Towards a Theory of Elaboration Tolerance: Logic Programming Approach

Michael Gelfond
University of Texas at El Paso
mgelfond@cs.utep.edu

Halina Przymusinska
California State Polytechnic University
halina@cs.ucr.edu

Abstract

This paper is an attempt at mathematical investigation of software development process in the context of declarative logic programming. We introduce notions of specification and specification constructor which are developed from natural language description of a problem. Generalizations of logic programs, called lp-functions are introduced to represent these specifications. We argue that the process of constructing lp-function representing a specification S should be supported by certain types of mathematical results which we call representation theorems. We present two such theorems to illustrate the idea.

1 Introduction

This paper is written in the framework of declarative logic programming paradigm (see, for instance, [17, 14]) which strives to reduce a substantial part of a programming process to the description of objects comprising the domain of interest and relations between these objects. After such description is produced by a programmer it can be queried to establish truth or falsity of statements about the domain or asked to find objects satisfying various

properties. The paradigm, which first appeared in artificial intelligence, had substantial impact on other areas of computing science, such as databases, programming languages and software engineering, and led to the development of programming language Prolog and its dialects (see [6, 16]).

In declarative logic programming the software development process starts with a natural language description of the domain. The description is analyzed and represented in a formal logical language with precisely defined entailment relation. Originally, proponents of the approach believed that first-order logic with classical entailment can serve as the main tool for knowledge representation. It was soon realized, however, that for representation of commonsense knowledge, necessary in many non-mathematical domains, this tool is inadequate. The difficulty is rather deep and related to so-called "monotonicity " of theories based on classical logic. A logic is called monotonic if the addition of new axioms to the theory based on it never leads to the loss of any theorems proved in this theory. Commonsense reasoning is nonmonotonic: new information constantly forces us to withdraw previous conclusions. This observation has led to the development and investigation of new logical formalisms, nonmonotonic logics (see [18, 22, 19]).

This paper is an attempt at mathematical investigation of software development process in the context of declarative logic programming. In particular we are interested in investigating an important property of formal representations called *elaboration tolerance*. Informally, a formal representation T of a specification S is called *elaboration tolerant* if a small modification of S does not require major changes of T. Development of formal representations with this property is one of the most important challenges facing researches concerned with declarative representation of knowledge. So far not much work has been directed towards systematic investigation of tolerance. Partially, this is due to the fact that degree of tolerance depends on the multiplicity of factors. The (incomplete) list includes the type of specification S and its intended use, the syntax and semantics of the language used for representation T of S, and T's syntactic form. Insufficient development of knowledge representation languages and their mathematical theories, as well as the absence of a general mathematical notion of specification tailored for declarative logic programming paradigm, also contribute to the complexity of the problem.

In the last decade however, the field of knowledge representation matured significantly. In particular a substantial progress has been made towards

developing mathematical foundations necessary to understand properties of representations in various formalisms and the relations between them. People working in this area accumulated valuable experience in constructing representations of several important domains which are reasonably tolerant w.r.t. certain types of possible modifications. The main goal of this paper is to contribute to the better understanding and to the generalization of this experience. We hope that some of the presented ideas will prove useful for the development of a comprehensive theory of formal representation for declarative logic programming.

In the first section we start with giving a mathematical definition of the notion of *specification*, independent of particulars of the language used for its description. Intuitively we view a specification as a (possibly incomplete) description of new relations between objects from the domain of discourse given in terms of the old, known relations. [1] We will not concentrate here on the most general notion of specification. Instead, we will limit ourselves to a special case of so called functional specifications. These specifications can be defined by a specifier directly in a simple set-theoretic language, or they can be build from previously defined specifications with the help of *specification constructors* - simple functions from specifications to specifications. Several specification constructors commonly used in practice will be discussed in this paper. Discovery of a comprehensive list of such constructors is an important topic for future research.

In the next section we define a notion of a *representation* of a specification S. This notion is language dependent and requires a choice of the representation language. Our choice is based on the belief that any such language should have the following properties:

- possibility of concise representation of statements of natural language commonly used in informal descriptions of various domains. In particular we should be able to represent:

[1] This view is substantially different from general notions of specifications [9, 20, 25] tailored to imperative paradigm of programming as well as from traditional logic programming approach where a program, understood as a first-order theory, played a role of specification. The inadequacy of the former for our purpose is to be expected while the latter is not suitable for problems whose solutions require representation of commonsense knowledge due to its limited expressive power.

  - default assumptions, i.e. statements of the form "Elements of the class A normally (typically, as a rule) have property P"
  - epistemic statements, i.e. statements of the form "P is unknown", "P is possible", etc.

- availability of query answering systems which allow rapid prototyping
- existence of a well developed and mathematically precise semantics of the language
- existence of mathematical theory that provides basis for proving properties of the theories in the given language.

These requirements determined our choice of the language of declarative logic programming as the representation language.[2] A logic programming representation (or simply *lp-representation*) of a functional specification S consists of a theory (program) in this language together with a collection of input and output predicates. Entities of this sort are called *lp-functions*.
Space limits prevent us from presenting serious justification of this choice. Such a justification together with a discussion of the alternatives probably deserves a separate paper. The next few lines however give a hint on the reasons for our decision:
A discussion of methodology of expressing normative and epistemic statements in the language of declarative logic programming together with related mathematical results can be found, for instance, in [4]. The Prolog programming language and its recent dialects can be used for rapid prototyping. It is worth mentioning that some of these dialects (see, for instance [24, 1]) are more declarative and have better terminating properties than Prolog. As a semantic basis for the language we use the answer sets semantics of [12]. This selection is based on the simplicity of the semantics and the existence of a substantial body of knowledge related to it. There are several other interesting semantics of declarative logic programming, which give slightly different interpretations to logic programming connectives. Further research is necessary to determine the ramifications of our choice of the semantics

[2] In this paper by a declarative logic program we mean a program with negation as failure, strong (classical) negation and epistemic disjunction. A precise definition is given in section 3.

but we believe that the general approach advocated in this paper does not significantly depend on it.

We hope that investigation of the notions of specification, specification constructor, and an lp-function representation of a specification will significantly facilitate the process of a gradual development of knowledge based software systems.
The process will start with the natural language description of the knowledge about a particular domain to be represented in a computer. This description provides an informal specification of the problem at hand. We will divide the software development process based on the given informal specification into three stages with three main characters, specifier, representer and implementer.

In the first stage the specifier will divide available knowledge about the domain into small independent chunks (modules) for which functional specifications $s_1, \ldots, s_n$ will be defined in a mixture of natural and mathematical language. He will proceed by applying standard specification constructors to build specifications for larger modules until the final specification will be obtained. For the sake of discussion, let us assume that this specification has the form, say, $S = k(s_1, m(s_2, \ldots, s_n))$ where k and m are names of the standard constructors used in the process.

In the next step of the development process the specifier will be replaced by the representer whose task will be to use S to gradually build its lp-representation. He will start with building lp-representations $\Pi_{s_1}, \Pi_{s_2}, \ldots, \Pi_{s_n}$ of specifications $s_1, s_2, \ldots, s_n$. After this is done, he will address a task of finding an lp-representation Π_0 of the specification $s_0 = m(s_2, \ldots, s_n)$. One way to accomplish this is to find a mapping α_m from lp-representations to lp-representations such that $\alpha_m(\Pi_{s_2}, \ldots, \Pi_{s_n})$ represents $s_0 = m(s_2, \ldots, s_n)$. We will call such α_m a *realization* of m. Complexity of such mapping can serve as a good measure of the degree of tolerance of representations $(\Pi_{s_2}, \ldots, \Pi_{s_n})$ w.r.t. m. To complete the process and obtain representation of S the representer will apply realization α_k of k to (Π_{s_1}, Π_{s_0}).

We hope that the representer will be assisted in his task by mathematical results describing constructions of realizations of common specification constructors and giving sufficient conditions guaranteeing their correctness. We call such results *realization theorems*. Theorem 1 from section 5 and Theorem

2 from section 6 provide examples of such results. Another example, dealing with a constructor which takes as an input a specification S containing the closed world assumption for its predicates and returns a new specification obtained from S by removing this assumption, can be found in [5].

In the process of his work the representer will, of course, require frequent communication with the specifier. The existence of extensions of Prolog capable of answering queries about knowledge represented by lp-functions allows rapid prototyping and facilitates this communication. Ideally, after certain number of iterations both will agree on the adequacy of the representation of the original knowledge. Though correctness of the final functional specification does not lend itself to mathematical treatment, correctness of the corresponding representation can be stated and proven mathematically. Finally, an implementor, the main character in the last stage of the system development, will be confronted with the task of transferring declarative logic programs into their efficient executable counterparts. An ongoing work on optimization and generalization of logic programming systems together with the investigation of terminating properties of logic programs should be a substantial help during this stage.

The paper is organized as follows: Sections 2 and 3 contain definitions of functional specifications and lp-functions as well as examples which illustrate the elaboration process involved in building more complicated specifications from the simpler ones. Sections 4 and 5 investigate two important modifications of specifications, called incremental extension and simple input extension. Sufficient conditions guaranteeing tolerance w.r.t. such modifications are given by Theorems 1 and 2. In section 6 we introduce some results on conservative extension property of logic programs and use them in section 7 to prove Theorems 1 and 2. To make the paper self - contained in Appendix we introduce basic information about general and extended logic programs.

2 Functional Specifications

We will start with some basic terminology. Let $\mathcal{L}$ be a first-order language over the alphabet Ω. Formulas of the form $p(\overline{t})$ and $\neg p(\overline{t})$, where p is a predicate symbol and $\overline{t}$ is a vector of terms, are called (positive and negative) literals. Positive literals are often called atoms. Ground literals are literals

not containing variables. For any set P of predicate symbols from Ω, $lit(P)$ ($atoms(P)$) denotes the set of all ground literals (atoms) of $\mathcal{L}$ formed with symbols from P. S_P denotes the collection of all consistent subsets of $lit(P)$. For any literal l from $lit(P)$, $\overline{l}$ is $\neg p(\overline{t})$ if l is an atom $p(\overline{t})$, and $p(\overline{t})$ if l is $\neg p(\overline{t})$. For any set of literals X by $\overline{X}$ we will denote the set of all $\overline{l}$ such that $l \in X$.

In many situations specifications can be viewed as descriptions of relations (denoted by predicate symbols from some set Q) in terms of other relations (denoted by predicate symbols from, say, P). Since our knowledge about relations (or the relations themselves) can be incomplete we will represent them by sets of literals.

Partial description of Q in terms of P can be given by any condition on the corresponding sets of literals but, in this paper we concentrate on a special class of descriptions defined by functions. This view leads to the following:

Definition 1. By a *functional specification* we mean a four tuple $< F, P, Q, \Omega >$ where P and Q are sets of predicate symbols from Ω and F is a (possibly partial) function from S_P into S_Q. The domain of F will be denoted by $Dom(F)$.

Example 1. We are given a collection of disjoint classes of birds, such as eagles, canaries, penguins, etc.. (called a catalog) together with lists of birds belonging to each class and the list of unclassified birds. We assume that the lists contain correct but possibly incomplete information and can be frequently expanded.

Our task is to find a representation of this information capable of answering queries on classification of particular birds.

As most specifications heavily relying on natural language the one above is somewhat ambiguous. Below we describe its refinement (formal specification) which will contain the description of the language of discourse, relations involved, and the corresponding function.

Since our domain consists of two types of objects, classes of birds and individual birds, we assume that our alphabet Ω contains two types of constants (i.e. names of objects), $c_0, \ldots, c_n$ for classes and $n_0, \ldots, n_k$ for individual birds. We will also need two binary predicate symbols is and $subclass$ where $is(n, c)$ intuitively means "bird n belongs to class c" and $subclass(c_1, c_2)$ states that

"class c_1 is a subclass of class c_2". In anticipation of future extensions we introduce constant b for the class of all birds.

The catalog will be represented by a set of atoms of the form $subclass(c_i, b)$. The list of birds belonging to a class c will be described by a collection of atoms of the form $is(n_i, c)$.

Our specification can now be viewed as a function that given as an input the catalog and the list of birds returns information about birds membership in classes. More precisely our specification is a partial function B from S_P to S_Q where $P = \{is, subclass\}$ and $Q = \{is\}$. To fully define this function we have to describe its domain (i.e. valid inputs) and specify how its values are to be computed.

- According to our informal specifications classes are to be disjoint. This implies that a set X of literals from S_P belongs to the $Dom(B)$ if $X \subset atoms(P)$ and for any bird n there is at most one subclass c from the catalog such that $is(n, c) \in X$.

- The value of the function B on the given input X consists of all facts about birds membership in classes that can be deduced from the information given in the input. All such facts can be divided into three categories:

  1. facts about birds membership in classes given explicitly in the input
  2. facts about the membership of any member of any class in the class b of all birds
  3. facts about non - membership of any bird that belongs to one of the classes in any other class.

  This implies that for any set $X \in Dom(B)$, we have $B(X) = (X \cap lit(Q)) \cup I_1 \cup I_2$ where $I_1 = \{is(n, b) : is(n, c) \in X)$ for some class $c\}$ and $I_2 = \{\neg is(n, c_i) : is(n, c_j) \in X$ for some $i \neq j\}$.

Notice that there is no name for the class of unidentified birds in the language. Instead a bird n is considered to be undefined in input $X \in Dom(B)$ if $is(n, b) \in X$ and there is no c from the catalog such that $is(n, c) \in X$.[3]

[3]This allows us to expand the input database X by a new information, say, $is(n, c)$ without removing anything from X.

Specification defines a query language consisting of literals formed by is. Answer to a ground query q w.r.t. input database X is *yes* if $q \in B(X)$, *no* if $\overline{q} \in B(X)$ and *unknown* otherwise. If a query q contains a variable then answer to q is a list of terms $\overline{t}$ such that $q(\overline{t})$ is in $B(X)$.

Example 2. Assume now that in addition to information from Example 1 we want to represent a default statement *birds normally fly* and that we are given a list $c_{i_1}, \ldots, c_{i_k}$ of classes of birds from the catalog that do not fly. Our representation should now not only be able to classify individual birds but also to make conclusions about their flying abilities.

To refine our specification B to include this information we expand alphabet Ω by new predicate symbols ab and fly. The first relation is defined on elements of a catalog and used to represent exceptions to the default. (Abbreviation ab stands for *abnormal* or *exceptional*). It will be added to the input language P while fly will be added to the output language Q. To provide a formal specification that corresponds to the above information we define a function B_1 whose domain is like that of the function B from Example 1 but contains also information about some classes being exceptional in the sense that their members are non-flying birds. This implies that $X_1 \in Dom(B_1)$ iff $X_1 = X \cup Y$ where $X \in Dom(B)$ and $Y \subset atoms(ab)$.

The value of the function B_1 on the given input X should consist of all facts about birds membership in classes and their ability to fly that can be deduced from the information given in the input. All such facts can be divided into three categories:

1. facts about birds membership in classes that are deduced from the information supplied in the input in the same way it was done in Example 1

2. facts about flying ability of those birds that do not belong to any class that is known to be exceptional with respect to flying

3. facts about inability to fly for birds that belong to at least one class that is atypical with respect to flying.

This implies that for any $X_1 \in Dom(B_1)$ we have $B_1(X_1) = B(X) \cup F(X_1) \cup \overline{F}(X_1)$ where $F(X_1) = \{fly(n) : is(n, b) \in B(X)$ and for any class c if

$ab(c) \in Y$ then $\neg is(n,c) \in B(X)\}$ and $\overline{F}(X_1) = \{\neg fly(n)$: there is class c such that $is(n,c) \in B(X)$ and $ab(c) \in Y\}$.

It is worth noticing that our new specification B_1 is defined using the original specification B. Studying various specification constructors which allow us to construct new specifications from the existing once in a systematic way is one of the main topics of our paper. First however we need to discuss our choice of representational language.

3 Representing specifications by lp-functions

In this section we briefly describe the language we propose to use for representing knowledge and give several examples of its use. The language is based on the notion of an extended logic program from [12]. Let us recall the necessary definitions. More information can be found in the Appendix.

Let $\mathcal{L}$ be a first-order language over an alphabet Ω. An extended logic program Π in $\mathcal{L}$ is a collection of rules of the form:

$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$

where l's are literals of $\mathcal{L}$ and not stands for negation as failure ([7]). Rules with variables are viewed as schemata representing the sets of their ground instantiations. The entailment relation between extended logic programs and ground literals of $\mathcal{L}$ is based on the answer set semantics of [11], according to which $\Pi \models l$ iff l belongs to all answer sets of Π. (For a definition of answer sets see the Appendix). By $head(\Pi)$ we will denote the set of all literals that belong to conclusions of the rules from Π while by $body(\Pi)$ we will denote literals that belong to premises of those rules.

Applicability of the language specified above and of its extensions to knowledge representation has been extensively studied by various researchers (for a survey see [4]). To reflect the functional character of the specifications that we are planning to represent we adopt the approach suggested in [5] which associates logic programs with functions from sets of literals to sets of literals. The approach is similar to that used in Datalog, with some additional care taken to account for the existence of inconsistent programs, possible multiplicity of answer sets, and the absence of the closed world assumption. This view led to the following definition:

Definition 2. By an *lp-function* we mean a four tuple $< \Pi, P, Q, \Omega >$ where Π is a logic program in language $\mathcal{L}$, and P and Q are sets of predicate symbols from its alphabet Ω called input and output predicates respectively.

Whenever possible we identify lp-function $< \Pi, P, Q, \Omega >$ with Π. S_P and S_Q will be denoted by $inp(\Pi)$ and $out(\Pi)$ respectively.

Now we will define the *domain* ($Dom(\Pi)$) and the *value* ($\Pi(X)$) of the lp-function Π.

Let $X \in inp(\Pi)$. Then

- $X \in Dom(\Pi)$ iff $\Pi \cup X$ has a consistent answer set
- $\Pi(X) = \{l : l \in lit(Q) \text{ and } \Pi \cup X \models l\}$.

The next definition describes the notion of the representation of functional specification by an lp-function.

Definition 3. We will say that an lp-function $\Pi =< \Pi, P, Q, \Omega >$ *represents* a functional specification $F =< F, P, Q, \Omega >$ if

- $Dom(F) \subseteq Dom(\Pi)$
- For any $X \in Dom(F)$, $F(X) = \Pi(X)$.

Example 3. Let B be the specification from Example 1. Consider the following lp-function Π_B:

$$\left.\begin{array}{ll} r1. & is(x,b) \leftarrow subclass(c,b), is(x,c) \\ r2. & \neg is(x,c_i) \leftarrow subclass(c_j,b), is(x,c_j), i \neq j \end{array}\right\} \Pi_B$$

with alphabet Ω from Example 1, $P = \{is, subclass\}$ and $Q = \{is\}$.

To better understand this representation let us consider the input

$X = \{subclass(c_0,b), subclass(c_1,b), is(n_1,c_1), is(n_2,b)\}$.

Then it is easy to see that

$B(X) = \{is(n_1,c_1), is(n_1,b), is(n_2,b), \neg is(n_1,c_0)\}$

and hence answers to the queries $is(n_1,c_1)$, $is(n_1,c_0)$, and $is(n_2,c_1)$ are *yes*, *no* and *unknown* respectively, which corresponds to our specification. It

follows immediately from the basic properties of extended logic programs that the lp-function Π_B represents specification B.

Example 4. Now let B_1 be the specification from Example 2 and consider the following lp-function Π_{B_1}:

$$\left.\begin{array}{l} r1.\ \ is(x,b) \leftarrow subclass(c,b), is(x,c) \\ r2.\ \ \neg is(x,c_i) \leftarrow subclass(c_j,b), is(x,c_j), i \neq j \\ r3.\ \ may\_be\_in(x,c) \leftarrow not\ \neg is(x,c) \\ r4.\ \ may\_be\_ab(x) \leftarrow ab(c), may\_be\_in(x,c) \\ r5.\ \ fly(x) \leftarrow is(x,b), not\ may\_be\_ab(x) \\ r6.\ \ \neg fly(x) \leftarrow ab(c), is(x,c) \end{array}\right\} \Pi_{B_1}$$

with alphabet Ω from Example 2, $P = \{is, subclass, ab\}$ and $Q = \{is, fly\}$.

Consider an input

$X = \{subclass(c_0,b), subclass(c_1,b), ab(c_0), is(n_0,c_0), is(n_1,c_1), is(n_2,b)\}$

then

$\Pi_{B_1}(X) = \{is(n_0,c_0), is(n_1,c_1), is(n_0,b), is(n_1,b), is(n_2,b),$
$\neg is(n_0,c_1), \neg is(n_1,c_0), fly(n_1), \neg fly(n_0)\}$

and hence answers to the queries $fly(n_1), fly(n_0), fly(n_2)$ are *yes*, *no* and *unknown* respectively.

As in the previous example the lp-function Π_{B_1} represents specification B_1.

It is worth noting that lp-functions Π_B and Π_{B_1} give us not only provably correct but also (rather efficiently) executable representations of specifications B and B_1.

In the next example we illustrate the use of an important, though simple, family of specification constructors, called *simple input extensions*.

Example 5. Let us consider the specification B_2 obtained from specification B_1 in Example 2 by adding to the possible input a list of facts containing information about the flying ability of individual birds, i.e. a consistent collection of literals from $lit(fly)$. We will say that B_2 is a simple input extension of the specification B_1 w.r.t. predicate fly. Somewhat more precisely,

- $X \in Dom(B_2)$ iff $X = X_1 \cup X_2$ where $X_1 \in Dom(B_1)$ and X_2 is a consistent set of literals from $lit(fly)$ such that for every bird n and class c either $\{is(n,c), ab(c)\} \not\subset X_1$ or $fly(n) \notin X_2$.

- For any $X \in Dom(B_2)$ and for any $l \in lit(\{is, fly\})$ we have $l \in B_2(X)$ iff $l \in X_2 \cup (B_1(X_1) \setminus \overline{X_2})$.

The "distance" between specifications B_1 and B_2 seems to be rather small, which raises a natural question: is it possible to find an lp-function Π_{B_2} representing B_2 which is "close" to Π_{B_1}?

It is easy to see that the first (natural) attempt to construct such Π_{B_2} by using the same program Π_{B_1} and extending the domain under consideration by allowing in it literals formed by predicate fly fails. (Check, for instance, that $\{is(t,b), \neg fly(t)\} \in Dom(B_2)$ but $\{is(t,b), \neg fly(t)\} \notin Dom(\Pi_{B_2})$.)

A more sophisticated transformation α is obtained by replacing $r5$ and $r6$ in Π_{B_1} by the rules:

$$\left.\begin{array}{l} r5'.\ \ fly(x) \leftarrow is(x,b), not\ may\_be\_ab(x), not\ \neg fly(x) \\ r6'.\ \ \neg fly(x) \leftarrow ab(c), is(x,c), not\ \neg fly(x) \end{array}\right\}$$

As a result we obtain a new lp-function

$\Pi_{B_2} =< \Pi_{B_2}, \{is, subclass, ab, fly\}, \{is, fly\}, \Omega >$

which represents specification B_2.

It is interesting to notice that lp-function

$\Pi^*_{B_2} =< \Pi_{B_2}, \{is, subclass, ab\}, \{is, fly\}, \Omega >$

provides an alternative representation of specification B_1. Which one is preferable? Π_{B_1} is slightly simpler and allows more efficient query-answering while $\Pi^*_{B_1}$ is more tolerant with respect to possible modifications. It will allow the specifier to go from B_1 to B_2 without changing the representation. The choice will be determined by the specifier based on the likelihood of the possible expansion of the domain of B_1 to its values.

Next example demonstrates another specification constructor commonly used in building specifications. It is called *incremental extension* and denoted by $\circ$.

Example 6. Consider specification B_2 from Example 5 together with the following simple specification C informally described by the rule:
"If a bird does not fly, its cage does not need the top. Otherwise, (i.e. when the bird flies or when its flying abilities are not known) the top is necessary".

The language of C consists of the language of B_2 together with a new predicate *top*, where $top(n)$ stands for "a cage for bird n needs the top". The specification can be defined as follows:

$C =< C, \{fly, is\}, \{top\}, \Omega \cup \{top\} >$

where for any consistent set X from $lit(\{fly, is\})$

- $\neg top(n) \in C(X)$ iff $\{is(n, b), \neg fly(n)\} \subseteq X$
- $top(n) \in C(X)$ iff $is(n, b) \in X$ and $\neg fly(n) \notin X$

It is easy to check that C is represented by the following lp-function:

$$\left.\begin{array}{ll} r1. & top(x) \leftarrow is(x, b), not \ \ \neg fly(x) \\ r2. & \neg top(x) \leftarrow is(x, b), \neg fly(x) \end{array}\right\} \Pi_C$$

The specifications B_2 and C can be used to construct a new specification $C \circ B_2$ which takes sets from the domain of B_2 as inputs and uses B_2 and C to compute the values from $lit(\{is, fly, top\})$ as follows:

$C \circ B_2(X) = B_2(X) \cup C(B_2(X))$

This new specification is called incremental extension of B_2 by C.

Again we are interested in finding lp-function representation for $C \circ B_2$ as a natural combination of lp-functions Π_{B_2} and Π_C. The lp-function $\Pi_{B_2} \cup \Pi_C$ seems to be a natural candidate and it is easy to see that it works.

The above constructions and observations were discussed for fairly simple examples. Do they allow generalization to broader classes of specifications and their representations? What are the typical modifications of specifications and what is their relationship with the corresponding representations? Can complex specifications be constructed from some collections of "basic" specifications with the help of a relatively small number of operations? If the answer to the last question is positive can we use the structure of a specification to automatically build its representation from the representations of its basic parts? To draw attention to these questions and to outline a possible approach to finding the answers is the main goal of this paper.

In the next two sections we provide formal definitions of input and incremental extensions as specification constructors and state theorems giving sufficient conditions for existence of their realizations.

4 Incremental Extensions of Functional Specifications

We first introduce a binary specification constructor we call incremental extension.

Definition 4. Let $< F, P, Q, \Omega >$ and $< G, Q, R, \Omega >$ be functional specifications such that:

- $R \cap Q = \emptyset$
- for any $X \in Dom(F)$, $F(X) \in Dom(G)$.

Functional specification $G \circ F =< G \circ F, P, Q \cup R, \Omega >$ with the properties:

- $Dom(G \circ F) = Dom(F)$
- $G \circ F(X) = F(X) \cup G(F(X))$

will be called an *incremental extension* of F by G.

It is easy to see that functional specification discussed in Example 6 is an incremental extension of specifications B_2 and C. As we have noted before representation of $C \circ B_2$ can be obtained by combining the rules from B_2 and C. Unfortunately this is not always the case as illustrated by the following:

Example 7. Consider an alphabet Ω consisting of an object constant a and predicate symbols p, q, r. Let F and G be two specifications defined as follows

$F =< F, \emptyset, \{p, q\}, \Omega >$ and

$G =< G, \{p, q\}, \{r\}, \Omega >$ where

$Dom(F) = \{\emptyset\}$, $Dom(G) = \{\emptyset, \{q(a)\}\}$ and

$F(\emptyset) = \emptyset$ and $G(\emptyset) = G(\{q(a)\}) = \{r(a)\}$.

It is easy to see that F and G can be represented by lp-functions

$\Pi_F = \{p(a) \leftarrow not\ q(a); q(a) \leftarrow not\ p(a)\}$ and

$\Pi_G = \{r(a) \leftarrow; \neg r(a) \leftarrow p(a)\}$ and that

$\Pi_G \cup \Pi_F$ does not represent an incremental extension of F by G. Indeed,

$G \circ F(\emptyset) = \{r(a)\}$ while $\Pi_G \cup \Pi_F(\emptyset) = \{r(a), q(a)\}$

This is of course not surprising. The difficulty is caused by the multiplicity of answer sets of Π_F. One can expect properties of a program to depend on the collection of its answer sets and not on the collection of entailed literals only. Even though $\Pi_{G \circ F}$ is consistent with the intersection of answer sets of Π_F it is not consistent with one of them, namely $\{p(a)\}$ which causes the loss of incrementality. In some cases $\Pi_{G \circ F}$ will not even be consistent with the intersection of answer sets of Π_F. This observation leads us to the following definition.

Definition 5. We will say that an lp-function $< \Pi, P, Q, \Omega >$ is *categorical* if for any $X \in Dom(\Pi)$ and any answer sets A_1 and A_2 of $\Pi \cup X$ we have

- $A_1 \cap lit(Q) = A_2 \cap lit(Q)$

In what follows by $L(\Pi)$ we will denote the set of all literals that either appear in program Π or belong to $lit(P \cup Q)$.

The following condition will guarantee correctness of our construction of $\Pi_{G \circ F}$ w.r.t. incremental extension.

Definition 6. We will say that lp-functions $< \Pi_F, P, Q, \Omega >$ and $< \Pi_G, Q, R, \Omega >$ are *upward compatible* if

- $head(\Pi_G) \cap L(\Pi_F) = \emptyset$
- $L(\Pi_G) \cap L(\Pi_F) \subseteq lit(Q)$

We can now state the realization theorem for incremental extensions.

Theorem 1. Let F and G be functional specifications represented by categorical lp-functions $< \Pi_F, P, Q, \Omega >$ and $< \Pi_G, Q, R, \Omega >$ respectively, and let $G \circ F$ be the incremental extension of F by G. If Π_F is upward compatible with Π_G then lp-function $\Pi_{G \circ F} =< \Pi_G \cup \Pi_F, P, Q \cup R, \Omega >$ represents $G \circ F$.

Example 8. It is easy to see that lp-functions Π_{B_2} and Π_C from Example 6 satisfy the conditions of Theorem 1 which explains correctness of the corresponding representation of $C \circ B_2$.

5 Simple Input Extensions of Functional Specifications

We now describe a class of unary specification constructors we call simple input extensions.

Definition 7. Let $F =< F, P, Q, \Omega >$ be a functional specification. A specification $F^* =< F^*, P \cup Q, Q, \Omega >$ such that

- $Dom(F) \subset Dom(F^*)$
- for any $X \in Dom(F^*)$, $X_P = X \cap lit(P) \in Dom(F)$
- for any $X \in Dom(F^*)$ and for any literal $l \in lit(Q)$, $l \in F^*(X)$ iff $l \in ((X_Q \cup F(X_P)) \setminus \overline{X}_Q)$ where $X_Q = X \setminus X_P$

is called a *simple input extension* of F.

It is easy to see that specification B_2 from Example 5 is a simple input extension of specification B_1.

In what follows we present a realization theorem which provides a general method of constructing lp-function representations of simple input extensions from the lp-function representations of the original functional specifications and which implies correctness of the construction of Π_{B_2} in Example 5.

Let $F =< F, P, Q, \Omega >$ be a functional specification represented by lp-function Π and let $F^* =< F^*, P \cup Q, Q, \Omega >$ be a simple input extension of specification F. We are interested in constructing an lp-function Π^* representing specification F^*. The Example 5 from the earlier section suggests a natural candidate for Π^* that can be defined as follows:

Definition 8. Let Π be a program and Q be a collection of predicates. The result of replacing every rule

$l_0 \leftarrow l_1, \ldots, l_m, not \;\; l_{m+1}, \ldots, not \;\; l_n$

of Π with $l_0 \in lit(Q)$ by the rule

$l_0 \leftarrow l_1, \ldots, l_m, not \;\; l_{m+1}, \ldots, not \;\; l_n, not \;\; \overline{l}_0$

is called a *guarded version* of Π (w.r.t. Q) and denoted by $\hat{\Pi}_Q$. (Whenever possible, the index Q will be omitted.)

Unfortunately, the guarded version $\hat{\Pi}$ of Π does not always work as a representation of F^* as illustrated by the following:

Example 9. Consider an alphabet Ω consisting of object constants a, b and predicate symbols p, q, r. Let functional specification F be defined as follows: $F =< F, \{p\}, \{q\}, \Omega >$ where $Dom(F) = \{\{p(a)\}\}$ and $F(\{p(a)\}) = \{\neg q(a), q(b)\}$ and let Π be the following lp-function:

$q(X) \leftarrow r(X)$

$\neg q(a) \leftarrow$

$r(a) \leftarrow not \;\; r(b), p(a)$

$r(b) \leftarrow not \;\; r(a), p(a)$

It is easy to see that $\Pi(\{p(a)\}) = \{\neg q(a), q(b)\}$ and hence it represents F while lp-function $\hat{\Pi}$

$q(X) \leftarrow not \;\; \neg q(X), r(X)$

$\neg q(a) \leftarrow not \;\; q(a)$

$r(a) \leftarrow not \;\; r(b), p(a)$

$r(b) \leftarrow not \;\; r(a), p(a)$

has a property $\hat{\Pi}(\{p(a\}) = \emptyset$ and represents no simple extension of F.

We will now introduce a notion of a *well structured* lp-function and show that for any specification F represented by such lp-function its guarded version represents all simple extensions of F.

Definition 9. Let $\Pi =< \Pi, P, Q, \Omega >$ be an lp-function. We will say that Π is *well structured* if there are programs Π_1 and Π_2 (called components of Π) such that

- $\Pi = \Pi_1 \cup \Pi_2$
- $(lit(\Pi_1) \cup body(\Pi_2)) \cap lit(Q \setminus P) = \emptyset$
- $head(\Pi_2) \subset lit(Q \setminus P)$
- for any set $X \in Dom(\Pi)$ and any answer set A of $\Pi_1 \cup X$, $A \cup \Pi_2$ is consistent.

We can now state realization theorem for simple extensions.

Theorem 2. Let F be a functional specification represented by a well structured lp-function Π with components Π_1 and Π_2. Then the lp-function $\Pi^* =< \Pi_1 \cup \hat{\Pi_2}, P \cup Q, Q, \Omega >$ represents every simple extension F^* of F.

It is easy to see that lp-function Π_{B_1} from Example 4 is well structured and thus functional specification B_1 satisfies the conditions of Theorem 2 which explains correctness of the corresponding representation Π_{B_2} of the specification B_2.

6 Conservative Extension Property of Logic Programs

Theorem 1 is closely related to the so called conservative extension property of logic programs. If we extend a program Π by rules whose heads do not occur in Π (which can be viewed as expanding Π by definition of new predicates), then, typically, we do not expect to see any new consequences of the program among the literals occurring in Π. This conservative extension property (called "the weak principle of stratification" by Schlipf [23] and "relevance" by Dix [8]) is not valid even for general logic programs under stable model semantics without additional restrictions. Consider, for instance, a program $\Pi = \{p \leftarrow not\ q;\ q \leftarrow not\ p;\}$ and the definition $D = \{r \leftarrow not\ r, q;\ r \leftarrow p\}$ of r. It is easy to see that $\Pi \cup D$ entails p while Π does not. The question of discovering conditions under which the conservative extension property holds was addressed in [13]. Some of these results were generalized in [15]. To prove Theorem 1 we give here another (stronger) sufficient condition of conservativeness. [4]

Recall that $head(\Pi)$ stands for the set of all literals that belong to conclusions of the rules from Π. By $neg(\Pi)$ we will denote the set of all literals whose negation as failure appears in Π while $lit(\Pi)$ is the set of all literals formed by predicates from Π.

Theorem 3. Let D and Π be logic programs such that:

[4] A slightly weaker version will suffice for this goal but we believe that Theorem 3 is of independent interest. In particular, it is useful for proving the counterpart of Theorem 1 for non-functional specifications.

- $head(D) \cap lit(\Pi) = \emptyset$
- for any consistent answer set A of Π program $D \cup (A \cap lit(D))$ is consistent.

Then we have:

- If A_1 is a consistent answer set of Π then there is a consistent set $A_2 \subseteq head(D)$ such that $A = A_1 \cup A_2$ is an answer set of the program $D \cup \Pi$.
- If A is a consistent answer set of the program $D \cup \Pi$ then set $A_1 = A \cap lit(\Pi)$ is a consistent answer set of the program Π.

Corollary 1. Under the same conditions as in Theorem 3 program $D \cup \Pi$ is a conservative extension of the program Π.

Before proving Theorem 3 we will recall an important property of logic programs called Splitting Set Theorem [15].

A *splitting set* for a logic program Π is any set U of literals such that, for every rule $r \in \Pi$, if $head(r) \cap U \neq \emptyset$ then $lit(r) \subseteq U$.

Given program Π and its splitting set U the set of rules $r \in \Pi$ for which $lit(r) \subseteq U$ is called the *bottom* of Π w.r.t. U and denoted by $b(\Pi, U)$. The remaining set of rules in Π will be called the *top* of Π w.r.t. U and denoted by $t(\Pi, U)$.

Given program Π and set of literals S we will denote by $red(\Pi, S)$ a new program obtained from program Π by removing from it all rules whose premises do not hold in S and by removing from the remaining rules in Π all expressions of the form l or *not* l where $l \in S$.

Theorem 4. (Splitting Set Theorem [15]). Let U be a splitting set for a program Π. A set A of literals is a consistent answer set for Π if and only if $A = A_1 \cup A_2$ where A_1 is a consistent answer set for the $b(\Pi, U)$ and A_2 is a consistent answer set of $red(t(\Pi, U), A_1)$.

Corollary 2. ([15]). Let Π be a logic program and let X be a set of literals such that $X \cap head(\Pi) = \emptyset$. A set A of literals is a consistent answer set for $\Pi \cup X$ if and only if X is consistent and $A = A_1 \cup X$ where A_1 is a consistent answer set for the $red(\Pi, X)$.

The following simple consequence of the Splitting Set Theorem will be used in the prove of Theorem 2.

Observation 1. If Π is a well structured lp-function with components Π_1 and Π_2 then for every X from the $Dom(\Pi)$ a set of literals A is a consistent answer set of $\Pi \cup X$ iff $A = A_1 \cup A_2$ where A_1 is a consistent answer set of the program $\Pi_1 \cup X$ and $A_2 = red(\Pi_2, A_1) \subset lit(Q \setminus P)$.

Now we are ready for the proof of the Theorem 3.

Proof. Since $head(D) \cap lit(\Pi) = \emptyset$ therefore $lit(\Pi)$ is a splitting set for the program $D \cup \Pi$ with $b(D \cup \Pi, lit(\Pi)) = \Pi$ and $t(D \cup \Pi, lit(\Pi)) = D$. In virtue of the Splitting Set Theorem set A is a consistent answer set of this program if and only if $A = A_1 \cup A_2$ where A_1 is a consistent answer set of the bottom part of the program (i.e. Π) and A_2 is a consistent answer set of the reduct of the top of the program (i.e. D) w.r.t. A_1. To complete the proof of the theorem we have to show that $red(D, A_1)$ is consistent for every consistent answer set A_1 of Π. Let us first observe that $red(D, A_1) = red(D, A_1')$ where $A_1' = A_1 \cap lit(D)$. Using assumptions of the Theorem we can conclude that program $D \cup A_1'$ has a consistent answer set. Since $A_1' \subset lit(\Pi)$ we have $A_1' \cap head(D) = \emptyset$ and thus in virtue of the Corollary 2 above this answer set is a union of A_1' and an answer set of $red(D, A_1')$. This shows that $red(D, A_1)$ is consistent and completes the proof of the Theorem.

Proposition 1. The sufficient condition for the conservative extension property given in Corollary 1 is a generalization of the condition provided by Proposition 1 in [15]. More precisely if Π is a logic program and C is a consistent set of literals that do not occur in Π and whose complements also do not occur in Π and if D is a nondisjunctive program such, that for every rule $r \in D$, $head(r) \subseteq C$ and $neg(r) \subseteq lit(\Pi)$ then $D \cup \Pi$ is a conservative extension of Π.

Proof. To prove Proposition 1 it is enough to show that programs Π and D satisfy the assumptions of Theorem 3. It follows from the assumptions of the Proposition that $head(D) \cap lit(\Pi) = \emptyset$. To prove that for any consistent answer set A of Π program $D \cup (A \cap lit(D))$ is consistent let us notice that program D and set of literals $A \cap lit(D)$ satisfy assumptions of Corollary 2 and thus $D \cup (A \cap lit(D))$ is consistent if and only if program $red(D, A \cap lit(D))$ is consistent. Since for every rule $r \in D$ we have $neg(r) \subseteq lit(\Pi)$, program

$red(D, A \cap lit(D))$ contains no negation as failure and since set C is consistent it has a consistent answer set.

The next example together with Proposition 1 shows that Corollary 1 is stronger than Proposition 1 from [15].

Example 10. Let Π be a program consisting of rules $r5, r6, \ldots$ of the program Π_{B_1} from Example 3 and let $D = \Pi_{B_1} \setminus \Pi$. It is easy to see that $D \cup X$ is consistent for any consistent set $X \subset lit(b, w, c_1, \ldots, c_n)$ and hence, by Corollary 1, Π_{B_1} is a conservative extension of Π. At the same time Proposition 1 ([15]) is not applicable to these Π and D.

7 Proofs of Theorems 1 and 2

We will now use the results of the previous section to prove realization theorems for incremental and simple extensions.

We will begin with the following:

Proposition 2. If lp-function $< \Pi, P, Q, \Omega >$ is categorical then for any $X \in Dom(\Pi)$ and any answer set A of $\Pi \cup X$ we have:

- $\Pi(X) = A \cap lit(Q)$

Proof. By definition $\Pi(X)$ consists of all elements of $lit(Q)$ which belong to all answer sets of the program $\Pi \cup X$. In case of categorical lp-functions intersections of all answer sets of $\Pi \cup X$ with $lit(Q)$ are identical.

We will also need the following:

Lemma 1. Let F and G be functional specifications represented by categorical lp-functions $< \Pi_F, P, Q, \Omega >$ and $< \Pi_G, Q, R, \Omega >$ respectively, and let $G \circ F$ be the incremental extension of F by G. If Π_F is upward compatible with Π_G then for any $X \in Dom(\Pi_F)$ programs Π_G and $\Pi_F \cup X$ satisfy assumptions of Theorem 3.

Proof. The condition that $head(\Pi_G) \cap lit(\Pi_F \cup X) = \emptyset$ is guaranteed by upward compatibility of Π_F with Π_G. We have to show that for any consistent answer set A of $\Pi_F \cup X$ program $\Pi_G \cup (A \cap lit(\Pi_G))$ is consistent. From the assumption that $G \circ F$ is an incremental extension of F by G it follows that $F(X) \in Dom(G)$ and since Π_F and Π_G represent functional specifications of

F and G respectively we can conclude that $\Pi_F(X) \in Dom(G)$ and therefore $\Pi_G \cup \Pi_F(X)$ is consistent. Since Π_F is categorical using Proposition 2 we obtain $\Pi_F(X) = A \cap lit(Q)$ and thus program $\Pi_G \cup (A \cap lit(Q))$ is consistent. It is easy to see that set of literals $A \cap (lit(Q) \setminus lit(\Pi_G))$ is a splitting set for this program and therefore a consistent answer set of $\Pi_G \cup (A \cap lit(Q))$ is the union of $A \cap (lit(Q) \setminus lit(\Pi_G))$ and the consistent answer set of $\Pi_G \cup (A \cap (lit(\Pi_G) \cap lit(Q))$. Using again upward compatibility of Π_F with Π_G we have $A \cap (lit(\Pi_G) \cap lit(Q)) = A \cap lit(\Pi_G)$. This shows that for any consistent answer set A of $\Pi_F \cup X$ program $\Pi_G \cup (A \cap lit(\Pi_G))$ is consistent and completes the proof of the Lemma.

Now we are ready to prove Theorem 1.

Theorem 1. Let F and G be functional specifications represented by categorical lp-functions $< \Pi_F, P, Q, \Omega >$ and $< \Pi_G, Q, R, \Omega >$ respectively, and let $G \circ F$ be an incremental extension of F by G. If Π_F is upward compatible with Π_G then lp-function $\Pi_{G \circ F} =< \Pi_G \cup \Pi_F, P, Q \cup R, \Omega >$ represents $G \circ F$.

Proof. To show that $\Pi_{G \circ F}$ represents $G \circ F$ we have to show that:

- If $X \in Dom(G \circ F)$ then $X \in Dom(\Pi_{G \circ F})$.
- If $X \in Dom(G \circ F)$ then $G \circ F(X) = \Pi_{G \circ F}(X)$.

Let us first observe that by definition $Dom(G \circ F) = Dom(F)$. Since F is represented by lp-function Π_F we have $Dom(F) \subseteq Dom(\Pi_F)$ and thus if $X \in Dom(G \circ F)$ then $X \cup \Pi_F$ is consistent. On the other hand using definition of $\Pi_{G \circ F}$ we can conclude that $X \in Dom(\Pi_{G \circ F})$ iff $X \cup \Pi_F \cup \Pi_G$ is consistent. Therefore we have to show that for any consistent set $X \subset lit(P)$, if $X \cup \Pi_F$ is consistent then $X \cup \Pi_F \cup \Pi_G$ is consistent. In virtue of Lemma 1 programs Π_G and $\Pi_F \cup X$ satisfy assumptions of Theorem 3 and hence, by the first part of this theorem, $X \cup \Pi_F \cup \Pi_G$ is consistent and hence $X \in Dom(\Pi_{G \circ F})$.

Now let us observe that $G \circ F(X) = F(X) \cup G(F(X))$ and since Π_F and Π_G are representations of F and G respectively we have $G \circ F(X) = \Pi_F(X) \cup \Pi_G(\Pi_F(X))$. Let $X \in Dom(G \circ F)$ (i.e. $X \cup \Pi_F$ is consistent) and let A be any consistent answer set of $X \cup \Pi_F$. Using Proposition 2 w.r.t. Π_F we obtain $G \circ F(X) = (A \cap lit(Q)) \cup \Pi_G(A \cap lit(Q))$. Let B be any consistent answer set of the program $\Pi_G \cup (A \cap lit(Q))$. In virtue of Corollary 2 $B = (A \cap lit(Q)) \cup B_2$

where B_2 is an answer set of the program $red(\Pi_G, (A \cap lit(Q))$. Using again Proposition 2 (now w.r.t. Π_G) we can conclude that $G \circ F(X) = (A \cap lit(Q)) \cup (B_2 \cap lit(Q \cup R))$.

Let us now consider $\Pi_{G \circ F}(X)$. By definition $\Pi_{G \circ F}(X) = \{l \in lit(Q \cup R) : \Pi_F \cup \Pi_G \cup X \models l\}$ i.e. it is the set of all literals in $lit(Q \cup R)$ which belong to all consistent answer sets of the program $\Pi_F \cup \Pi_G \cup X$. In virtue of Lemma 1 programs $\Pi_F \cup X$ and Π_G satisfy assumptions of the Theorem 3 and thus every consistent answer set of the program $\Pi_F \cup \Pi_G \cup X$ has the form $A_1 \cup B_1$ where A_1 is an answer set of the program $\Pi_F \cup X$ and B_1 is an answer set of the $red(\Pi_G, A_1)$. This implies that literal l belongs to $\Pi_{G \circ F}(X)$ iff it belongs to all the sets of the form $(A_1 \cap lit(Q)) \cup (B_1 \cap lit(Q \cup R))$ where as before A_1 is an answer set of the program $\Pi_F \cup X$ and B_1 is an answer set of the $red(\Pi_G, A_1)$.

To complete the proof of the Theorem it is now enough to show that for any A_1 and B_1 above we have:

- $A_1 \cap lit(Q) = A \cap lit(Q)$
- $B_1 \cap lit(Q \cup R) = B_2 \cap lit(Q \cup R)$

where A and B_2 are the answer sets of $X \cup \Pi_F$ and $red(\Pi_G, (A \cap lit(Q))$ respectively, chosen above.

The first part is a consequence of the categoricity of lp-function Π_F. To prove the second part we observe that upward compatibility of Π_F with Π_G implies that $L(\Pi_F) \cap L(\Pi_G) \subseteq lit(Q)$ and therefore $red(\Pi_G, A_1) = red(\Pi_G, A_1 \cap lit(Q) = red(\Pi_G, A \cap lit(Q))$. Since B_1 is an answer set of the $red(\Pi_G, A_1) = red(\Pi_G, A \cap lit(Q))$ we can conclude that $A \cap lit(Q) \cup B_1$ is an answer set of the program $\Pi_G \cup (A \cap lit(Q))$ and thus using categoricity of Π_G we obtain $B_1 \cap lit(Q \cup R) = B_2 \cap lit(Q \cup R)$ which completes the proof of the Theorem.

Theorem 2. Let F be a functional specification represented by a well structured lp-function Π with components Π_1 and Π_2. Then the lp-function $\Pi^* =< \Pi_1 \cup \hat{\Pi}_2, P \cup Q, Q, \Omega >$ represents every simple extension F^* of F.

Proof. To prove that the lp-function Π^* represents every simple extension F^* of F we have to show that for every $X \in Dom(F^*)$ we have $F^*(X) = \Pi^*(X)$. Let $X_1 = X \cap lit(P)$ and $X_2 = X \setminus X_1$ then $\Pi^*(X) = \{l \in Q : l \in A$ for any consistent answer set A of $\Pi_1 \cup \hat{\Pi}_2 \cup X_1 \cup X_2\}$. It is easy to see

that $lit(\Pi) \setminus lit(head(\Pi_2))$ is a splitting set for $\Pi_1 \cup \hat{\Pi}_2 \cup X_1 \cup X_2$ and the bottom of the program is $\Pi_1 \cup X_1$. It implies that every consistent answer set A of $\Pi_1 \cup \hat{\Pi}_2 \cup X_1 \cup X_2$ has a form $A_1 \cup A_2$ where A_1 is a consistent answer set of $\Pi_1 \cup X_1$ and A_2 is a consistent answer set of $red(\hat{\Pi}_2 \cup X_2, A_1)$. Let us first observe that $red(\hat{\Pi}_2 \cup X_2, A_1) = red(\hat{\Pi}_2, A_1) \cup X_2$. On the other hand since lp-program Π represents functional specification F and since F^* is a simple extension of F we can conclude that $X_1 \in Dom(\Pi)$ and $F(X_1) = \Pi(X_1)$. It means that program $\Pi_1 \cup \Pi_2 \cup X_1$ is consistent and in virtue of the Observation 1 set of literals B is an answer set of this program if and only if $B = B_1 \cup B_2$ where B_1 is a consistent answer set of the program $\Pi_1 \cup X_1$ and $B_2 = red(\Pi_2, B_1)$. From this it follows that

- $F(X_1) = \{l \in lit(Q) : l \in B_1 \cup red(\Pi_2, B_1)$ for every consistent answer set B_1 of $\Pi_1 \cup X_1\}$.

Let B_1 be such an answer set. Since Π is well structured we can conclude that $red(\Pi_2, B_1)$ is a consistent set of literals. This implies that $red(\Pi_2, B_1)$ is the unique answer set of the program $red(\hat{\Pi}_2, B_1)$ and thus $B_2 = (X_2 \cup red(\Pi_2, B_1)) \setminus \overline{X_2}$ is the unique answer set of the program $red(\hat{\Pi}_2 \cup X_2, B_1)$.

Let us now assume that $l \in lit(Q)$. From the above considerations it follows that $l \in \Pi^*(X)$ iff $l \in (B_1 \cup X_2 \cup red(\Pi_2, B_1)) \setminus \overline{X_2}$ for every consistent answer set B_1 of $\Pi_1 \cup X_1$. It is easy to see that this is equivalent to $l \in (F(X_1) \cup X_2) \setminus \overline{X_2} = F^*(X)$ which proves that for every $l \in Q$ and for every $X \in Dom(F^*)$ we have $l \in F^*(X)$ iff $l \in \Pi^*(X)$ and thus completes the proof of the Theorem.

8 Conclusions

To make logic programming a viable paradigm for software engineering one should understand each step in the following diagram describing a standard software development process:

Specification $\Rightarrow$ Representation $\Rightarrow$ Implementation

In the last decade most of the theoretical work in logic programming concentrated on the last two steps, especially on the development of declarative logic programming as a language of knowledge representation and on improving efficiency and termination properties of various logic programming

implementations. The goal of this paper is to contribute to the better understanding of the first step of the diagram. To this goal we

- introduced the notion of specification independent of particulars of the language used for its description
- suggested the use of lp-functions as representations of functional specifications
- defined the notions of specification constructor and its realization.

We used these definitions to demonstrate, by way of examples, that correctness and quality of representation of a specification S depends on:

- the type of input to the corresponding knowledge base
- the type of allowed queries
- the likelihood of possible modifications of the system.

The first two dependencies are reflected in the definition of lp-function. To study the third dependency we suggest to consider various typical modifications and carefully investigate their realizations. Two such modifications, incremental extension and simple extension, were investigated and conditions guaranteeing elaboration tolerance of lp-representations w.r.t. these modifications were presented in Theorems 1 and 2. (A sufficient condition allowing to prove the conservative extension property of logic programs was established as a side effect). There are of course other important modifications of functional specifications. One such modification - removing the closed world assumption [21] from the input predicates of specification S - was studied in [5] from the position similar to the one advocated here. In general, adding and removing assumptions about the domain of specification S, such as the closed world assumption, the unique name assumption, etc., seem to frequently occur in practice. We believe that the study of these, as well as the discovery and investigation of other frequent modifications, is an important topic for the further research. We are also interested in extending the proposed approach to the class of arbitrary (not necessarily functional) specifications.

9 Appendix: Extended Logic Programs

Extended logic programs provide a powerful tool for knowledge representation that allows direct representation of the incomplete knowledge.

Formally, by an extended logic program, Π, we mean a collection of rules of the form

$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n$

where the l's are literals, i.e., formulas of the form p or $\neg p$, where p is an atom and *not* denotes negation as failure. When all l's are atoms the program is called a *general* logic program.

The set of all literals in the language of Π will be denoted by $lit(\Pi)$.

The semantics of an extended logic program assigns to it a collection of its *answer sets* - sets of literals corresponding to beliefs which can be built by a rational reasoner on the basis of Π. We will say that literal l is *true* in an answer set S if $l \in S$ and that *not* l is *true* in S if $l \notin S$. We will say that Π's answer to a literal query q is *yes* if q is *true* in all answer sets of Π, *no* if $\bar{q}$ [5] is *true* in all answer sets of Π and *unknown* otherwise.

The answer set semantics of extended logic programs treats a rule with variables as shorthand for the set of its ground instances and thus it suffices to define answer sets for variable free programs.

To give a definition of answer sets of extended logic programs, let us first consider programs without negation as failure.

The *answer set* of the variable free program Π not containing *not* is the smallest (in the sense of set-theoretic inclusion) subset S of $lit(\Pi)$ such that

- for any rule $l_0 \leftarrow l_1, \ldots, l_m$ from Π, if $l_1, \ldots, l_m \in S$, then $l_0 \in S$
- if S contains a pair of complementary literals, then $S = lit(\Pi)$.

Obviously, every program Π that does not contain negation as failure has a unique answer set.

Definition 10. Let Π be an extended logic program without variables. For any set S of literals, let Π^S be the logic program obtained from Π by deleting

[5] For any literal l, the symbol $\bar{l}$ denotes the literal opposite in sign to l. i.e. for an atom a, if $l = \neg a$ then $\bar{l} = a$, and if $l = a$ then $\bar{l} = \neg a$.

- each rule that has a formula $not \;\; l$ in its body with $l \in S$
- all formulas of the form $not \;\; l$ in the bodies of the remaining rules.

Clearly, Π^S does not contain not , so that its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π.

Example 11. Assume that our language contains two object constants a and b and consider

$\Pi = \{p(X) \leftarrow not \;\; q(X), \quad q(a) \leftarrow\}$.

Let us show that a set $S = \{q(a), p(b)\}$ is an answer set of Π. By construction, $\Pi^S = \{p(b) \leftarrow, \;\; q(a) \leftarrow\}$ whose answer set is obviously equal to S. Later we will show that there are no other answer sets of Π.

Example 12. Let the extended program Π consist of just one rule: $\neg q \leftarrow not \;\; p$. Intuitively, this rule means: "q is $false$ if there is no evidence that p is $true$." The only answer set of this program is $\{\neg q\}$. The answers that the program should give to the queries p and q are, respectively, *unknown* and *no*.

Uniqueness of an answer set is an important property of a program. Programs which have a unique answer set are called *categorical*.

The next two examples show that not all programs are categorical. There are programs with multiple answer sets and with no answer sets at all.

Example 13. Consider the logic program $\Pi = \{p \leftarrow not \;\; p\}$. We will show that this program has no answer sets. Consider two cases:

- If $p \in S$ then Π^S is empty and so is its answer set. Since S is not empty it is not an answer set of Π.
- If $p \notin S$ then $\Pi^S = \{p \leftarrow\}$ and hence S is not an answer set of Π.

The program from the next example has two answer sets.

Example 14. Consider a logic program

$p \leftarrow not \; q$

$q \leftarrow not \; p$

It is easy to check that this program has two answer sets $\{p\}$ and $\{q\}$.

An extended logic program Π is said to be inconsistent iff it has the unique answer set $lit(\Pi)$.

Applicability of the extended logic programs to knowledge representation depends on the existence of practical query evaluation methods. We will start discussing this problem by considering a subclass of the class of extended logic programs consisting of general logic programs.

A general logic program Π is said to be *stratified* if there is no recursion through negation as failure *not* in Π. More precisely it means that program Π can be divided into a family of disjoint subprograms $\{\Pi_k : k < n\}$ where Π_0 is a positive logic program and for any $0 < k < n$ if *not* p appears in the body of the rule in Π_k then p does not appear in the head of any rule from Π_j where $j \geq k$.

The following theorem describes an important property of stratified programs.

Proposition 3. [2, 10] Any stratified general logic program is categorical.

It is easy to see that the program from Example 11 is stratified and therefore has only one stable model while program from Example 14 is not stratified.

For stratified logic programs SLDNF resolution [7] provides a sound computational mechanism. This implies that PROLOG which is based on SLDNF resolution can be viewed as a practical query answering mechanism for a broad class of programs i.e. stratified programs which satisfy safety and termination conditions.

Example 15. Consider general logic program Π from Example 11. By changing "$\leftarrow$" to "$:-$" we obtain the PROLOG program consisting of clauses $p(X) :- not\ q(X).$ and $q(a)$.

It is easy to see that for any ground query q an answer *yes* is given by this PROLOG program iff q belongs to the unique answer set of Π.

To answer queries with respect to general logic programs with a multiple number of answer sets, several approaches have been suggested. For an interesting method that uses linear programming see [3].

Let us now show that extended logic programs can be reduced to general logic programs. We will need the following notation:

For any predicate p occurring in Π, let p' be a new predicate of the same arity. The atom $p'(x_1, \ldots, x_n)$ will be called the *positive form* of the negative literal $\neg p(x_1, \ldots, x_n)$. Every positive literal is, by definition, its own positive form. The positive form of a literal l will be denoted by l^+. Π^+ stands for the general logic program obtained from Π by replacing each literal in each rule in Π by its positive form. For any set $S \subset lit(\Pi)$, S^+ stands for the set of the positive forms of the elements of S.

Proposition 4. [11] A consistent set $S \subset Lit$ is an answer set of Π if and only if S^+ is an answer set of Π^+.

This Proposition suggests the following simple way of evaluating queries in extended logic programs. To obtain an answer for query p run queries p and p' on the program Π^+. If Π^+'s answer to p is *yes* then Π's answer to p is *yes*. If Π^+'s answer to p' is *yes* then Π's answer to p is *no*. In the remaining cases the answer is *unknown*. Thus under rather general conditions evaluating query for an extended logic program can be reduced to evaluation of two queries in a general logic program. This implies that for the broad class of extended logic programs PROLOG can be used as the query answering mechanism .

Example 16. For the program Π from Example 12 we have $\Pi^+ = \{q' \leftarrow not\ p\}$. It is easy to see that the PROLOG program corresponding to Π^+ gives answers *no* to q, p, p' and *yes* to q' and therefore the answers to queries q and p based on program Π are, respectively, *no* and *unknown*.

Acknowledgments

We are grateful to Franz Baader, Chitta Baral, Olga Kosheleva, Vladimir Lifschitz, and Rey Reiter for useful discussions on the subject of this paper.

References

[1] J. Alferes and L. Pereira. On logic program semantics with two kinds of negation. In K. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, Washington DC*, pages 574–588. MIT Press, Nov 1992.

[2] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, San Mateo, CA., 1988.

[3] C. Bell, A. Nerode, R. Ng, and V.S. Subrahmanian. Computation and implementation of non-monotonic deductive databases. Technical Report CS-TR-2801, University of Maryland, 1991 (a revised version is to appear in JACM 94.

[4] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, 12:1–80, 1994.

[5] Chitta Baral, Michael Gelfond, and Olga Kosheleva. Approximating general logic programs. In *Proc. of ILPS93*, pages 181–198, 1993.

[6] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un Systeme de Communication Homme-Machine en Francais. Technical report, Groupe de Intelligence Artificielle Universitae de Aix-Marseille II, Marseille, 1973.

[7] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[8] Jurgen Dix. Classifying semantics of disjunctive logic programs. In Krzysztof Apt, editor, *Proc JICSLP'92*, pages 798–812, 1992.

[9] H. Ehrig and Mahr B. *Fundamentals of Algebraic Specifications.* Springler Verlag, 1985.

[10] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.*, pages 1070–1080, 1988.

[11] Michael Gelfond and Vladimir Lifschitz. Logic programs with classical negation. In David Warren and Peter Szeredi, editors, *Logic Programming: Proc. of the Seventh Int'l Conf.*, pages 579–597, 1990.

[12] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–387, 1991.

[13] Michael Gelfond and Halina Przymusińska. Definitions in epistemic specifications. In Anil Nerod, Victor Marek, and Subramanian V. S., editors, *Logic Programming and Non–monotonic Reasoning: Proc. of the First Int'l Workshop*, pages 245–259, 1991.

[14] R.A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[15] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proc. of the Eleventh Int'l Conf. on Logic Programming*, pages 23–38, 1994.

[16] J. Lloyd. *Foundations of Logic Programming.* Springer–Verlag, second edition, 1987.

[17] J. McCarthy. Programs with common sense. In *Proc. of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationery Office.

[18] J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1, 2):27–39,171–172, 1980.

[19] R. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1):75–94, 1985.

[20] Carroll Morgan. *Programming from Specifications.* Prentice Hall, 1990.

[21] Raymond Reiter. On closed world data bases. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum Press, New York, 1978.

[22] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1,2):81–132, 1980.

[23] John Schlipf. Complexity and undecidability results in logic programming. In *Workshop on Structural Complexity and Recursion-theoretic methods in Logic Programming*, 1992.

[24] D. S. Warren and W. Chen. Query evaluation under well-founded semantics. In *Proc. of PODS 93*, 1993.

[25] J. B. Wordsworth. *Software Development with Z.* Addison-Wesley, 1992.