A Functional Action Language Front-End

Pedro Cabalar

Dept. Computer Science,
Corunna University,
Fac. Informatica, Campus Elvina,
E-15071, Corunna, SPAIN,
cabalar@dc.fi.udc.es

Abstract. In this paper we extend the notation of functional logic programs introduced in [1] by: (i) describing a formal translation of nested function references; (ii) dealing with set terms in the rule heads to allow non-determinism; and (iii) introducing high level constructions for representing action domains. The semantics we consider in this work, both for functional logic programs and for the actions language, just consists in a translation into (non-ground) logic programs under the answer sets semantics.

1 Introduction

The use of logic programs for representing action domains typically involves dealing with a series of common representational tasks: declaring action and fluent sorts, encoding inertia, generating action occurrences, dealing with qualifications, etc. Although these features can be directly encoded in logic programs in a methodological way, they sometimes mean a considerable programming effort that may deviate the attention from the actions domain itself, apart from making the representation less readable. In this way, it has been frequent instead to the define high-level action languages [2–4] (in many cases with their own high-level semantics) for which a translation into logic programs is provided.

One of those typical situations we face when encoding most action scenarios is dealing with some relational fluent or action, say for instance $location(x, y)$, that has an implicit functional nature $location(x) = y$. In this case, the underlying program must be extended with several constraint rules for explicitly asserting the uniqueness of value for $location(x)$. In fact, some action languages allow handling these so-called *multivalued fluents* [3], to avoid the explicit representation of these constraints. However, the functional nature of a fluent can also be exploited for achieving a more compact and readable representation if we consider nesting function references (something not done in action languages) like for instance in a term like:

$$accessible(location(x)) \wedge distance(location(robot), location(x)) < 30$$

In this work we revisit one extension [1] of the Logic Programming (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 [5]) in a similar way to [6], where functions are also introduced into the classical planning language STRIPS. In the paper, we present a syntactic extension of the approach [1] where: (i) we describe a formal translation of nested function references; (ii) we allow set terms, both for sort definitions and for a new construct in the rule heads that provide non-deterministic choice; and finally (iii) we introduce new types of functions for dealing with representation and reasoning in action domains.

Our main interest is focused on the construction of a front-end that translates from the functional notation into standard logic programs. For this reason, rather than providing a functional semantics (as done in [1]), we will focus instead on the translation into Answer Set Programming (ASP) (which may also be seen as a kind of indirect semantics). The translation follows a similar strategy to other Functional Logic Programming (FLP) systems (like, for instance, `NUE-Prolog` [7]) where functional programs are converted into logic programs via *flattening* (functions of arity N become predicates of arity $N + 1$ with an extra argument for the function value).

An important remark should perhaps be made here. Despite of sharing a similar syntax, our motivation is quite different from the usual programming style in the area of FLP [8]. On the one hand, FLP is closer to Prolog-style programming where we deal with complex data structures (lists, functors, etc) and the solution to a problem is some instantiation of variables. In this way, the main research topics in FLP have to do with lazy evaluation, narrowing (a kind of rewriting) or dealing with higher order functions. On the other hand, the approach we present here can be seen as the functional counterpart of ASP, where we are more concerned with a comfortable knowledge representation for constraint satisfaction problems. In a similar way to ASP, we do not consider functors or lists (in fact, we deal with a finite domain), and the solution has the shape of a set of models (or *functional* answer sets). Besides, our approach is suitable for nonmonotonic reasoning since, as shown in [1], the availability of default values for functions allows the same expressiveness as logic programs with default negation.

The paper is organized as follows. In the next section, we study the syntax of many-sorted functional logic programs, extended here for dealing with sorts and set terms. Section 3 explains the translation of this syntax into logic programs under asnwer sets semantics. Next, we informally comment some aspects for improving the translation with respect to variable grounding. In Section 5, we present the syntax extension for representing actions scenarios. Finally, Section 6 concludes the paper with a discussion and addresses some lines for future work.

2 Many-Sorted Functional Logic Programs

The syntax is defined as follows. We start from a *signature* $\Sigma = \langle \mathcal{F}, \mathcal{S}, \mathcal{V} \rangle$ consisting of a finite set $\mathcal{F}$ of *function names*, a finite set of *sort names*, and a set $\mathcal{V}$

of *constant values*. Letters $f, g, \ldots$ will be used to denote elements of $\mathcal{F}$ whereas $v, w, \ldots$ will stand for constant values. We assume that $\mathcal{V}$ contains at least the boolean constants `true` and `false`. Similarly, $\mathcal{S}$ contains at least the sort name *boolean* whose interpretation will be later fixed to $\{\texttt{true}, \texttt{false}\}$.

We define two types of terms: scalar and set terms. A *scalar term* can be built up with:

- any constant value v,
- any variable X,
- a function reference $f(t_1, \ldots, t_n)$ with t_i scalar terms or
- an arithmetic expression like $-t_1$ or $t_1 \otimes t_2$, with $\otimes \in \{+, -, *, /, mod\}$ and t_1, t_2 scalar terms.

A *set term* is:

- any sort name,
- an expression like $\{t_1, \ldots, t_n\}$, with $n \geq 0$ and t_i scalar terms,
- any construction $A \cup B$, $A \cap B$ or $A \setminus B$, with A, B set terms, or
- any expression like $\{t \mid L_1, \ \ldots, \ L_m \ , \ X_1 : A_1, \ \ldots, \ X_n : A_n\}$, with $m, n \geq 0$, L_i *literals* (to be defined later), X_j variable names and A_j set terms.

This last construction is called an *intensional set* term, and the X_i are its *bounded* variables. An occurrence of a variable X in a term is said to be *free* when it is not in the scope of some intensional set, or is not one of its bounded variables otherwise.

Each function $f \in \mathcal{F}$ will be accompanied by a sentence called the *definition* of f, of the form:

$$f : D_1 \times D_2 \times \cdots \times D_n \longrightarrow R \quad [= d] \tag{1}$$

with $n \geq 0$ (when $n = 0$ the arrow is omitted), where $D_1, D_2, \ldots D_n$ and R are set terms (to be defined next), and the declaration of the scalar term d is optional, representing a *default value* $d \in R$. We will assume that a function without default value can be partial (that is, some elements in the domain may have no associated value). 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:

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

We assume that no free variable occurs in the domain or the range of f.

In fact, one additional expected condition is imposed now on scalar terms to be considered syntactically correct: for any function reference $f(t_1, \ldots, t_n)$ we

must have $arity(f) = n$ in the definition of f. For commodity sake, we adopt the notation $\bar{t}$ to stand for some tuple of expressions $t_1, \ldots, t_n$ with $n \geq 0$.

A *literal* is defined as any expression like $t_1 \otimes t_2$ with $\otimes \in \{\leq, \geq, <, >, =, \neq\}$ where t_1, t_2 are scalar terms, or the expression $unknown(f(\bar{t}))$ (possibly negated), with $\bar{t}$ a tuple of scalar terms. Intuitively, this literal will be true whenever $f(\bar{t})$ has no associated value in the current state of affairs. When the literal has the shape $f(\bar{t}) = t_0$ and $range(f) = boolean$ we also admit the logical notation, so that literal $f(\bar{t})$ stands for $f(\bar{t}) = \texttt{true}$ whereas $\neg f(\bar{t})$ stands for $f(\bar{t}) = \texttt{false}$.

A *rule* is a construction like:

$$H \leftarrow B_1, \ldots, B_m, X_1 : A_1, \ldots, X_n : A_n$$

where H is called the *head* of the rule, the B_i's are literals that receive altogether the name of *body* of the rule, and the A_j are set terms, and X_j are the *bounded* variables of the rule. We assume that rules do not contain free variable occurrences. The rule head H can be a literal like $f(\bar{t}) = t_0$ with $\bar{t}$ a tuple of scalar terms or an expression like $f(\bar{t}) \in A$ with A a set term. In both cases, f receives the name of *head function* of the rule. We will also allow H to be a special symbol $\perp$ to stand for inconsistency, assuming that when this symbol is derived, the final model will be rejected. When $m = 0$ the rule is called a *fact* and we simply write H, omitting the arrow.

As an abbreviation, we allow declaring general variables:

$$\texttt{var} \;\; X : A$$

with X a variable name and A a set term, so that X becomes an implicit bounded variable for all rules in which X occurs free in some literal.

For defining the extent of a sort, we adopt a similar criterion to [9] allowing both an explicit and complete description using a sentence like:

$$\texttt{sort} \;\; s = A$$

with s a sort name, and A a set term, or using instead a boolean function $s : \mathcal{V} \longrightarrow boolean = \texttt{false}$ with the same sort name s (but now with arity 1), so that the sort extent is implicitly defined by:

$$\texttt{sort} \;\; s = \{X \mid s(X)\}$$

A *functional logic program* (FLP for short) is a set of rules, function and sort definitions. We say that an FLP is *well-formed* when there are no cyclic dependences for function and sort definitions. To be precise, let us define a dependence graph where each node is a sort name s, a function name f or the definition of some function f, write it $def(f)$. We add an edge (x, y), pointing out that y depends on x, when:

- y is the head function of some rule, and x is a (different) function or sort name occurring in that rule,

- y is a sort name and x a fluent or sort name occurring in the definition of y,
- $y = def(f)$ and x a fluent or sort name occurring in the definition of f, or
- $y = f$ and $x = def(f)$ for some function name f.

Then, a program is said to be *well-formed* if there is no cyclic path in the dependences graph including a sort name s or a function definition $def(f)$.

Example 1. **(Hamiltonian circuit)**
Given a graph G, a hamiltonian circuit is a round path that visits all the nodes of G exactly once. We assume that G contains at least some node, call it 0.

$$\begin{aligned} arc &: node \times node \longrightarrow boolean = \texttt{false} \\ visited &: node \longrightarrow boolean = \texttt{false} \\ next &: node \longrightarrow node \\ \texttt{var} \quad & X, Y : node \end{aligned}$$

$$\bot \leftarrow next(X) = next(Y),\ X \neq Y \tag{2}$$

$$next(X) \in \{Z \mid arc(X, Z), Z : node\} \tag{3}$$

$$visited(0) \tag{4}$$

$$visited(next(X)) \leftarrow visited(X) \tag{5}$$

$$\bot \leftarrow \neg visited(X) \tag{6}$$

□

Function *arc* acts as a predicate that is false by default, so that we just wish to specify the existing arcs (and not the non-existing ones). Function *next* points out, for any node, which is the next node in the path. As it is a function, the next node is unique. Rule (2) is used to guarantee that two different nodes do not "jump" to a common next node. Rule (3) generates some possible value for each $next(Y)$. In this way, with (2) and (3) we guarantee that all nodes have exactly one successor and one predecessor. However, it could still be the case that we had several unconnected cycles. To rule out this possibility, predicate *visited* (false by default) is used to point out which nodes are accessible from node 0, including itself with fact (4) and propagating through connected nodes with rule (5). Finally, (6) guarantees that all nodes are visited.

The FLP in Example 1 is well-formed, provided that the definition of sort *node* refers to constants, or at least, does not refer to *arc*, *visited* or *next*.

3 Translation into logic programs

Since our aim is to introduce this language as a front-end for ASP solvers, the semantics we consider in this work[1] is just described by a translation into logic programs.

[1] In [1] the reader will find a proper semantics for a syntactic subset of FLPs where sort definitions and set terms were not present yet.

Given an FLP Π, we will call Π' to the corresponding logic program (LP). For each function in Π defined as in (1) we will handle a predicate like

$$holds(f, X_1, \ldots, X_n, V)$$

in Π', or $holds(f, \overline{X}, V)$ for short.

We will provide a translation of terms and literals as follows. For each term (or literal) t in the FLP we will describe the corresponding term t' in the LP plus an additional set of LP literals, call them $lits(t)$. The idea for these additional literals is that t can be replaced by t' in a rule body, provided that $lits(t)$ are also included in that LP body.

For an easier description of translations, we will consider rule bodies as sets of literals and, furthermore, we will also allow including a set of literals $lits$ in a body $\{B_1, \ldots, B_n, lits\}$ actually meaning $\{B_1, \ldots, B_n\} \cup lits$. For any body $B = \{L_1, \ldots, L_n\}$, we define B' and $lits(B)$ respectively as $\{L'_1, \ldots, L'_n\}$ and $lits(L_1) \cup \cdots \cup lits(L_n)$. Similarly, given tuple $\bar{t} = \langle t_1, \ldots, t_n \rangle$, we have $(\bar{t})' = \langle t'_1, \ldots, t'_n \rangle$ and $lits(\bar{t}) = lits(t_1) \cup \cdots \cup lits(t_n)$.

Definition 1 (Translation of scalar terms). *The* translation of a scalar term t *is the pair* $\langle\ t', lits(t)\ \rangle$ *defined as follows:*

1. *When t is a constant or a variable $t' \stackrel{\text{def}}{=} t$ and $lits(t) \stackrel{\text{def}}{=} \emptyset$.*
2. *When $t = f(\bar{t})$, we define $t' \stackrel{\text{def}}{=} V$ with V a new fresh variable and*

$$lits(t) \stackrel{\text{def}}{=} \{\ holds(f, (\bar{t})', V)\ \} \cup lits(\bar{t})$$

3. *When t is an arithmetic expression $t_1 \otimes t_2$, $t' \stackrel{\text{def}}{=} t'_1 \otimes t'_2$ and $lits(t) \stackrel{\text{def}}{=} lits(t_1) \cup lits(t_2)$. Similarly, when $t = -t_1$, we have $t' = -t'_1$ and $lits(t) \stackrel{\text{def}}{=} lits(t_1)$.*

□

Definition 2 (Translation of literals). *The* translation of a literal L *is again a pair* $\langle\ L', lits(L)\ \rangle$ *where:*

1. *When $L = t_1 \otimes t_2$ with $\otimes$ a relational symbol, the translation is similar to that of arithmetical terms: $L' \stackrel{\text{def}}{=} t'_1 \otimes t'_2$ and $lits(L) \stackrel{\text{def}}{=} lits(t_1) \cup lits(t_2)$.*
2. *When $L = t \in A$, $L' = A'(t')$ and $lits(L) = lits(t)$, where A' is obtained from the translation of set term A, defined below.*
3. *When $L = unknown(f(\bar{t}))$, $L' \stackrel{\text{def}}{=} not\ known(f, (\bar{t})')$ and $lits(L) = lits(\bar{t})$.*

□

The translation of a set term A will be done by including a new predicate symbol A' in the LP with the corresponding set of LP rules, $rules(A)$, to fix the set extent. A problem we must solve is the case in which A contains free variables $\overline{Y}$. Note that this may only happen when A is a rule head like in:

$$f(\bar{t}) \in A \leftarrow B_1, \ldots, B_m, X_1 : A_1, \ldots, X_n : A_n$$

On the one hand, these free variables $\overline{Y}$ are needed for fixing the set extent, but on the other hand, once they are included in *different* rules they lose their meaning, unless we repeat the body B (or its relevant part) we had in the original FLP rule where A appeared. Let us call $body(B)$ to the translation of a rule body, to be defined later.

Definition 3 (Translation of set terms). *The* translation of a set term A, *possibly with free variables* $\overline{Y}$ *and occurring in a rule with body* B, *is a pair* $\langle\, A', rules(A) \,\rangle$ *where* A' *is a predicate name and* $rules(A)$ *a set of rules, defined as follows:*

1. *When* A *is a sort name* s, *we just define* $A' \stackrel{\text{def}}{=} s$ *and* $rules(A) = \emptyset$.
2. *When* $A = \{t_1, \ldots, t_n\}$, *we fix* A' *to some new fresh predicate name, and* $rules(A)$ *contains the rules:*

$$A'(\overline{Y}, t'_i) \leftarrow lits(t_i), body(B)$$

   varying $1 \leq i \leq n$. *Note that, when* $n = 0$, $rules(A)$ *would be empty, but we still fix* A' *to some new predicate name.*
3. *When* $A = B \cup C$ *we again fix* A' *to some new name and* $rules(A)$ *to include* $rules(B) \cup rules(C)$ *plus the rules:*

$$A'(\overline{Y}, X) \leftarrow B'(\overline{Y}, X), body(B)$$
$$A'(\overline{Y}, X) \leftarrow C'(\overline{Y}, X), body(B)$$

   Similarly, when $A = B \cap C$, *the rule to be added to* $rules(B) \cup rules(C)$ *would be:*

$$A'(\overline{Y}, X) \leftarrow B'(\overline{Y}, X), C'(\overline{Y}, X), body(B)$$

   and for $A = B \setminus C$ *we would include instead:*

$$A'(\overline{Y}, X) \leftarrow B'(\overline{Y}, X), not\ C'(\overline{Y}, X), body(B)$$

4. *When* $A = \{t \mid L;\ X_1 : A_1,\ \ldots,\ X_n : A_n\}$, *with* L *some set of literals, we again define* A' *as a new predicate name and* $rules(A)$ *would contain* $\bigcup_{i=1}^{n} rules(A_i)$ *plus the rule:*

$$A'(\overline{Y}, t') \leftarrow lits(t), L', lits(L), A'_1(X_1), \ldots, A'_n(X_n), body(B)$$

When A *does not occur in a rule, the translation is the same but removing* $\overline{Y}$ *and* $body(B)$ *everywhere.* □

Definition 4 (Translation of sort definitions). *The* translation of a sort definition *like* `sort` $s = B$ *is a set of rules consisting of* $rules(B)$ *plus the rule:*

$$s(X) \leftarrow B'(X)$$

If no definition for s *was given, then the extent of* s *is given by a unary fluent with the same name, and so, the translation would just contain the rule:*

$$s(X) \leftarrow holds(s, X, \texttt{true})$$

Definition 5 (Translation of a body). *The* translation of a rule body B *like* $B_1, \ldots, B_m, X_1 : A_1, \ldots, X_n : A_n$ *is the LP body denoted as* $body(B)$*:*

$$B'_1, \ldots, B'_m, lits(B_1), \ldots, lits(B_m), A'_1(X_1), \ldots, A'_n(X_n)$$

together with the set of rules called $rules(B) = \bigcup_{i=1}^{n} rules(A_i)$*.*

Definition 6 (Translation of a rule). *The* translation of an FLP rule r *is the set of rules* $\{r'\} \cup sets(r)$ *where* $sets(r)$ *is the union of all the* $rules(A)$ *for each set term* A *occurring in* r*, whereas* r' *is defined as:*

1. *If* r *is like* $\bot \leftarrow B$ *then* r' *is the constraint:*

$$\bot \leftarrow body(B)$$

2. *If* r *is like* $f(\bar{t}) = t_0 \leftarrow B$ *with some body* B*, then* r' *is the rule:*

$$holds(f, (\bar{t})', t'_0) \leftarrow lits(\bar{t}), lits(t_0), body(B)$$

3. *If* r *is like* $f(\bar{t}) \in A \leftarrow B$ *and* $\overline{Y}$ *occurs free in* A *then* r' *is the rule:*

$$1 \ \{ \ holds(f, (\bar{t})', V) \ : \ A'(\overline{Y}, V) \ \} \ 1 \leftarrow lits(\bar{t}), body(B) \tag{7}$$

□

With the inclusion of (7) we are assuming that the underyling ASP language contains weight constraints (like the backends [10, 11] for `lparse` [12]). A different option for a disjunctive LP solver (like `DLV` [13]) could be replacing (7) by the pair of rules:

$$\begin{aligned} holds(f, (\bar{t})', V) \vee \neg holds(f, (\bar{t})', V) &\leftarrow lits(\bar{t}), body(B) \\ \bot &\leftarrow not\ known(f, (\bar{t})', V) \end{aligned}$$

Definition 7 (Translation of a function definition). *The* translation of a function definition *like (1) consists of the union of rules from all its set terms* $rules(D_1) \cup \cdots \cup rules(D_n) \cup rules(R)$*, plus the following axioms:*

$$\begin{aligned} known(f, \overline{X}) &\leftarrow holds(f, \overline{X}, V), R'(V) \\ \neg holds(f, \overline{X}, W) &\leftarrow holds(f, \overline{X}, V), R'(V), R'(W), V \neq W \\ holds(f, \overline{X}, d') &\leftarrow not\ \neg holds(f, \overline{X}, d'), lits(d) \end{aligned}$$

□

Note that retrieving the function values from the final answer sets we obtain from Π' is straightforward: it suffices with considering $f(\overline{v}) = w$ for each atom $holds(f, \overline{v}, w)$ occurring in the asnwer set.

The FLP in Example 1 would lead to a logic program that contains, among other, the rules:

$$
\begin{aligned}
\bot \leftarrow\ & V_0 = V_1, X \neq Y, holds(next, X, V_0), \\
& holds(next, Y, V_1), \\
& node(X), node(Y) \\
s_0(X, Z) \leftarrow\ & holds(arc, X, Z, \texttt{true}), \\
& node(X), node(Z) \\
1\ \{\ holds(next, X, V)\ :\ s_0(X, V)\ \}\ 1 \leftarrow\ & node(X) \\
holds(visited, 0, \texttt{true}) & \\
holds(visited, Y, \texttt{true}) \leftarrow\ & holds(visited, X, \texttt{true}), \\
& holds(next, X, Y), \\
& node(X), node(Y) \\
\bot \leftarrow\ & holds(visited, X, \texttt{false}), node(X)
\end{aligned}
$$

4 Some ideas about grounding

The translation procedure explained before is still a preliminary approach that can be improved in many different ways, especially for avoiding the introduction of unneeded extra variables and for exploiting grounding features of ASP solvers.

A first improvement is detecting repeated occurrences of both scalar and set terms. For instance, when we apply our translation to a rule like:

$$\bot \leftarrow f(g(a)) > 0, g(a) < g(b)$$

we would generate two different auxiliary variables to capture the value of the two occurrences of the same term $g(a)$, leading to:

$$
\begin{aligned}
\bot \leftarrow\ & holds(g, a, V_0), holds(f, V_0, V_1), V_1 > 0 \\
& holds(g, a, V_2), holds(g, b, V_3), V_2 < V_3
\end{aligned}
$$

In many cases, this can be avoided by a simple syntactic check, although an improved procedure should also detect situations like $g(a + b)$ and $g(b + a)$.

A trivial improvement for simplifying the obtained logic program is converting boolean functions into predicates in the LP without need of reifying the truth value. This change is straightforward, although would have complicated the current description of the translation procedure.

Another consideration has to do with the problem of fixing the range of variables introduced in the LP by using domain predicates. Although we have said that all FLP variables are bounded by some definition like $X : A$, the set term A may depend on functions which can be defined by rules and default values. In other words, a set term in the FLP does not necessarily correspond to a domain predicate in the LP. However, an important observation should be made here: if we deal with a well-formed FLP, function and sort definitions are not cyclic. Therefore, we could reorganize the FLP in levels and apply the splitting

theorem [14] for a modular computation of the answer sets of its translation. At each level, we would handle a set of rules but the extent of all the involved set terms would have been decided at the previous level. Moreover, although we did not adopt any restriction in this sense, it seems more natural that the extent of sorts and function domains and ranges are fixed for all the obtained models (this is imposed for instance in [9]).

One more feature that can be exploited is that, when we use a variable as a function argument or value, we are implicitly limiting the range of that variable. For instance, in Example 1, we could actually remove the declaration of X and Y as global variables of range *node* because they are always used as a *node* argument for some function. For instance, the mere occurrence of term $next(X)$ in a literal points out that X must vary in range *node*. This can be important too for delimiting the range of the auxiliary variables we introduce in the translation. Depending on the ASP solver we use, we may be required to fix each new variable V with a domain predicate. Although this was not explained in the translation, it can be easily done by referring to the function range. As an example, a term like $f(\overline{t})$ translated as $holds(f, (\overline{t})', V)$ could be accompanied by the LP literal $R'(V)$ where R is the declared range[2] of function f.

Of course, this sorted nature of FLP descriptions could be exploited in a much better way for a future front-end that also accomplishes the grounding task (replacing, for instance, to `lparse`).

5 Reasoning about actions and change

The methodology of using Logic Programming as an underlying nonmonotonic formalism for representing action domains was originally proposed in [15], and has been followed in many cases, like the use of stable models for $\mathcal{A}$ and $\mathcal{B}$ languages [2], the use of a variant[3] of Clark's completion for $\mathcal{C}+$ language [3] or the use of both WFSX and stable models for language PAL [5]. In this section we briefly comment how to adapt the FLP syntax for a simpler representation of action domains, leading to a *Functional Action Language*.

First of all, in an actions reasoning 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.

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

[2] Note that as said before, if we apply splitting, the range of the function can be considered a domain predicate.

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

of f at the successor state, reserving f itself for referring to the current (or predecessor) 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, we will have as default its previous value. So, in some sense, the fact for the previous state $f(\bar{t}) = d$ acts like declaring $default(f') = d$ for f at the successor 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 corresponds 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:

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

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. Note that restricting ourselves to static fluents is the same than just dealing with functional logic programs. Thus, the tag `static` is optional, assuming that any function definition will correspond to a static function by default.

As inertial fluents would be the most frequent ones in actions scenarios, we will use the tag `fluent` to define them. Although their default value is fixed by inertia, declaring an extra default value could also be interesting. 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, when we want complete knowledge for that fluent. 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:

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

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 $press\_brake$ of pressing the brakes with a given intensity in interval $\{0, \ldots, 8\}$, 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 $press\_brake = 0$.

Example 2. **(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. We will use an action function $move(P) = Q$ meaning that we move the top disk of peg P to peg Q. Two action attributes[4] (located in the same state than $move$) called $source$ and $dest$ will represent the values of P and Q respectively. We include the definitions:

$$\begin{aligned}
&\texttt{sort} && peg = \{p_1, p_2, p_3\}\\
&\texttt{sort} && disk = \{1, \ldots, n\}\\
&\texttt{sort} && location = disk \cup \{ground\}\\
&\texttt{static} && size : disk \longrightarrow integer\\
&\texttt{fluent} && loc : disk \longrightarrow location\\
&\texttt{fluent} && top : peg \longrightarrow location = ground\\
&\texttt{fluent} && source : peg\\
&\texttt{fluent} && dest : peg\\
&\texttt{action} && move : peg \longrightarrow peg
\end{aligned}$$

together with the rules:

$$\begin{aligned}
move(P) &\in peg\\
source = P &\leftarrow move(P) = Q\\
dest = Q &\leftarrow move(P) = Q\\
top'(dest) &= top(source)\\
top'(source) &= loc(top(source))\\
loc'(top(source)) &= top(dest)\\
\bot &\leftarrow top(source) = ground\\
\bot &\leftarrow size(loc(D)) < size(D)
\end{aligned}$$

□

[4] See [16] for an interesting discussion of action attributes and their utility for an elaboration tolerant representation.

Note that we generated one $move(P) \in peg$ but we did not require asserting that only one peg is moved: this will just be guaranteed by the use of attribute *source* which cannot take more than one value.

Example 3. **(8-puzzle)**
The well known 8-puzzle planning problem could be modeled as follows. Let n denote the size of each size of the square puzzle (for 8-puzzle, $n = 3$).

$$\begin{aligned}
\texttt{sort}\ & coord = \{1, \ldots, n\} \\
\texttt{fluent}\ & hole_row : coord \\
\texttt{fluent}\ & hole_col : coord \\
\texttt{fluent}\ & board : coord \times coord \longrightarrow \{1, \ldots, n^2-1\} \cup \{empty\} = empty \\
\texttt{action}\ & move : \{up, down, left, right\}
\end{aligned}$$

The set of rules would be:

$$\begin{aligned}
hole_row' = hole_row + 1 &\leftarrow move = down \\
hole_row' = hole_row - 1 &\leftarrow move = up \\
hole_col' = hole_col + 1 &\leftarrow move = right \\
hole_col' = hole_col - 1 &\leftarrow move = left \\
board(hole_row, hole_col) &= empty \\
board'(hole_row, hole_col) &= board(hole_row', hole_col')
\end{aligned}$$

□

For space reasons, the translation of this extended syntax will be just briefly commented, although it can be easily guessed. First, all atoms like $holds(f, \bar{t}, V)$ will take an extra argument $holds(f, \bar{t}, V, I)$ where I points the situation number, i.e., the position of the considered state along the sequence of transitions. Second, standard axioms like:

$$holds(f, \bar{t}, V, I+1) \leftarrow holds(f, \bar{t}, V, I), not\ \neg holds(f, \bar{t}, V, I+1)$$

allow dealing with the inertia default. For non-static functions, we maintain the default value behavior:

$$holds(f, \overline{X}, d', I) \leftarrow not\ \neg holds(f, \overline{X}, d', I), lits(d)$$

although in the case of inertial fluents, I is fixed to 0. Finally, FLP rules where a non-static function or a primed function f' occurs are replicated indexing these functions for all the possible values of situation index I (we assume that the narrative length is fixed to some integer constant N). In this way, there is no real need for an explicit distinction between "static" rules and "dynamic" rules.

6 Discussion and future work

As said in the introduction, the current work can be seen as a proposal for a functional "variant" of ASP. Our motivation has been mainly practical, focusing on the availability of a translation from the functional notation to the ASP syntax for non-ground programs. The approach must be seen as a preliminary step towards a full functional logic programming system. Therefore, many interesting topics remain open and deserve future study. Clearly, one of them is finding a closer relation to the approaches from the area of Functional Logic Programming (FLP). To put an example, a similar treatment of sorts has been already done in the family of languages `OBJ` [17]: ideas like the definition of *subsorts* can be directly extrapolated here[5].

As pointed out by a referee, an interesting topic from FLP to be analyzed in this framework is the treatment of *higher order functions*: that is, writing generic code by using functions as arguments of other functions. It must be noticed that, although most FLP systems deal with this feature, in the ASP context, where general use of functors is avoided and domains are kept finite, introducing higher order functions is far from straightforward. For this reason, we have considered that this is out of the scope of the current work. However, a related recent approach for writing generic code for ASP is the idea of introducing templates [18] in ASP programs. This work can perhaps be used as a starting point in the future for the introduction of a limited kind of higher order functions in the current framework.

Future work will also involve replacing the translation into non-ground programs by a full grounding system (so that, for instance, it is possible to use `smodels` directly as a back-end). In principle, the sorted nature of the proposed language should help to improve the grounding process, avoiding in some cases the need for restricting variables with domain predicates, by guessing the sort of the variable by its use as argument or value of a given function. One more interesting topic for future analysis is the construction of a solver that directly deals with ground functional programs, so that the uniqueness of value of each function could be embodied in the search process.

Acknowledgements Thanks to Ramón P. Otero for his past guidance during the development of language PAL [5], which somehow motivated the current research and which had took some essential ideas[6] from his Expert Systems Tool *Medtool* [19]. This work was partially supported by CICyT project TIC-2003-9001-C02 and WASP (IST-2001-37004).

[5] Notice that, still, the current language allows elaborated definitions for the extension of any sort s, as we can use to this aim *any* set of program rules with head $s(X)$.

[6] The *unknown* operator used here was already present in Medtool formalism.

References

1. Cabalar, P., Lorenzo, D.: Logic programs with functions and default values. In: Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04). (2004)
2. Gelfond, M., Lifschitz, V.: Action languages. Linköping Electronic Articles in Computer and Information Science **3** (1998)
3. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153** (2004) 49–104
4. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, ii: the DLV^{K} system. Artificial Intelligence **144** (2003) 157–211
5. Cabalar, P., Cabarcos, M., Otero, R.P.: PAL: Pertinence action language. In: Proceedings of the 8th Intl. Workshop on Non-Monotonic Reasoning NMR'2000 (Collocated with KR'2000), Breckenridge, Colorado, USA (2000) (`http://xxx.lanl.gov/abs/cs.AI/0003048`).
6. Geffner, H. In: Functional STRIPS: a more flexible language for planning and problem solving. Kluwer (2000)
7. Naish, L.: Adding equations to NU-Prolog. In: Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming. Number 528 in Lecture notes in computer science, Passau, Germany, Springer-Verlag (August, 1991) 15–26
8. Hanus, M.: The integration of functions into logic programming: From theory to practice. Journal of Logic Programming **19&20** (1994) 583–628
9. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. In: Proc. of the 7th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). (2004)
10. : SMODELS web page `http://www.tcs.hut.fi/software/smodels/` (2005)
11. : CMODELS web page `http://www.cs.utexas.edu/users/tag/cmodels.html` (2005)
12. Syrjnen, T.: Lparse 1.0 user's manual (2005)
13. : DLV web page `http://www.dbai.tuwien.ac.at/proj/dlv/` (2005)
14. Lifschitz, V., Turner, H.: Splitting a logic program. In: International Conference on Logic Programming. (1994) 23–37
15. Gelfond, M., Lifschitz, V.: Representing action and change by logic programs. The Journal of Logic Programming **17** (1993) 301–321
16. Lifschitz, V.: Missionaries and cannibals in the causal calculator. In: Principles of Knowledge Representation and Reasoning: Proceedings of the 7th International Conference (KR'00). (2000) 85–96
17. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In Goguen, J., ed.: Applications of Algebraic Specification using OBJ. Cambridge (1993)
18. Ianni, G., Ielpa, G., Calimeri, F., Pietramala, A., Santoro, M.C.: Enhancing answer set programming with templates. In: Proceedings of the 10th International Workshop on Non-Monotonic Reasoning NMR2004, Whistler, BC, Canada (June, 2004)
19. : The Medtool project and related activities are described in web documents `http://www.dc.fi.udc.es/ai/medtool.html` (2001)