From Functional Specifications to Logic Programs

Michael Gelfond, Alfredo Gabaldon
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968, USA
{mgelfond,alfredo}@cs.utep.edu

Abstract

The paper investigates a methodology for representing knowledge in logic programming using functional specifications. The methodology is illustrated by an example formalizing several forms of inheritance reasoning. We also introduce and study a new specification constructor which corresponds to removal of the closed world assumption from input predicates of functional specifications.

1 Introduction

> *"The only effective way to raise the confidence level of a program significantly is to give a proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand. ...If one first asks oneself what the structure of a convincing proof would be and, having found this, then construct a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance."*
>
> *E. Dijkstra, The Humble Programmer*

This paper continues the mathematical investigation of the *process* of representing knowledge in declarative logic programming (DLP). We are looking for some insights into the ways to specify knowledge, to gradually transfer an initial specification into an executable (and eventually efficient) logic program and to insure the correctness of this transformation. We hope that such insights will help to facilitate the construction of correct and efficient knowledge based systems. In this paper we leave out some of the important aspects of the process of representing knowledge and focus our attention on specific types of representational problems. In particular, we concentrate on the early stages of program development and almost completely ignore the question of elaborating executable (but possibly inefficient) specifications into their efficient counterparts[1]. We are primarily interested in what is entailed by our program and not in specific algorithms used to compute this

[1]Our use of the term "specification" follows [Mor90] which eliminates the distinction between programs and specifications.

entailment. In this sense our approach is complementary to the work on program development in Prolog (see for instance [Dev90]) which concentrates on properties of a particular inference engine. We further simplify our task by limiting attention to a special type of knowledge representation problem which consists in formalizing (possibly partial) definitions of new relations between objects of the problem domain given in terms of old, known relations between these objects. We call such problems *functional KR problems*. They frequently occur in the development of databases when new relations (views) are defined in terms of basic relations stored in the database tables. They are also typical in artificial intelligence (see [Lif93]), e.g., in formalizing knowledge about action and change when we need to define the state of the world at a given moment in terms of its initial (known) state.

The restriction to functional problems allows us to start the programming process with formalizing a natural language description of a problem in terms of functional specifications (f-specifications) [GP96] - functions which map collections of facts about known relations from the domain into collections of facts about new, defined relations. Such specifications can be defined by a specifier directly in a simple set-theoretic language, or they can be built from previously defined specifications with the help of specification constructors - simple mappings from specifications to specifications. After the construction of an f-specification f the designer of the system is confronted with the task of representing f in a logical language with a precisely described entailment relation. [GP96] advocates the use of a language $\mathcal{L}$ of logic programs with two types of negations and the answer set semantics. The choice is determined by the ability of $\mathcal{L}$ to represent default assumptions, i.e., statements of the form "Elements of the class A normally have property P", epistemic statements "P is unknown", "P is possible", and other types of statements needed for describing commonsense domains. Other important factors are the simplicity of the semantics, the existence of a mathematical theory providing a basis for proving properties of programs in $\mathcal{L}$, and the availability of query answering systems which can be used for rapid prototyping. The alternative approach which uses logic programs with well-founded semantics and its extensions can be found in [AP96].

At the end of the second stage of the program development the implementor will have a logic program π_f which, taken in conjunction with a collection X of facts about known relations of f, will entail exactly those facts about the new relations which belong to $f(X)$. Programs of this sort are called lp-functions. In [GP96] the authors suggest that the construction of π_f from f can be substantially facilitated by so called realization theorems which relate specification constructors to some operations on logic programs. They can provide an implementor with a useful heuristic guidance and the means to establish the correctness of his programs. Several examples of such theorems and their applications will be given in the paper.

At the last stage of the process, the lp-function π_f representing f-specification f will be transfered into an efficient logic program Π_f computing (or approx-

imating) the entailment relation of π_f. Unlike π_f, the construction of Π_f will depend significantly on the choice of the query answering system used by the implementors.

Space limitations preclude us from giving any serious comparison with other methodologies of representing knowledge. Moreover, we believe that such comparison can only be done when all of these methodologies are more fully developed. Still a short remark is in order. At the moment, the specification language most frequently used for the first formal refinement of a problem is probably the language of first-order logic (FOL). As others before us we conjecture that FOL is not fully adequate for our purpose. Its expressive power is insufficient to define even fairly simple f-specifications such as transitive closure of database relations. It also doesn't seem to be the best language for representing defaults, epistemic statements, and other types of "commonsense" knowledge. These observations are well known and led to various extensions and modifications of FOL. One of such modifications, DLP, is used by us at the second stage of the programming process. Why not to use it directly? There are two reasons for it. The first advantage of the language of f-specifications over DLP is its simplicity. The construction of f requires knowledge of a simple set-theoretic notation together with definitions of a (hopefully small) collection of specification constructors. The specifier involved at the first stage of the process does not need to know anything about semantics of DLP. Another possible advantage of translating a natural language description of a functional KR problem into an f-specification f is the ability to use the structure of f and the corresponding realization theorems for reducing the construction of π_f to the construction of simpler programs. Examples of such reductions can be found in [GP96].

The previous discussion shows that the success of our approach depends to a large extent on our ability to discover a collection of specification constructors which can serve as building blocks for the construction of f-specifications. This paper is a continuation of a search for such constructors. We introduce and study a new specification constructor, called input opening, which is defined on f-specifications of KR problems which assume the closed world assumption (CWA) [Rei78] on its input predicates. Informally, the input opening f° of f is the result of the removal of this assumption. The notion of input opening is closely related to the notion of interpolation of a logic program from [BGK93]. Interpolation can be viewed as a particular case of input opening defined for specifications which assume the CWA for their outputs as well as inputs and whose input relations are independent from each other. There are many interesting domains which do not satisfy these assumptions, which led us to the introduction of input opening. We give a definition of the constructor, show how it can be decomposed into simpler ones, and prove some useful realization theorems. The use of input opening (in combination with several previously defined constructors) is illustrated by the design of a concise, but fairly powerful program representing a "classical" KR problem associated with inheritance hierarchies. Our solu-

tion generalizes previously suggested solutions to this problem by allowing information about class membership in hierarchies to be incomplete. The program development is accompanied by a simultaneous proof of its correctness. We find that our confidence level in the correctness of the result was significantly improved by this approach. The paper is organized as follows. Section 2 contains the definitions of f-specification and lp-function. Their use is illustrated by formalizing a simple hierarchical reasoning problem under CWA. In Section 3 we define input opening and use this constructor to represent several other problems related to inheritance reasoning in the absence of CWA.

2 F-specifications and lp-functions

2.1 Definitions

A signature is a triple of disjoint sets called object constants, function constants, and predicate constants. Signature $\sigma_1 = \{O_1, F_1, P_1\}$ is a subsignature of signature $\sigma_2 = \{O_2, F_2, P_2\}$ if $O_1 \subseteq O_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$; $\sigma_1 + \sigma_2$ denotes signature $\{O_1 \cup O_2, F_1 \cup F_2, P_1 \cup P_2\}$. Terms over σ are built as in the first-order language; positive literals (atoms) have the form $p(t_1, \ldots, t_n)$, where the t's are terms and p is a predicate symbol of arity n; negative literals are of the form $\neg p(t_1, \ldots, t_n)$. Literals of the form $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ are called contrary. By $\overline{l}$ we denote the literal contrary to l. Literals and terms not containing variables are called ground. The sets of all ground terms, atoms and literals over signature σ are denoted by $terms(\sigma)$, $atoms(\sigma)$ and $lit(\sigma)$ respectively. For a list of predicate symbols $p_1, \ldots, p_n$ from σ, $atoms(p_1, \ldots, p_n)$ ($lit(p_1, \ldots, p_n)$) denote the sets of ground atoms (literals) of σ formed with predicates $p_1, \ldots, p_n$. Consistent sets of ground literals over signature σ are called *states* of σ and denoted by $states(\sigma)$.

A four-tuple $f = \{f, \sigma_i(f), \sigma_o(f), dom(f)\}$ where

1. $\sigma_i(f)$ and $\sigma_o(f)$ are signatures;
2. $dom(f) \subseteq states(\sigma_i(f))$;
3. f is a function which maps $dom(f)$ into $states(\sigma_o(f))$

is called *f-specification* with input signature $\sigma_i(f)$, output signature $\sigma_o(f)$ and domain $dom(f)$. States over $\sigma_i(f)$ and $\sigma_o(f)$ are called input and output states respectively.

By a logic program π over signature $\sigma(\pi)$ we mean a collection of rules of the form

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

where l's are literals over $\sigma(\pi)$ and *not* is negation as failure [Cla78, Rei78]. $head(r) = \{l_0\}$, $pos(r) = \{l_1, \ldots, l_m\}$, $neg(r) = \{l_{m+1}, \ldots, l_n\}$. $head(\pi)$ is the union of $head(r)$ for all rules from π. Similarly for *pos* and *neg*. We say that a literal $l \in lit(\sigma(\pi))$ is entailed by π ($\pi \models l$) if l belongs to all answer sets of π. A program with a consistent answer set is called consistent.

A four-tuple $\pi = \{\pi, \sigma_i(\pi), \sigma_o(\pi), dom(\pi)\}$ where

1. π is a logic program (with some signature $\sigma(\pi)$);
2. $\sigma_i(\pi), \sigma_o(\pi)$ are sub-signatures of $\sigma(\pi)$ called input and output signatures of π respectively;
3. $dom(\pi) \subseteq states(\sigma_i(\pi))$

is called *lp-function* if for any $X \in dom(\pi)$ program $\pi \cup X$ is consistent, i.e., has a consistent answer set. For any $X \in dom(\pi)$,

$$\pi(X) = \{l : l \in lit(\sigma_o(\pi)), \pi \cup X \models l\}.$$

We say that an lp-function π *represents* an f-specification f if π and f have the same input and output signatures and domains and for any $X \in dom(f)$, $f(X) = \pi(X)$.

2.2 An Example

In this section, we illustrate the notions of functional specification and lp-function by solving a knowledge representation problem associated with a simple type of taxonomic hierarchies called the *is-nets*. The problem of specifying and representing is-nets is commonly used to test strengths and weaknesses of various nonmonotonic formalisms. Logic programming approaches to this problem (which assume completeness of its domain) can be found in [AP96] and [Lin91]. Modifications of this example will be used throughout this paper.

An is-net N can be viewed as a combination of graphs N_s and N_d where N_s describes the subclass relation between classes and N_d consists of positive and negative links connecting classes with properties. These links represent defaults "elements of class c normally satisfy (do not satisfy) property p". To simplify the presentation we will assume that N_s is acyclic and that a class c and a property p can be connected by at most one link. We use (possibly indexed) letters o, c, p, and d to denote objects, classes, properties and defaults respectively. Fig 1a gives a pictorial representation of a net. Here $c_0, \ldots, c_5$ are classes and p is a property. Links from c_5 to p and from c_4 to p represent positive and negative defaults while the other links represent subclass relationships.

There are many knowledge representation problems which can be associated with a net N. We start with the simplest one when N is viewed as an

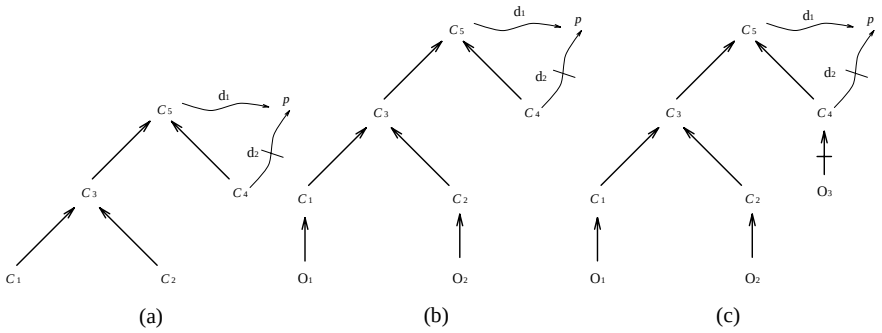

Figure 1: A simple taxonomic hierarchy

informal specification of a function f_N which takes as an input *complete* collections of ground literals formed by predicate symbol *is* and computes all possible conclusions about relation *has* which a rational agent can obtain from this net.[2] Pictorially, the input to f_N is represented by positive links from objects to classes (see Fig 1b). It is assumed that an object o is an element of a class c iff N_s contains a path from o to c.

We are interested in applying our methodology for providing a rigorous specification of this function and for finding its logic programming representation. Later we consider more complex functions which can also be associated with N.

• We start by playing the role of a specifier and give a precise definition of function f_N using the language of f-specifications. To this goal, we first identify graphs N_s and N_d with some encoding of collections of literals of the form $subclass(c_1, c_2)$, $default(d, c, p, +)$, $default(d, c, p, -)$ specified by these graphs. (The last parameter in $default$ is used to distinguish positive and negative defaults.) For instance, a net N_s may be encoded by a logic program consisting of rules

$subclass_0(c_i, c_j)$. (where c_i, c_j are classes connected by a link of N_s.)

$subclass(C_i, C_j) \leftarrow subclass_0(C_i, C_j)$

$subclass(C_i, C_j) \leftarrow subclass_0(C_i, C_k), subclass(C_k, C_j)$

or by some other means. N_d for the net from Fig 1 consists of two statements:

$default(d_1, c_5, p, +)$. $\qquad$ $default(d_2, c_4, p, -)$.

Since we assume that our information about the membership relation *is* is complete, i.e., for any object o and class c, $is(o, c)$ or $\neg is(o, c)$ belongs to the net's input, we call f_N a *closed domain* specification of N. (To simplify the notation we will from now on omit the index N whenever possible). A closed domain f-specification f of a net N can be defined as follows.

[2] $is(o, c)$ stands for "object o is an element of class c"; $has(o, p)$ means that "object o has property p". Both predicates are typed.

1. Input signature $\sigma_i(f)$ of f consists of object constants for objects and classes of the hierarchy and the predicate symbol *is*; output signature $\sigma_o(f)$ consists of object constants for the hierarchy objects and properties and predicate symbol *has*.

2. $dom(f)$ consists of complete states of $\sigma_i(f)$ which satisfy the constraints:[3]

$$\leftarrow is(O, C_1), subclass(C_1, C_2), \neg is(O, C_2) \qquad (1)$$

3. For any $X \in dom(f)$, $has(o, p) \in f(X)$ iff there are d_1 and c_1 s.t.

   (a) $default(d_1, c_1, p, +) \in N$

   (b) $is(o, c_1) \in X$

   (c) for any $default(d_2, c_2, p, -) \in N$,

$$\neg is(o, c_2) \in X \text{ or } subclass(c_1, c_2) \in N$$

Similarly for $\neg has(o, p)$.

Note that this definition does not require any sophisticated mathematics. In particular, it presupposes no knowledge of logic programming.

• Now let us assume the role of an implementor, who just received a description of N and f and is confronted with the task of building an executable lp-function representing f. To simplify the discussion let us assume that we will only be interested in getting answers to ground queries formed by predicate *has*. According to our methodology, we will first ignore the executability requirement and proceed with the construction of an lp-function π representing the f-specification f. We start with considering a program π:

$$\left.\begin{array}{rll}
has(X,P) \leftarrow & default(D,C,P,+), & \\
 & is(X,C), & \\
 & not\ exceptional(X,D,+). & \\
\neg has(X,P) \leftarrow & default(D,C,P,-), & \\
 & is(X,C), & \\
 & not\ exceptional(X,D,-) & \\
exception(E,D_1,+) \leftarrow & default(D_1,C,P,+), & \\
 & default(D_2,E,P,-), & \\
 & not\ subclass(C,E). & \\
exception(E,D_1,-) \leftarrow & default(D_1,C,P,-), & \\
 & default(D_2,E,P,+), & \\
 & not\ subclass(C,E). & \\
exceptional(X,D,S) \leftarrow & exception(E,D,S), & \\
 & is(X,E). & \\
N_s \cup N_d & &
\end{array}\right\} \pi$$

[3] A constraint is a rule of the form $\leftarrow \Delta$ where Δ is a list of literals from some signature σ. A set $X \in states(\sigma)$ *satisfies* the constraint $\leftarrow \Delta$ if $\Delta \not\subseteq X$. X satisfies a collection C of constraints if it satisfies every constraint in C.

(Here N_s and N_d are logic programming encodings of the corresponding nets.) We would like π to be viewed as an lp-function whose input and output signatures are the same as in f and whose domain is $dom(f)$, i.e., we need the following

Proposition 2.1 For any $X \in dom(f)$, program $\pi \cup X$ is consistent.

Now we can show the correctness of our construction.

Proposition 2.2 Lp-function π represents f, i.e., for any $X \in dom(f)$, $\pi(X) = f(X)$.

The next refinement of our program will address the question of specifying its input X. We decided that the input to f will be represented by positive links from objects to classes and that an object o is an element of a class c iff N_s contains a path from o to c. It is easy to check that, under this assumption, we can replace a complete input X to our lp-function π by a program π_X consisting of atoms of the form $is_0(o, c)$ for any link from o to c which is present in the graph ($is_0(o_1, c_1)$ and $is_0(o_2, c_2)$ in Fig 1b), together with three rules:

$$is(O, C) \leftarrow is_0(O, C)$$

$$is(O, C_2) \leftarrow is_0(O, C_1), subclass(C_1, C_2)$$

$$\neg is(O, C) \leftarrow not\ is(O, C)$$

It is also easy to show that for ground queries the program $\pi \cup \pi_X$ is executable by a simple modification of a Prolog interpreter which replaces $\neg has(O, P)$ by a new predicate symbol $\hat{has}(O, P)$. Since our focus in this paper is on the first two steps of the development process we will not discuss this question further. Instead we introduce another KR-problem associated with is-nets and demonstrate how a specification constructor, called *input opening*, can be used to specify and represent this problem.

3 Opening closed domain specifications

3.1 Specifying the problem

So far we assumed that the net N is used in conjunction with complete lists of ground literals characterizing the relation *is*. In the process of development and modification of the system this assumption may become too strong and the specifier may decide to remove it from his specification. Now the net N will be used in conjunction with a possibly incomplete set X of ground literals formed by predicate *is*. As before, X must satisfy the constraint (1). Pictorially, the input to the net will be represented by positive and negative links from objects to classes (see Fig 1c). Now the net N can be viewed

as a function F° which takes X as an input and returns all conclusions about relations *is* and *has* which a rational agent can obtain from N and X. (f° will be called the *open domain* specification of N.) The problem is to precisely define the set of all such conclusions. In order to do that the specifier may use a closed domain f-specification f of N together with a specification constructor called the input opening of f. To define this constructor we need the following terminology.

Let D be a collection of states over some signature σ. A set $X \in states(\sigma)$ is called *D-consistent* if there is $\hat{X} \in D$ s.t. $X \subseteq \hat{X}$; $\hat{X}$ is called a *D-cover* of X.

If, for instance, σ is a signature associated with net N_s from Fig 1 and D is the collection of complete sets from $lit(is)$ which satisfy the constraint (1) then $\{is(o_1, c_1), is(o_2, c_2)\}$ is D-consistent while $\{is(o_1, c_1),\ is(o_2, c_2),\ \neg is(o_2, c_3)\}$ is not.

The set of all D-covers of X is denoted by $c(D, X)$. The set of all D-consistent states of σ is called the *interior* of D and is denoted by D°. An f-specification f defined on a collection of complete states of $\sigma_i(f)$ is called *closed domain specification.*

Definition 3.1 *(Input Opening)* Let f be a closed domain specification with domain D. An f-specification f° is called the *input opening* of f if

$$\sigma_i(f^\circ) = \sigma_i(f) \qquad \sigma_o(f^\circ) = \sigma_i(f) + \sigma_o(f) \tag{2}$$

$$dom(f^\circ) = D^\circ \tag{3}$$

$$f^\circ(X) = \bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \ \cup \bigcap_{\hat{X} \in c(D,X)} \hat{X} \tag{4}$$

Now the open domain f-specification f°_N of a net N can be defined as the input opening of its closed domain specification f_N. (Again, we will omit the index whenever possible).

Our next problem is to find an lp-function representing f°. To do that we will show how the input opening of f can be expressed as a composition of two simpler specification constructors called *interpolation* and *domain completion.* We need the following definitions.

Definition 3.2 A set $X \in states(\sigma)$ is called *maximally informative* w.r.t. a set $D \subseteq states(\sigma)$ if X is D-consistent and

$$X = \bigcap_{\hat{X} \in c(D,X)} \hat{X} \tag{5}$$

By $\tilde{D}$ we denote the set of states of σ maximally informative w.r.t. D.

Consider the net N from Fig 1b. The set $\{is(o_1, c_1)$, $is(o_1, c_3)$, $is(o_1, c_5)$, $is(o_2, c_2)$, $is(o_2, c_3)$, $is(o_2, c_5)\}$ is maximally informative w.r.t. the set of all complete input states of N, while the set $\{is(o_1, c_1)$, $is(o_2, c_2)\}$ is not.

Definition 3.3 *(Interpolation)* Let f be a closed domain f-specification with domain D. F-specification $\tilde{f}$ with the same signatures as f and the domain $\tilde{D}$ is called the *interpolation* of f if

$$\tilde{f}(X) = \bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \tag{6}$$

This is a slight generalization of the notion of interpolation introduced in [BGK93], where the authors only considered interpolations of functions defined by general logic programs.

Definition 3.4 *(Domain Completion)*
Let D be a collection of complete states over signature σ. The *domain completion* of D is a function $\tilde{f}_D$ which maps D-consistent states of σ into their maximally informative supersets.

Specifications f and g s.t. $\sigma_o(f) = \sigma_i(g)$ and $lit(\sigma_i(g)) \cap lit(\sigma_o(g)) = \emptyset$ can be combined into a new f-specification $g \circ f$ by a specification constructor $\circ$ called *incremental extension* [GP96]. Function $g \circ f$ with domain $dom(f)$, $\sigma_i(g \circ f) = \sigma_i(f), \sigma_o(g \circ f) = \sigma_o(f) + \sigma_o(g)$ is called the *incremental extension* of f by g if for any $X \in dom(g \circ f), g \circ f(X) = f(X) \cup g(f(X))$. The following proposition follows immediately from the definitions.

Proposition 3.1 For any closed domain f-specification f with domain D

$$f^\circ = \tilde{f} \circ \tilde{f}_D \tag{7}$$

3.2 Realization theorems for domain completion and interpolation

The above proposition shows that a representation for f° can be constructed from lp-functions representing $\tilde{f}$ and $\tilde{f}_D$. In the construction of these functions we will be aided by realization theorems for domain completions and interpolations.

Let C be a collection of constraints of the form $\leftarrow \Delta$ where $\Delta \subset lit(\sigma)$. A constraint is called *binary* if Δ consists of two literals. We say that a domain D is defined by C if D consists of complete sets from $states(\sigma)$ satisfying C. Let C be a set of binary constraints and D be the closed domain defined by C. Let $\tilde{\pi}_D$ be a program obtained from C by replacing each rule $\leftarrow l_1, l_2$ by the rules $\neg l_1 \leftarrow l_2$ and $\neg l_2 \leftarrow l_1$.

Theorem 3.1 *(Realization Theorem for Domain Completion)*
If for every $l \in lit(\sigma)$ there is a set $Z \in D$ not containing l then a four-tuple $\{\tilde{\pi}_D, \sigma, \sigma, D^\circ\}$ is an lp-function which represents domain completion $\tilde{f}_D$ of D.

To give a realization theorem for the interpolation we need some auxiliary definitions.

Let D be a collection of complete states over a signature σ. Function f defined on the interior of D is called *separable* if

$$\bigcap_{\hat{X} \in c(D,X)} f(\hat{X}) \subseteq f(X)$$

or, equivalently, if for any $X \in dom(f)$ and any output literal l s.t. $l \notin f(X)$ there is $\hat{X} \in c(D, X)$ s.t. $l \notin f(\hat{X})$.
The following examples may help to better understand this notion.

Example 3.1 Let D be the set of complete states over some signature σ_i and let π be an lp-function defined on $D^\circ = states(\sigma_i)$, s.t.

1. The sets of input and output predicates of π are disjoint and input literals do not belong to the heads of π;
2. for any $l \in \sigma_i$, $l \notin lit(\pi)$ or $\overline{l} \notin lit(\pi)$. (By $lit(\pi)$ we mean the collection of all literals which occur in the rules of the ground instantiation of π.)

Then π is separable.

The next example shows that the last condition is essential.

Example 3.2 Let $D = \{\{p(a)\}, \{\neg p(a)\}\}$ and consider a function f_1 defined on D° by the program

$q(a) \leftarrow p(a)$

$q(a) \leftarrow \neg p(a)$

Let $X = \emptyset$. Obviously, $f_1(X) = \emptyset$ while $\bigcap_{\hat{X} \in c(D,X)} f_1(\hat{X}) = \{q(a)\}$ and hence f_1 is not separable.

Example 3.3 In some cases to establish separability of an lp-function π it is useful to represent π as the union of its independent components and to reduce the question of separability of π to separability of these components. Let π be an lp-function with input signature σ_i and output signature σ_o. We assume that the input literals of π do not belong to the heads of rules of π. We say that π is decomposable into independent components $\pi_0, \ldots, \pi_n$ if $\pi = \pi_0 \cup \ldots \cup \pi_n$ and $lit(\pi_k) \cap lit(\pi_l) \subseteq lit(\sigma_i)$ for any $k \neq l$. It is easy to check that, for any $0 \leq k \leq n$, four-tuple $\{\pi_k, \sigma_i, \sigma_o, dom(\pi)\}$ is an lp-function, and that if all these functions are separable then so is π. This observation can be used for instance to establish separability of function f_2 defined on the interior of the set D from the previous example by the program

$q_1(a) \leftarrow p(a)$

$q_2(a) \leftarrow \neg p(a)$

(The output signature of f_2 consists of a, q_1 and q_2).

□

Now we are ready to formulate our next theorem.

Theorem 3.2 *(Realization Theorem for Interpolation)* Let f be a closed domain specification with domain D represented by an lp-function π and let $\tilde{\pi}$ be the program obtained from π by replacing some occurrences of input literals l in $pos(\pi)$ by $not\ \overline{l}$. Then $\{\tilde{\pi}, \sigma_i(f), \sigma_o(f), dom(\tilde{f})\}$ is an lp-function and if $\tilde{\pi}$ is separable and monotonic then $\tilde{\pi}$ represents $\tilde{f}$.

3.3 Representing the open domain specification of N

As before, we now need to address the task of constructing an lp-function π° representing f°. We already know that $f^\circ = \tilde{f} \circ \tilde{f}_D$ where D is the domain of f. This means that we need to find representation $\tilde{\pi}$ of $\tilde{f}$ and $\tilde{\pi}_D$ of $\tilde{f}_D$. An important heuristic guidance in this task will be provided to us by the corresponding realization theorems. To find the representation of $\tilde{f}$ we use Theorem 3.2. The program $\tilde{\pi}$ can be obtained from π by replacing the rule defining predicate *exceptional* by the rule

$$\begin{aligned} exceptional(X, D, S) \leftarrow \quad & exception(E, D, S), \\ & not\ \neg is(X, E). \end{aligned} \tag{8}$$

(which is the only way to turn π into a signed program using the transformation from Theorem 3.2.) We may show that $\tilde{\pi}$ is separable and hence:

Proposition 3.2 $\tilde{\pi}$ represents $\tilde{f}$

To complete the construction of π° we need to find the representation $\tilde{\pi}_D$ of the domain completion of $D = dom(f)$. We use Theorem 3.1. The corresponding program $\tilde{\pi}_D$ consists of the rules

$$is(O, C_2) \leftarrow is(O, C_1), subclass(C_1, C_2) \tag{9}$$

$$\neg is(O, C_1) \leftarrow \neg is(O, C_2), subclass(C_1, C_2) \tag{10}$$

Proposition 3.3 $\tilde{\pi}_D$ represents $\tilde{f}_D$

Finally, using Propositions 3.2, 3.3 and the realization theorem for incremental extension from [GP96] we can prove:

Proposition 3.4 π° represents f°

Proposition 3.4 shows the correctness of π° w.r.t. our specification. However, due to the left recursion in the rules 9, 10 π° cannot be run with the Prolog interpreter. It was however run under a simple meta-interpreter based on the SLG inference engine [CSW95] which is sound w.r.t. our semantics. Of course, the left recursion can be eliminated by introducing a new predicate is_0 but we will not do it here due to the space limitations.

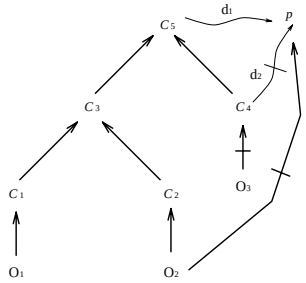

Figure 2: Hierarchy with links from objects to properties

3.4 A simple generalization

In this section we generalize the KR problem associated with a net N by allowing strict (non-defeasible) links from objects to properties to belong to the net's input (see Fig 2). We show that this generalization can be easily incorporated into the design. To do that we use another specification constructor from [GP96]. We will recall the following definitions from [GP96].

Definition 3.5 Let f be a functional specification with disjoint sets of input and output predicates. A f-specification f^* with input signature $\sigma_i(f)+\sigma_o(f)$ and output signature $\sigma_o(f)$ is called *input extension* of f if

1. f^* is defined on elements of $dom(f)$ possibly expanded by consistent sets of literals from $\sigma_o(f)$,
2. for every $X \in dom(f)$, $f^*(X) = f(X)$,
3. for any $Y \in dom(f^*)$ and any $l \in lit(\sigma_o(f))$,
   (i) if $l \in Y$ then $l \in f^*(Y)$
   (ii) if $l \notin Y$ and $\overline{l} \notin Y$ then $l \in f^*(Y)$ iff $l \in f(Y \cap lit(\sigma_i(f))$

Definition 3.6 Let π be an lp-function. 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 $l_0 \in lit(\sigma_o(f))$ 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 the *guarded version* of π and is denoted by $\hat{\pi}$.

Theorem 3.3 ([GP96]) *(Realization Theorem for Input Extension)*
Let f be a specification represented by lp-function π with signature σ. If the set $U = lit(\sigma) \setminus lit(\sigma_o)$ is a splitting set of π dividing π into two components $\pi_2 = top(\pi, U)$ and $\pi_1 = base(\pi, U)$ then lp-function $\pi^* = \pi_1 \cup \hat{\pi}_2$ represents the input extension f^* of f.

Now we can give a specification of the third function associated with a net N. It is defined by a specification

$$f^* = \tilde{f}^* \circ f_D$$

and the representation π^* is obtained by replacing the *has* rules in π° by

$$\left.\begin{array}{rll} has(x,p) \leftarrow & default(d,c,p,+), & \\ & is(x,c), & \\ & not\ exceptional(x,d,+), & \\ & not\ \neg has(x,p). & \\ \neg has(x,p) \leftarrow & default(d,c,p,-), & \\ & is(x,c), & \\ & not\ exceptional(x,d,-) & \\ & not\ has(x,p). & \end{array}\right\}$$

The following proposition follows immediately from the construction of π^* and Theorem 3.3

Proposition 3.5 π^* represents f^*

Again, specification constructors and their realization theorems provided a useful heuristic guidance and allowed to build a program provenly satisfying the corresponding specification.

4 Conclusion

The main contributions of this paper consist in

- Introducing the input opening of a closed domain specification and proving some properties of this constructor;
- Providing a case study for our methodology.

Somewhat surprisingly, the resulting class of programs formalizes inheritance reasoning with incomplete information which was not previously formalized. Unfortunately, the size limitations do not allow us to include proofs. They can be found in [GG97].

Acknowledgments

We would like to thank the referees for valuable comments. The first author acknowledges the support of NASA under grant NCCW-0089.

References

[AP96] J.J. Alferes and L.M. Pereira. *Reasoning with Logic Programming*, Lecture Notes in Artificial Intelligence. Springer. 1996.

[BGK93] C. Baral, M. Gelfond, and O. Kosheleva. Approximating general logic programs, *Proc. ILPS*, pp. 181-198, 1993.

[CSW95] W. Chen, T. Swift and D. Warren. Efficient top-down computation of queries under the well-founded semantics, *Journal of Logic Programming*, 24,3:161–201, 1995.

[Cla78] K. Clark. Negation as failure. In H. Gallaire and J. Minker, eds., *Logic and Data Bases*, pp. 293–322. Plenum Press, NY, 1978.

[Dev90] Y. Deville. *Logic Programming: systematic program development*, Clark, K., series editor, Addison-Wesley Publishing Co., 1990.

[GG97] M. Gelfond and A. Gabaldon. From functional specifications to logic programs. Technical report. Available from: http://cs.utep.edu/gelfond/gelfond.html

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365-385, 1991.

[GP96] M. Gelfond and H. Przymusinska. Towards a theory of elaboration tolerance: logic programming approach. *Int'l Journal of Software Engineering and Knowledge Engineering*, 6(1):89-112, 1996.

[Kun89] K. Kunen. Signed data dependencies in logic programs. *Journal of Logic Programming*, 7(3):231-245, 1989.

[Lif93] V. Lifschitz. Restricted Monotonicity. In *Proc. of AAAI-93*, pp. 432–437, 1993.

[LT94] V. Lifschitz and H. Turner. Splitting a Logic Program. In P. Van Hentenryck, editor, *Proc. 11th ICLP*, pp. 23–38, 1994.

[Lin91] F. Lin. *A study of nonmonotonic reasoning*, Ph.D. Thesis, Stanford U., 1991.

[Mor90] C. Morgan. In C.A.R. Hoare, series ed., *Programming from specifications*, Prentice Hall, 1990.

[Rei78] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, eds., *Logic and Data Bases*, Plenum Press, NY, pp. 119–140, 1978.

[Tur93] H. Turner. A monotonicity theorem for extended logic programs. In D. S. Warren, ed., *Proc. 10th ICLP*, pp. 567–585, 1993.

[Tur94] H. Turner. Signed Logic Programs. In Bruynooghe, M., ed., *Proc. ILPS*, pp. 61–75, 1994.