Logic Programming and Nonmonotonic Reasoning: Functions and Default Values

Pedro Cabalar and David Lorenzo

Dept. of Computer Science
University of Corunna, SPAIN
cabalar@dc.fi.udc.es

Abstract

In this work we explore the possibility of extending the Logic Programming paradigm by replacing predicate-like notation by functional terms. The motivation for this shift arises from the frequent occurrence of functional dependences in the representation of many domains. The exclusive use of predicate notation usually forces us to include extra axioms (mainly, uniqueness of value) to guarantee that they actually behave in a functional way. With the use of functions, we can not only avoid this explicit axiomatization, what leads to a more compact representation, but also obtain some additional interesting features. One of these features is the possibility of replacing default negation by the concept of default values. Besides, as functions range over finite sets of values, they offer the possibility of avoiding full grounding – only the function paramaters need to be grounded, not its value. Another interesting feature is the nested use of functions, which simplifies the representation by avoiding unnecessary variables. The proposed extension will be interpreted under a translation to normal logic programs. However, we also sketch the possibility of adapting standard semantics (like stable models, well-founded semantics or Clark's completion) to the functional case. Finally, we describe how this extension can be further elaborated for it use in Reasoning about Actions and Change.

1 Introduction

One of the most interesting uses of Logic Programming (LP) has probably been its application for practical knowledge representation, and particularly, for solving problems related to the areas of Nonmonotonic Reasoning (NMR) and Commonsense Reasoning about Actions and Change. This application became possible thanks to the availability of several semantics for LP (like *Clark's completion* [4], *stable models* [7] or *well-founded semantics* [16]) that allowed ignoring the less declarative aspects of Prolog, for focusing instead on a more declarative interpretation of logic programs. Besides, the strong connection found in [2] between the use of negation in Prolog programs and one of the most popular NMR formalisms, Reiter's *default logic* [15], offered a new perspective for interpreting logic programs as NMR theories and vice versa. One of the most interesting consequences of this mixed research has been the risen of many extensions of LP for making this paradigm more suitable for knowledge representation. Just to mention some of these extensions, we have the addition of a second negation [8, 1], disjunction in the head [8], nested operators [12] or, more recently, cardinality and weight constraints [14] and aggregate functions [5].

In this work we consider one more possible extension of the LP paradigm consisting in the use of functions instead of relation symbols. The original motivation for this extension comes from the use of LP as underlying semantics for an actions language that exploits functional notation (PAL [3]) in a similar way to [6], where functions are also introduced into the classical planning language STRIPS. The use of functions in logic programs or action theories has typically played a secondary role, mainly associated to expressions with variables for controlling the grounding process. When new user-declared functions are needed, a frequent solution has usually been implementing them as external code in some procedural language (this is the case of the LP tool SMODELS [13] that allows programming functions in C language) or as macros for controlling variable grounding (this is the case, for instance, of the action language $\mathcal{C}+$ [11]). Another typical situation is the one where some relational symbol, for instance $father(x, y)$ actually represents a function $father(x) = y$ in the domain that the logic program tries to capture. In this case, the program must be extended with several rules for explicitly asserting the uniqueness of value y in $father(x, y)$.

The introduction of functions in LP and action theories has provides interesting advantages that are briefly outlined in this paper. First, functions range over a set of values, so that we can generalize the notion of default negation (actually `false` by default) to the notion of *default value* of each function. Second, although we provide a translation from functional into normal logic programs, we also sketch the possibility of generalizing the main LP semantics for the functional case. These adapted semantics could also allow restricting the grounding process to the function arguments, avoiding the grounding of the function value. Third, functions can be arbitrarily nested in compact expressions that considerably reduce the need for auxiliary variables.

The paper is organized as follows. In the next section, we study the syntax of many-sorted functional logic programs and their translation into normal logic programs. After that, we comment some aspects about expressivity, showing that functional logic programs generalize normal and extended logic programming. The next section outlines the possibility of adapting the main LP semantics for a direct interpretation of functional logic programs. Then, we study expressions with nested functions, providing an sketch of how they can be transformed into non-nested ones with some examples. Finally, we talk about a further specialization of this formalism for its use in Reasoning about Actions and Change.

2 Many-Sorted Functional Logic Programs

Assume we have a set of function names $\mathcal{F}$ and, for each function $f \in \mathcal{F}$, a definition like:

$$f : D_1 \times D_2 \times \cdots \times D_n \longrightarrow R \quad [= d]$$

with $n \geq 0$, where $D_1, D_2, \ldots D_n$ and R represent finite sets of constant values, and the declaration of d is optional, representing a *default value* $d \in R$. We use the standard terminology for functions, so that:

$$\begin{aligned} domain(f) &\stackrel{\text{def}}{=} D_1 \times D_2 \times \cdots \times D_n \\ arity(f) &\stackrel{\text{def}}{=} n \\ range(f) &\stackrel{\text{def}}{=} R \end{aligned}$$

and, when specified:

$$default(f) \stackrel{\text{def}}{=} d$$

When f has no defined default value we will sometimes write $default(f) = \mathbf{null}$ where **null** is a pseudo-value, that cannot be compared to another constant value.

As an example, consider the function definitions:

$$\begin{aligned} \mathit{sex} &: \mathit{person} \longrightarrow \{\mathtt{male}, \mathtt{female}\} \\ \mathit{parent} &: \mathit{person} \times \mathit{person} \longrightarrow \mathit{boolean} = \mathtt{false} \\ \mathit{offspring} &: \mathit{person} \times \mathit{person} \longrightarrow \mathit{boolean} = \mathtt{false} \\ \mathit{father,\ mother,\ grandpa,\ grandma} &: \mathit{person} \longrightarrow \mathit{person} \\ \mathit{likes} &: \mathit{person} \times \mathit{person} \longrightarrow \mathit{boolean} \\ \mathit{nationality} &: \mathit{person} \longrightarrow \{\mathtt{es}, \mathtt{pt}, \mathtt{at}, \mathtt{uk}\} = \mathtt{uk} \\ \mathit{birth} &: \mathit{person} \longrightarrow [1900, 2100] \\ \mathit{older} &: \mathit{person} \times \mathit{person} \longrightarrow \mathit{boolean} \end{aligned}$$

where, as usual, we have sort $\mathit{boolean} = \{\mathtt{true}, \mathtt{false}\}$ and, for instance, we define sort $\mathit{person} = \{\mathtt{juan}, \mathtt{mary}, \mathtt{ann}, \mathtt{jim}, \mathtt{bob}, \mathtt{sue}, \mathtt{tom}\}$.

Given a set of function names $\mathcal{F}$ and their definitions, we call *literal* to any expression like

$$f(\overline{v}) = w$$

being f some function name and $\overline{v}$ a tuple of values $v_1, \ldots, v_n$ where, as expected, $n = arity(f)$, $\overline{v} \in domain(f)$ and $w \in range(f)$. Two examples of literals could be $\mathit{parent}(\mathtt{juan}, \mathtt{tom}) = \mathtt{true}$ or $\mathit{age}(\mathtt{juan}) = 65$. For simplicity sake, when the function f is boolean, we may omit the '$= w$' and use a standard literal instead, so that:

$$
\begin{aligned}
f(\overline{v}) &\stackrel{\text{def}}{=} f(\overline{v}) = \texttt{true} \\
\neg f(\overline{v}) &\stackrel{\text{def}}{=} f(\overline{v}) = \texttt{false}
\end{aligned}
$$

A *program* is a finite set of rules like:

$$H \leftarrow B_1, \ldots, B_m$$

where H is a literal called the *head* of the rule and the B_i's are literals that receive altogether the name of *body* of the rule. When $m = 0$ the rule is called a *fact* and we simply write H, omitting the arrow. Sometimes we will use as head a special literal $\perp$ to stand for inconsistence, assuming that when this literal is derived, the final model will be rejected.

In order to write more compact programs, we consider the use of (sort) variables that are understood as abreviations of all their possible instances. Besides, we will allow expressions relating these variables inside the rule bodies, so that they actually specify which combinations of variable values are actually used for grounding the rule.

Example 1 (Family relationships)

Consider now the following facts:

$$
\begin{array}{ll}
\mathit{sex}(\texttt{jim}) = \texttt{male} & \mathit{sex}(\texttt{ann}) = \texttt{female} \\
\mathit{sex}(\texttt{juan}) = \texttt{male} & \mathit{sex}(\texttt{mary}) = \texttt{female} \\
\mathit{sex}(\texttt{tom}) = \texttt{male} & \mathit{sex}(\texttt{sue}) = \texttt{female} \\
\mathit{sex}(\texttt{bob}) = \texttt{male} & \\
\mathit{parent}(\texttt{jim,tom}) & \mathit{parent}(\texttt{mary,tom}) \\
\mathit{parent}(\texttt{tom,bob}) & \mathit{parent}(\texttt{ann,bob}) \\
\mathit{parent}(\texttt{ann,sue}) & \mathit{parent}(\texttt{juan,sue}) \\
 & \\
\mathit{birth}(\texttt{tom}) = \texttt{1950} & \mathit{birth}(\texttt{ann}) = \texttt{1960} \\
\mathit{birth}(\texttt{bob}) = \texttt{1980} & \mathit{birth}(\texttt{sue}) = \texttt{1990} \\
\mathit{nationality}(\texttt{juan}) = \texttt{es} &
\end{array}
$$

The meaning for the functions we had defined before could be captured by the following rules:

$$
\begin{array}{rcll}
\mathit{father}(X) = Y & \leftarrow & \mathit{parent}(Y, X), \mathit{sex}(Y) = \texttt{male} & (1) \\
\mathit{mother}(X) = Y & \leftarrow & \mathit{parent}(Y, X), \mathit{sex}(Y) = \texttt{female} & (2) \\
\\
\mathit{offspring}(X, Y) & \leftarrow & \mathit{parent}(X, Y) & (3) \\
\mathit{offspring}(X, Y) & \leftarrow & \mathit{parent}(X, Z), Z \neq Y,\ \mathit{offspring}(Z, Y) & (4) \\
\\
\mathit{grandpa}(X, Y) & \leftarrow & \mathit{parent}(Z, Y), \mathit{father}(Z) = X & (5) \\
\\
\mathit{likes}(X, Y) & \leftarrow & \mathit{mother}(X) = Y & (6) \\
\neg \mathit{likes}(X, Y) & \leftarrow & \mathit{mother}(X) = M, \mathit{mother}(Y) = M, & \\
 & & \mathit{father}(X) = F, \mathit{father}(Y) = G, & \\
 & & \mathit{nationality}(A) = R, \mathit{nationality}(Y) = S, R \neq S & (7) \\
\\
\mathit{older}(X, Y) & \leftarrow & \mathit{birth}(X) = A, \mathit{birth}(Y) = B, A < B & (8) \\
\mathit{older}(X, Y) & \leftarrow & \mathit{offspring}(X, Y) & (9) \\
\bot & \leftarrow & \mathit{older}(X, Y), \mathit{older}(Y, X), X \neq Y & (10)
\end{array}
$$

□

Notice how boolean function *parent* has been declared false by default in order to avoid specifying those pairs of persons for which one is not parent of the other (which actually constitute most of the possible combinations). On the other hand, *likes* is unknown by default, since in some cases we know it is true, in some cases we know it is false, but in most cases we just do not have any information. For instance, rule (6) says that any person likes his/her mother, whereas rule (7) says that X dislikes Y if they have the same mother, but their fathers are of different nationality.

In order to define the semantics, we will instead provide a straightforward translation into standard extended logic programs. For each functional program P we define the corresponding extended logic program P' as follows. First, all rule literals like $f(\overline{v}) = w$ are reified into expressions like $\mathit{holds}(f(\overline{v}), w)$, using a new introduced predicate *holds*. Second, after applying this literal reification to all the rules in P, we include in the program P' the axiom rule schemata:

$$
\begin{array}{rcll}
\mathit{holds}(f(\overline{v}), d) & \leftarrow & \mathit{not}\ \mathit{holds}(f(\overline{v}), w_1), \ldots, \mathit{not}\ \mathit{holds}(f(\overline{v}), w_n) & (11) \\
\bot & \leftarrow & \mathit{holds}(f(\overline{v}), w), \mathit{holds}(f(\overline{v}), z), w \neq z & (12)
\end{array}
$$

where $f(\overline{v})$ is any ground functional term, d is its default value, and $\{w_1, \ldots, w_n\} = \mathit{range}(f) - \{d\}$. Axiom (11) allows assuming the default value d for any ground term $f(\overline{v})$, whenever the function does not take any of the rest of possible values. Axiom (12) simply gets rid of models where a function takes two different values.

Notice that this semantic description using a translation into a standard logic program has the additional advantage of leaving a free choice for the final LP semantics to be used: stable models, well-founded semantics, Clark's completion, etc.

3 Expressivity

The immediate question about this kind of programs is whether we have reduced the expressivity with respect to standard normal logic programs or extended logic programs.

It is easy to see that normal logic programs can be directly encoded in a straightforward way. Assume we have a (ground) normal logic program P whose set of atoms is Σ. Then, we would just declare each ground atom $p \in \Sigma$ as a 0-ary boolean function:

$$p : \mathit{boolean} = \texttt{false}$$

so that it is assumed to be false by default. Then, the only transformation needed in program rules would be replacing default negation '*not*' by the explicit[1] negation symbol '$\neg$'. Notice how explicit negation behaves as default negation when we have a `false` default value.

As for extended logic programs, the translation is slightly more complex, since we need handling simultaneously default and explicit negation. This can be accomplished by the inclusion of extra atoms for representing default negation. In this case, we would declare all atoms as boolean:

$$p : \mathit{boolean}$$

but without default value. Then, if Σ represents the set of atoms of the original extended logic program, we add a new special function declared as:

$$\mathit{know} : \Sigma \times \mathit{boolean} \longrightarrow \mathit{boolean} = \texttt{false}$$

together with the rule schemata:

$$\begin{array}{rcl} \mathit{know}(p, \texttt{true}) & \leftarrow & p \\ \mathit{know}(p, \texttt{false}) & \leftarrow & \neg p \end{array}$$

for any $p \in \Sigma$. Note that $\mathit{know}(p, v)$ is false by default, and so, the literal $\neg \mathit{know}(p, v)$ will work as default negation. Consequently, the translation would just consist in making the following replacements for extended default literals:

$$\begin{array}{rcl} \mathit{not}\ p & \mapsto & \neg \mathit{know}(p, \texttt{true}) \\ \mathit{not}\ \neg p & \mapsto & \neg \mathit{know}(p, \texttt{false}) \end{array}$$

[1] Indeed, as we saw before, the use of '$\neg$' is an abbreviation. Therefore, the real transformation is replacing each default literal *not* p by the functional literal $p = \texttt{false}$.

for any $p \in \Sigma$.

At a first sight, this special function *know* seems to be interesting for its inclusion as a predefined part of the language. For instance, we could extend its use for any function $f \in \mathcal{F}$ and not just for boolean ones. This would allow expressing conditions like $\neg know(f(\overline{v}), w)$ pointing out that we do not have evidence that $f(\overline{v})$ takes value w. However, this would not have much utility in practice, because we have no choice for just asserting that a function *does not take* a given value without providing more information. In fact, we actually have one these two cases:

(a) either we assert $f(\overline{v}) = w$

(b) or we have that for all $z \in range(f),\ \neg know(f(\overline{v}), z)$.

Thus, an operator like $unknown(f(\overline{v}))$ for representing case (b) seems more practical for our purposes. This operator can be defined as:

$$unknown : \Sigma \longrightarrow boolean = \texttt{true}$$

so that any $f \in \Sigma$ will be unknown by default. Implicitly, we would handle the rule schemata:

$$\neg unknown(f(\overline{v})) \quad \leftarrow \quad f(\overline{v}, w)$$

4 Adapted semantics

Another interesting topic would be the possibility of adapting standard LP semantics to the functional case. In this way, the interpretation of the functional logic program would not rely on a translation to a normal logic program, but could be provided by a direct definition.

Although this topic is currently under study, it is not too difficult to sketch how these functional semantics could be. Consider, for instance, the case of stable models. A set of ground functional literals is said to be consistent if it contains no pair $f(\overline{v}) = w,\ f(\overline{v}) = z$ where $w \neq z$, nor the special literal $\bot$. An interpretation I is any consistent set of ground functional literals. Interpretations can be partial, i.e., they need not contain a literal for each ground functional term. Given a functional program P, its reduct P^I with respect to some interpretation I is obtained by adding the fact $f(\overline{v}) = d$ with $default(f) = d$ for each f such that either $f(\overline{v}) = d\ \in I$ or f has no literal in I. If we apply transitive closure of rule application to P^I, we would obtain a set of literals J. This set J is said to be a (functional) *stable model* iff $J = I$.

Similar adaptations can be done for WFS and Clark's completion.

As a curiosity, it is perhaps interesting to note that some well known representational problems in Logic Programming may easily appear in apparently simple functional logic programs. Consider, for instance, the *Coherence problem* that motivated the introduction of the WFSX variant for WFS. A simple program like:

$$a, b, c \quad : \quad boolean = \mathtt{false}$$

$$
\begin{align}
a &\leftarrow \neg a \tag{13}\\
b &\leftarrow a \tag{14}\\
c &\leftarrow b \tag{15}\\
\neg b & \tag{16}
\end{align}
$$

would need a WFSX-style interpretation for obtaining a reasonable result. The informal justification is the following one. When just applying WFS, we would get that a results undefined, since it only depends on a itself, but under a negative cycle (13). Rules (14) and (15) respectively propagate this undefinedness to b and c. However, we had a fact (16) explicitly asserting that b is false, and so, condition of rule (15) should become unfounded, leaving c false by default. This reasoning is only possible if we apply the coherence principle as done in WFSX.

5 Nesting functional terms

Until now, the use of functions has just limited to a slight change in the shape of program literals. However, one of the most interesting advantages of functional terms is the possibility of constructing nested expressions. Consider, for instance, rule (5) in Example 1. Clearly, variable X is exclusively used for representing the value of $father(Z)$. Thus, it seems natural to replace this auxiliary variable by the term functional term $father(Z)$, writing instead:

$$grandpa(father(Z), Y) \quad \leftarrow \quad parent(Z, Y)$$

The informal reading of is rule would be: "if Z is one of Y's parents then $father(Z)$ is one of $Y's$ grandpas". Similar steps could be applied to rules (6) and (8), respectively leading to:

$$
\begin{aligned}
&likes(X, mother(X))\\
&older(X, Y) \quad \leftarrow \quad birth(X) < birth(Y)
\end{aligned}
$$

However, the most interesting example would be rule (7) where we can save many unnecessary variables:

$$
\begin{align}
\neg likes(X, Y) \quad \leftarrow \quad & mother(X) = mother(Y),\\
& nationality(father(X)) \neq nationality(father(Y)) \tag{17}
\end{align}
$$

Again, a possible topic for future work would be adapting the usual LP semantics for the case of nested functional expressions. Instead, it is easy to see that a nested rule can always be unfolded back into the unnested version by a sucessive introduction auxiliary variables. Thus, for instance, for a rule like (18) we would go replacing each inner subexpression by a fresh variable, generating the sequence of transformations:

$$\begin{array}{lcl} \neg \mathit{likes}(X,Y) & \leftarrow & \mathit{mother}(X) = \mathit{mother}(Y), \\ & & \mathit{nationality}(\mathit{father}(X)) \neq \mathit{nationality}(\mathit{father}(Y)) \end{array}$$

$$\begin{array}{lcl} \neg \mathit{likes}(X,Y) & \leftarrow & V_1 = V_2, \\ & & \mathit{nationality}(\mathit{father}(X)) \neq \mathit{nationality}(\mathit{father}(Y)), \\ & & \mathit{mother}(X) = V_1, \mathit{mother}(Y) = V_2 \end{array}$$

$$\begin{array}{lcl} \neg \mathit{likes}(X,Y) & \leftarrow & V_1 = V_2, \\ & & \mathit{nationality}(V_3) \neq \mathit{nationality}(V_4) \\ & & \mathit{mother}(X) = V_1, \mathit{mother}(Y) = V_2, \\ & & \mathit{father}(X) = V_3, \mathit{father}(Y) = V_4 \end{array}$$

$$\begin{array}{lcl} \neg \mathit{likes}(X,Y) & \leftarrow & V_1 = V_2, \\ & & V_5 \neq V_6, \\ & & \mathit{mother}(X) = V_1, \mathit{mother}(Y) = V_2, \\ & & \mathit{father}(X) = V_3, \mathit{father}(Y) = V_4 \\ & & \mathit{nationality}(V_3) = V_5, \mathit{nationality}(V_4) = V_6 \end{array}$$

that ends up with a rule equivalent to (7).

6 Reasoning about actions and change

In this last section we comment several aspects and examples on the use of LP as underlying nonmonotonic formalism for Reasoning about Actions and Change. This methodology, originally proposed in [9], has been followed in many cases, like the use of stable models for $\mathcal{A}$ and $\mathcal{B}$ languages [10], the use of Clark's completion [2] for $\mathcal{C}+$ language [11] or the use of both WFSX and stable models for language PAL [3]. Let us consider how to adapt the current proposal for representing action scenarios.

First of all, in a Reasoning about Actions problem, we must distinguish between two separated kinds of symbols: *fluents*, which correspond to system variables whose value may change along time; and *actions*, that describe the possible ways in which an external agent may cause a system transition. Although the use of functional fluents has already been considered, for instance, in functional STRIPS [6] or in language C+ (in this last case, only unnested), functional actions is a relatively unexplored possibility that can be directly proposed in our case.

[2] Actually, a generalization called *literal completion*.

For simplicity sake, we will consider a single transition with just two states: the predecessor and the successor states. We will write f' to stand for the value of f at the predecessor state, reserving f itself for referring to the successor one.

Apart from the distinction between actions and fluents, the main property of action theories is that they must deal with an implicit default different from the function default value: the *inertia* principle. Thus, instead of defining a fixed default value for each fluent f, we will have as default the value taken by f', that is, f at the previous state. So, in some sense, the fact for the previous state $f' = d$ acts like declaring $default(f) = d$ for f at the current state.

Fluents can be further specialized into three categories:

1. *static* fluents
2. *inertial* fluents
3. *events*

all of them allowing an explicit default value. The idea of a static fluent correspond to some function whose interpretation is fixed along the transitions sequence. For instance, if we have a planning algorithm for browsing a graph, the set of arcs of the graph can be described by a static fluent, since its structure does not change along the actions execution. This is also a good example for showing the interest of a default value for a static fluent. If we declare:

$$arc : node \times node \longrightarrow boolean = \texttt{false}$$

we will only need to declare the existing arcs, making false by default any pair $arc(a, b)$ for which (a, b) is not an arc.

Inertial fluents would be the most frequent ones, for which the inertia default is taken into account. However, declaring an extra default value could also be interesting, despite of handling the inertia default. This extra default value could be useful, for instance, for a compact declaration of particular states, like the initial state or the goal in a planning problem. As an example, think about a chess-like problem where most cells are empty. Clearly, the content of each cell will vary along time and follow the inertia law. However, when declaring the inital state, we are interested in *exclusively representing* the facts for occupied cells, assuming that the rest of the board cells are empty by default. This can be achieved by declaring the inertial fluent:

$$cell : [1, 8] \times [1, 8] \longrightarrow chessmen \cup \{empty\} = empty$$

Finally, an event would just be a fluent that does not follow the inertia law. In this case, when no value is specified for the fluent, it is usual to require a default value. For instance, consider a system where, at some situations, a *ring* can be heard, but with different volume levels 1,2,3,4 or it is not heard at all, using volume 0. This can be declared as an event:

$$ring : \{0, 1, 2, 3, 4\} = 0$$

so that, when no value has been derived for $ring$, we can assume $ring = 0$ by default.

As for functional actions, a default value would also make sense: it can be used for representing the situation in which the action is not performed at all. For example, consider an action of pressing the brakes with a given intensity in interval [0,8], having default value 0. If we want to apply some rule that depends on the *nonexecution* of $press\_brake$ we could just use the literal $press\_brake = 0$.

they can also

Example 2 (Hamiltonian circuit)
Given a graph G, a hamiltonian circuit is a round path that it visits all the nodes of G exactly once. This problem can be represented as a planning problem as follows. We declare the static fluent:

$$arc : node \times node \longrightarrow boolean = \texttt{false}$$

together with inertial fluents:

$$\begin{array}{rcl} visited & : & node \longrightarrow boolean = false \\ current & : & node \end{array}$$

and action:

$$jump\_to \quad : \quad node$$

The set of rules would be:

$$\begin{array}{l} current = move\_to \\ \quad visited(current) \end{array}$$

$$\begin{array}{rcl} \bot & \leftarrow & visited'(current) \\ \bot & \leftarrow & \neg arc(current', current) \end{array}$$

□

Example 3 (Hanoi towers)
An example also studied in [6] is the typical Hanoi Towers problem, where we have a set of n disks, that can be placed in three pegs. We can move one disk at a time from one peg to another, but taking into account that any disk can only be placed on top of a bigger disk. We want to move a tower of disks from one of the pegs to another.

Given the sorts $peg = \{p_1, p_2, p_3\}$, $disk = \{1, \ldots, n\}$ and $location = disk \cup \{ground\}$, we declare the static fluents:

$$size : disk \longrightarrow integer$$

the inertial fluents:

$$\begin{array}{rcl} loc & : & disk \longrightarrow location \\ top & : & peg \longrightarrow location = ground \end{array}$$

and the action:

$$move : peg \times peg \longrightarrow boolean = \texttt{false}$$

Then, the set of rules would be:

$$\begin{array}{rcl} top(P) = loc'(top'(P)) & \leftarrow & move(P,Q) \\ loc(top'(P)) = top'(Q) & \leftarrow & move(P,Q) \\ top(Q) = top'(P) & \leftarrow & move(P,Q) \\ \bot & \leftarrow & move(P,Q), top(P) = ground \\ \bot & \leftarrow & loc(D) < D \end{array}$$

□

Example 4 (8-puzzle)
The well known 8-puzzle planning problem could be modeled as follows. We have theinertial fluents:

$$\begin{array}{rcl} hole\_row, hole\_col & : & [1,3] \\ board & : & [1,3] \times [1,3] \longrightarrow [1,8] \cup \{empty\} = empty \end{array}$$

and the action:

$$move \quad : \quad \{up, down, left, right\}$$

The set of rules would be:

$$\begin{array}{rcl} hole\_row = hole\_row' + 1 & \leftarrow & move = down \\ hole\_row = hole\_row' - 1 & \leftarrow & move = up \\ hole\_col = hole\_col' + 1 & \leftarrow & move = right \\ hole\_col = hole\_col' - 1 & \leftarrow & move = left \\ board(hole\_row, hole\_col) = empty & & \\ board(hole\_row', hole\_col') = board'(hole\_row, hole\_col) & & \end{array}$$

□

References

[1] J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. 'classical' negation in nonmonotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20(1):107–142, 1998.

[2] N. Bidoit and C. Froidevaux. Minimalism subsumes default logic and circumscription. In *Proc. of the IEEE Symposium on Logic in Computer Science (LICS-87)*, pages 89–97, 1987.

[3] P. Cabalar, M. Cabarcos, and R. P. Otero. PAL: Pertinence action language. In *Proceedings of the 8th Intl. Workshop on Non-Monotonic Reasoning NMR'2000 (Collocated with KR'2000)*, Breckenridge, Colorado, USA, april 2000. (`http://xxx.lanl.gov/abs/cs.AI/0003048`).

[4] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 241–327. Plenum, 1978.

[5] T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in dlv. In *Proc. of the Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, Acapulco, México, 2003.

[6] H. Geffner. *Functional STRIPS: a more flexible language for planning and problem solving*. Kluwer, 2000.

[7] M. Gelfond and V. Lifschitz. The stable models semantics for logic programming. In *Proc. of the 5th Intl. Conf. on Logic Programming*, pages 1070–1080, 1988.

[8] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[9] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *The Journal of Logic Programming*, 17:301–321, 1993.

[10] M. Gelfond and V. Lifschitz. Action languages. *Linköping Electronic Articles in Computer and Information Science*, 3(16), 1998. (`http://www.ep.liu.se/ea/cis/1998/016`).

[11] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, (to appear).

[12] V. Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[13] Ilkka Niemela and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429, 1997. Lecture Notes in Artificial Intelligence, volume 1265.

[14] Ilkka Niemela and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

[15] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[16] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.