LET\ M_1\ BE\ E_3$
$LET\ M_1\ BE\ E_4$

where E_i is the translation of the premisses of clause (i) ($1 \leq i \leq 4$). S_0 is the root of the stratified logic program, so $h(P) = M_0$.

Before we can prove that this translation function is valid, we have to prove the following lemma:

Lemma 4.1 $\alpha_{SL}(\epsilon^n_{SL}(P)) = \alpha_{LM}(\epsilon^n_{LM}(h(P)))$ *for all* $n \geq 0$.

Proof: This proof will be given, using induction on n.

- If $n = 0$, they both result in $\emptyset$.
- Suppose the equation holds for a certain n.

  $\alpha_{SL}(\epsilon^{n+1}_{SL}(P)) =$ (definition of ϵ_{SL})
  $\alpha_{SL}(\vee^m_{i=1} C_i[S_j(x_{j_1}, ..., x_{j_k}) := \epsilon^n_{SL}(S_j(x_{j_1}, ..., x_{j_k}))]) =$ (induction step, definition of macros and UNION)
  $\alpha_{LM}(UNION_{M(\omega_0\ X_1\ \omega_1\ \ldots\ X_n\ \omega_n)\downarrow} M(\omega_0\ X_1\ \omega_1\ ...\ X_n\ \omega_n)[X_i := P_i]^n_{i=1}$
  $[\psi_0\ Y_1\ \psi_1\ ...\ Y_m\ \psi_m := \epsilon^n_{LM}(\psi_0\ Y_1\ \psi_1\ ...\ Y_m\ \psi_m)] =$ (definition of ϵ_{LM})
  $\alpha_{LM}(\epsilon^{n+1}_{LM}(h(P)))$

  By using induction, we can conclude that the lemma holds.

Remark: We would have liked to give an example of a stratified logic program and its translation to LISA-D and how they both will be evaluated to the same result, but the translation of a stratified logic program of any reasonable size will be too big to make this process any clearer (due to the result of lemma 3.12). In fact, after a few steps of the ϵ_{LM}-function, the resulting LISA-D query could hardly be written on one page.

After this lemma, we can easily prove that LISA-D (with macros) is at least as expressive as the class of stratified logic programs:

Theorem 4.1 $QL_{SL}(IS_{SL}) \leq_{f,g} QL_{LM}(IS_{LM})$

Proof: We define the following functions:

- $f : DBS_{SL} \rightarrow DBS_{LM}$
  $f(R(x_1, ..., x_n)) = R$.
- $g : range(\alpha_{SL}) \rightarrow range(\alpha_{LM})$
  $g((x_1, ..., x_n)) = ((x_1, ..., x_n), (x_1, ..., x_n))$
- h will be defined as in definition 4.5

The following equation must hold:

$$\forall_{(db,q)\in\alpha_{SL}}\, g(\alpha_{SL}(db, q)) = \alpha_{LM}(f(db), h(q)).$$

The correctness of this equation follows from the result of lemma 4.1. This lemma implies that the tuples in the result of $h(P)$ are the same tuples as in the result of P after n evaluation steps.

The only problem that can arise is that one of the evaluations may stop, while the other one continues. The evaluation process of $h(P)$ stops if the result after a step m is equal to the

result after step $m+1$. This also holds for the evaluation of P. So both evaluation processes will stop after the same step.

This means directly that the equation holds.

Due to definition 3.3, the theorem holds.

The next condition, to prove that LISA-D with macros is more expressive than the stratified logic programs, is that LISA-D with macros and the stratified logic programs must not be equally expressive. This is stated in the following theorem, but to be able to prove this theorem, we need the following macro:

```
LET EVEN n BE
  IF (n = 0) OR (n = 1)
  THEN n
  ELSE EVEN (n - 2)
```

It is quite obvious that this macro results in 0 if n is even and in 1 if n is odd.

With this macro, we can prove the following theorem:

Theorem 4.2 $QL_{SL}(IS_{SL}) \neq_{f,g} QL_{LM}(IS_{LM})$

Proof: Suppose the opposite holds ($QL_{SL}(IS_{SL}) =_{f,g} QL_{LM}(IS_{LM})$).

Then $QL_{LM}(IS_{LM}) \leq_{f^*,g^{-1}} QL_{SL}(IS_{SL})$ (see definition 3.4).

Then a function $h^{\approx} : QL_{SL}^{\approx} \rightarrow QL_{LM}^{\approx}$, as in definition 3.4 exists (remark 2).

This function is bijective, so a function $h^{\approx -1} : QL_{LM}^{\approx} \rightarrow QL_{SL}^{\approx}$ exists. This leads to the existence of an element $q \in QL_{SL}^{\approx}$ for which the following equation holds:

$$h^{\approx -1}(EVEN\ NUMBER(R)) = q,$$

where the macro $EVEN\ n$ is the macro described before this theorem.

But this query (called Even (see [SW91])) is not expressible in a stratified logic program (for a proof see [CH82]).

Contradiction!

So the theorem is correct.

After we have proved the previous two theorems, we can prove the main theorem of this subsection (LISA-D with macros is more expressive than the class of stratified logic programs):

Theorem 4.3 $QL_{SL}(IS_{SL}) <_{f,g} QL_{LM}(IS_{LM})$

Proof: This follows directly from theorems 4.1 and 4.2 and from definition 3.5.

4.3 Fixed-point queries

In this subsection, we will show that LISA-D with macros is more expressive than the class of fixed-point queries.

In the first part we will define what we mean by a fixed-point and a fixed-point operation. After that we will define the information system of fixed-point queries.

In the next part we will show that every fixed-point query can be translated to a corresponding LISA-D query. This will be shown by proving that a recursive macro can be used to calculate the least fixed-point of a first order formula.

4.3.1 Definition of the information system

In this part, we will define the information system of the class of fixed-point queries (see e.g. [CH82], [GS86]). We will first define the operation F:

$$F(\varphi) = \cup_{i=1}^{\infty}(\varphi)^i$$

Such an operation will be called a *fixed-point operation.*

We define n to be a *fixed-point*, if the following equation holds:

$$\cup_{i=1}^{n}(\varphi)^i = \cup_{i=1}^{n+1}(\varphi)^i$$

The operation F has a fixed-point, if the sequence $(\varphi)^1$, $(\varphi)^2$, ... is monotone. A fixed-point x is defined to be the *least fixed-point* if (1) x is a fixed-points and (2) for all fixed-points y holds that $x \leq y$.

We will continue with a definition of the information system of fixed-point queries. Informally, we can say that the fixed-point queries are those queries that can be formed by using the queries in first order logic, together with the fixed-point operation F. The exact definition of the information system of fixed-point queries will be:

Definition 4.6 *Let* $IS_{FP} := < DBS_{FP}, QL_{FP}, AS_{FP}, \alpha_{FP} >$*, where the four components of this structure have the following definitions:*

- DBS_{FP} *has the same definition as* DBS_{FO} *in definition 2.6.*
- QL_{FP} *consists of all fixed-point formulas, which are defined as:*
  - $R(x_1, ..., x_n)$ *is a fixed-point formula.*
  - *If* φ_1 *and* φ_2 *are fixed-point formulas, then* $\varphi_1 \wedge \varphi_2$ *is a fixed-point formula.*
  - *If* φ *is a fixed-point formula, then* $\neg\varphi$, $\exists_x[\varphi]$ *and* $\forall_x[\varphi]$ *are fixed-point formulas.*
  - *If* φ *is a* **first order formula**, *then* $F(\varphi)$ *is a fixed-point formula, where* F *is defined as described before.*

  Remark: *In [GS86] it is shown that it is no restriction to allow only one occurrence of a fixed-point operation in the formula (possibly followed by some first order operations), so the last rule in this definition is well-defined.*

  In the last rule, φ *has to be monotone (as mentioned before). This is true, iff* $F(\varphi)$ *appears positive (under an even number of negations) in the complete fixed-point formula.*
- AS_{FP} *is defined as* AS_{FO} *in definition 3.7.*
- α_{FP} *is a partial function, that assigns an answer* $a \in AS_{FP}$ *to every instance* $db \in DBS_{FP}$ *and* $q \in QL_{FP}$*. This answer consists of all tuples* $< c_1, ..., c_n >$*, (where all* c_i *are part of the domain of* x_i *for all* $1 \leq i \leq n$*) for which the formula results in* **true**. *The variables* $x_1, ..., x_n$ *are all free variables in the formula. The bound variables do not appear in the answer.*

4.3.2 LISA-D compared with fixed-point queries

We define the following function that will be used to translate fixed-point queries to LISA-D:

$h : QL_{FP} \rightarrow QL_{LM}$

$h(\varphi) = h'(\varphi)$

where h' is the function defined in theorem 3.4, except that the occurrence of $F(\psi)$ has to be translated with a call of the macro FP that will be defined as:

$$LET\ FP\ BE\ E\ FP$$

where E is defined to be $h'(\psi)$.

To prove that the results of φ and $h(\varphi)$ are equivalent, we only have to prove that the macro FP results in the least fixed-point of φ.

Lemma 4.2 *The macro FP as defined before results in the least fixed-point of the formula φ.*

Proof: According to theorem 3.4, the expression E will be a translation of φ.

It is quite obvious that the execution of the procedure EvaluateMacro (defined in the first part of this section) applied to this macro FP (which is recursive) results in E^n, where n is the least number for which $E^n = E^{n+1}$. As E will be monotone (because φ is monotone), this is the least fixed-point of E and therefore of φ.

So the lemma holds.

An example of this translation will be:

Example 4.3 *We have the following fixed-point query:*

$$\forall_{y_1}\exists_{y_2}[R_1(x_1, y_1) \wedge F(\forall_{y_3}[R_2(x_2, y_2, y_3)]) \wedge \neg R_3(x_2, y_2)]$$

The fixed-point part of this query $F(\forall_{y_3}[R_2(x_2, y_2, y_3)])$ can be translated to:

$LET\ FP\ BE\ E\ FP$

where E is the translation of the (first order) formula $\forall_{y_3}[R_2(x_2, y_2, y_3)]$, according to theorem 3.4. According to lemma 4.2, this results in the fixed-point of E and also of the first order formula.

The resulting formula (with the result of the fixed-point operation seen as an ordinary relation), can be translated according to theorem 3.4, because it is a normal first order formula now.

So both the fixed-point formula and its translation result in the same tuples.

Remark: We have chosen not to write out the translation according to theorem 3.4 and therefore we did not work with explicit populations of the relations, because the translations of the first order formulas would be too complex to make it any clearer.

At this time, we can easily prove that LISA-D (with macros) is at least as expressive as the class of fixed-point queries:

Theorem 4.4 $QL_{FP}(IS_{FP}) \leq_{f,g} QL_{LM}(IS_{LM})$

Proof: We define the following functions:

- $f : DBS_{FP} \rightarrow DBS_{LM}$
  $f(R(x_1, ..., x_n)) = R$.
- $g : range(\alpha_{FP}) \rightarrow range(\alpha_{LM})$
  $g((x_1, ..., x_n)) = ((x_1, ..., x_n), (x_1, ..., x_n))$
- h will be defined as in the beginning of this part.

The following equation must hold:

$$\forall_{(db,q)\in\alpha_{FP}} g(\alpha_{FP}(db, q)) = \alpha_{LM}(f(db), h(q)).$$

This follows directly from theorem 3.4 and lemma 4.2, because the result of the macro FP can be seen as a relation in a normal first order formula. So the formula φ can be seen as a ordinary first order formula.

This implies that the tuples in the result of $h(\varphi)$ contains the same tuples as the result of φ.

This means directly that the equation holds.

Due to definition 3.3, the theorem holds.

The next condition, to prove that LISA-D with macros is more expressive than the class of fixed-point queries, is that LISA-D with macros and the fixed-point queries must not be equally expressive. This is stated in the following theorem:

Theorem 4.5 $QL_{FP}(IS_{FP}) \neq_{f,g} QL_{LM}(IS_{LM})$

Proof: Suppose the opposite holds ($QL_{FP}(IS_{FP}) =_{f,g} QL_{LM}(IS_{LM})$).

Then $QL_{LM}(IS_{LM}) \leq_{f^*,g^{-1}} QL_{FP}(IS_{FP})$ (see definition 3.4).

Then a function $h^{\approx} : QL_{FP}^{\approx} \rightarrow QL_{LM}^{\approx}$, as in definition 3.4 exists (remark 2).

This function is bijective, so a function $h^{\approx -1} : QL_{LM}^{\approx} \rightarrow QL_{FP}^{\approx}$ exists. This leads to the existence of an element $q \in QL_{FP}^{\approx}$ for which the following equation holds:

$$h^{\approx -1}(EVEN\ NUMBER(R)) = q,$$

where the macro $EVEN\ n$ is the macro described in the previous subsection.

But this query (called Even (see [SW91])) is not expressible in a fixed-point query (for a proof see [CH82]).

Contradiction!

So the theorem is correct.

After we have proved the previous two theorems, we can prove the main theorem of this subsection (LISA-D with macros is more expressive than the class of fixed-point queries):

Theorem 4.6 $QL_{FP}(IS_{FP}) <_{f,g} QL_{LM}(IS_{LM})$

Proof: This follows directly from theorems 4.4 and 4.5 and from definition 3.5.

5 Conclusion

In this thesis, we have examined the expressive power of LISA-D. The two aspects of LISA-D (constraint-language and query-language) proved to have the same expressive power, compared with the hierarchy of classes of constraints or queries we used. We have seen that LISA-D can express all constraints and queries in first order logic, but not all constraints and queries in the class of the Horn clause programs.

As we mentioned in the introduction, the variables in the answers to the queries, were difficult to deal with, but it was not impossible. So the position of LISA-D in the hierarchy of classes would not change if we eliminated the operations on constraints (AND, OR, NO, FOR_EACH), but the resulting constraints would be less easy to formulate and understand. The same holds for the confluention operation on queries. We have not used it in our thesis, but in practice, it could lead to more readable queries.

Another conclusion can be that the operations that are part of LISA-D have some differences with the meaning they seem to have, according to their names. For example the operations AND_ALSO or INTERSECTION cannot be used as general as the $\wedge$ operation in first order logic, because the variables in the front and the back of the involved information descriptors need to be of the same types (by using AND_ALSO only the fronts have to be of the same type). That does not hold for the $\wedge$ operation in first order logic. A formula like $R_1(x_1, x_2) \wedge R_2(x_2, x_3, x_4)$ is a well-formed first order formula.

The use of macros does increase the expressive power of LISA-D quite a lot. As we have shown, LISA-D with these macros is more expressive than the class of fixed-point queries. Because we did not use the confluention operation to prove this, it also holds for constraints.

We did not compare the expressive power of LISA-D with macros with the class of queries computable in polynomial time (the class $QPTIME$ mentioned in the introduction). So this is a point for further research in the future and perhaps (if LISA-D with macros even proves to be more expressive than this class) even bigger classes have to be examined (classes like $QPSPACE$ (see [CH82]) or *bounded-loop queries* or *while-queries* (see [C88]).

Because we do not know the exact expressive power of LISA-D with macros, it is difficult to say if it is necessary to add operations to LISA-D to improve its expressive power. It has proved to be good (from the point of the expressive power) to add some sort of macro-mechanism to the standard-operations of LISA-D.

References

[AU79] A.V. Aho and J.D. Ullman. *Universality of data retrieval languages*, 6th ACM symposium on principles of programming languages (pp. 110-117), 1979.

[C88] Ashok K. Chandra. *Theory of Database Queries*, Proceedings of the Seventh ACM symposium on Principles of Database Systems, 1988.

[CH82] Ashok K. Chandra and David Harel. *Structure and Complexity of Relational Queries*, Journal of Computer and System Sciences, vol. 25, pp. 99-128, 1982.

[CH85] Ashok K. Chandra and David Harel. *Horn Clause Queries and Generalizations*, Journal Logic Programming, vol. 1, pp. 1-15, 1985.

[GS86] Y. Gurevich and S. Shelah. *Fixed-point extensions of first order logic*, Annals of pure and applied logic, vol. 32, pp. 265-280, 1986.

[H93] Arthur H.M. ter Hofstede. *Information modelling in data intensive domains*, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[K91] Phokion G. Kolaitis. *The Expressive Power of Stratified Logic Programs*, Information and Computation, vol. 90, pp. 50-66, 1991.

[LP81] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the theory of computation*, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[P94] H.A. Proper. *A theory for Conceptual Modelling of Evolving Application Domains*, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[SW91] Peter Schäuble and Beat Wüthrich. *On the Expressive Power of Query Languages*, 1991.

Index