Logic Programs with Functions and Default Values

Pedro Cabalar[1] and David Lorenzo[1]

Dept. of Computer Science,
University of Corunna,
E-15071, A Coruña, SPAIN.
{cabalar,lorenzo}@dc.fi.udc.es

Abstract. In this work we reconsider the replacement of predicate-like notation by functional terms, using a similar syntax to Functional Logic Programming, but under a completely different semantic perspective. Our starting point comes from the use of logic programs for Knowledge Representation and Nonmonotonic Reasoning, especially under three well-known semantics for default negation: Clark's completion, stable models and well-founded semantics. The motivation for introducing functions in this setting arises from the frequent occurrence of functional dependences in the representation of many domains. The use of functions allows us to avoid explicit axiomatization (to ensure the uniqueness of value) and provides a more compact representation by nesting functional terms. This gets rid of a considerable amount of unnecessary variables, what may be exploited for the process of program grounding, something common in this type of applications of Logic Programming. From a representational point of view, the most interesting introduced feature is the possibility of replacing default negation by the concept of default value of a function. In the paper, we explore this idea of functions with default values, providing adapted versions of the three mentioned semantics for the functional case, and equivalent translations into logic programs.

1 Introduction

One of the uses of Logic Programming (LP) that is probably attracting more research interest is its application for practical knowledge representation, and particularly, for solving problems related to the area of Nonmonotonic Reasoning (NMR). 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* [19]) that allowed ignoring the operational aspects of Prolog, focusing instead on the use of default negation as a declarative tool for NMR. As a consequence of this application, a considerable number of extensions of the LP paradigm have emerged to cope with different knowledge representation issues: addition of a second negation [8, 1], nested operators [11], cardinality and weight constraints [15] or, more recently, 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, much in the style of the field of *Functional Logic Programming* [10] (FLP). Our aim, however, differs from FLP in that we tackle the functional extension from the NMR perspective. In this way, rather than being worried about the operational behavior of unification in FLP (usually related to the rewriting technique called *narrowing* [17]) we will omit the use of functors (like the list constructors) from the very beginning, so that we handle a finite set of ground terms. This assumption is usual, for instance, in the LP area of Answer Sets Programming [12] or in other related NMR approaches [9], where most practical applications involve a preliminary process of *program grounding* (i.e., replacement of variables by all their possible ground instances).

When representing many domains in NMR we face the typical situation where some relational symbol, for instance $father(x, y)$ actually represents a function $father(x) = y$. In this case, the program must be extended with several rules for explicitly asserting the uniqueness of value y in $father(x, y)$. Functions avoid this explicit axiomatization and, thanks to the possibility of nesting functional terms, allow removing a considerable number of unnecessary variables. Apart from a more comfortable representation, the most important representational feature we consider in this paper is the generalization of default negation to the notion of *default value* of each function. This concept is described under the three above-mentioned semantics.

The paper is organized as follows. Section 2 contains a brief recall of LP definitions, in order to make the paper self-contained. In the next section, we begin considering ground functional logic programs, where we handle 0-ary functions with default values. Besides, we describe the three mentioned semantics for these functional programs, and provide translations into LP. After that, we comment some aspects about expressiveness, showing that functional logic programs generalize normal and extended logic programming. The next section briefly describes non-ground programs with nested functions. Finally, Section 6 outlines some connections to related work and contains the conclusions of the paper.

2 Recall of Logic Programming

Given a finite set of atoms $\mathcal{H}$ called the *Herbrand Base*, we define a *program literal* as an atom $p \in \mathcal{H}$ or its default negation *not* p (being the latter also called *default literal*). By *normal logic program* (or just *program* for short) we mean a set of rules like:

$$H \leftarrow B_1, \dots, B_n$$

where H is an atom called the *head* of the rule, and the $B_i's$ are program literals. We will write B as an abbreviation of $B_1, \dots, B_n$ and call it the *body* of the rule. We will also allow a special atom $\bot \notin \mathcal{H}$ as rule head, standing for inconsistence and used for rejecting undesired models. When $n = 0$ we say that the rule is a *fact* and directly write H, omitting the arrow. A program P is said to be *positive* iff it contains no default negation.

A *propositional interpretation* I is any subset of $\mathcal{H}$. We use symbol $\models$ to represent classical propositional satisfaction, provided that $\leftarrow$, comma and *not* are understood as classical implication, conjunction and negation, respectively. Using this reading, the concept of (classical) *model* of a program is defined in the usual way. We also define the *direct consequences* operator T_P on interpretations, as follows: $T_P(I) \overset{\text{def}}{=} \{H \mid (H \leftarrow B) \in P \text{ and } I \models B\}$.

A well-known result [18] establishes that any positive program P has a least model, we will denote as $least(P)$. Furthermore, for positive programs, T_P is monotonic and its least fixpoint (computable by iteration on $\emptyset$) coincides with $least(P)$.

A *supported model* I of a program P is any fixpoint of T_P, that is, any $I = T_P(I)$. Supported models can also be computed as classical models of a propositional theory called *Clark's completion* [4] which can be easily obtained from P, although we omit here its description for brevity sake.

The *reduct* of a program P with respect to interpretation I, written P^I corresponds to: (1) removing from P all rules with a program literal *not* p such that $p \in I$; and (2) removing all the default negated literals from the remaining rules. Therefore, P^I is a positive program and has a least model $least(P^I)$. We represent this model as $\Gamma_P(I)$ or simply $\Gamma(I)$ when there is no ambiguity.

A *stable model* I of a program P is any fixpoint of Γ, that is: $I = \Gamma(I)$. Furthermore, operator Γ^2 (i.e., Γ applied twice) is monotonic and has a greatest and a least fixpoint, respectively represented as $gfp(\Gamma^2)$ and $lfp(\Gamma^2)$. The *well-founded model* (WFM) of a program P is a pair of interpretations (I, J) where $I = lfp(\Gamma^2)$ and $J = gfp(\Gamma^2)$. As $I \subseteq J$, we can see the WFM as a three-valued interpretation where atoms in I are *true* (or *founded*), atoms in $J - I$ *undefined*, and atoms not in J are *false* (or *unfounded*).

We will also consider *Extended Logic Programming* (ELP), that is, programs dealing with *explicit negation* '$\neg$'. For simplicity sake, however, we understand ELP as a particular case of normal programs where the Herbrand Base contains an atom "$\neg p$" per each atom p without explicit negation. Given an atom $A \in \mathcal{H}$, we write $\overline{A}$ to denote its complementary atom, that is $\overline{p} = \neg p$ and $\overline{\neg p} = p$. Under this setting, an interpretation I is said to be *consistent* iff it contains no pair of atoms p and $\neg p$. Consistent stable models for ELP receive the name of *answer sets*. As for (consistent) supported models for ELP, they can be computed by an adaptation of Clark's completion called *Literal Completion* [13].

In the case of WFS for ELP, some counterintuitive results have led to the need for a variation called WFSX (*WFS with eXplicit negation*) [16]. This semantics guarantees the so-called *coherence principle*: if an atom $A \in \mathcal{H}$ is founded in the WFM, its complementary atom $\overline{A}$ must be unfounded. In other words, explicit negation $\neg p$ must imply default negation *not* p. The definition of WFSX relies on the idea of seminormal programs. For any ELP rule $r = (H \leftarrow B)$, its *seminormal* version r_s is defined as $(H \leftarrow B, not\ \overline{H})$. Similarly, given program P, its seminormal version P_s consists of a rule r_s per each rule r in P. We write $\Gamma_s(I)$ to stand for $least(P_s^I)$, and say that Γ_s is not defined for an inconsistent I. When defined, operator $\Gamma\Gamma_s$ is monotonic. The *WFM* of a program P (under WFSX) is a pair of interpretations (I, J) such that $I = lfp(\Gamma\Gamma_s)$ and $J = \Gamma_s(I)$, provided that $lfp(\Gamma\Gamma_s)$ is defined (otherwise, the program is said to be *inconsistent*). It has been shown [16] that the WFM under WFSX satisfies the coherence principle.

3 Functional Logic Programs

3.1 Syntax

For describing the syntax of Functional Logic Programs, we begin considering a finite set of ground terms $\mathcal{F}$, that we can consider as 0-ary *function names*, together with a finite set of *constant values* $\mathcal{V}$. We will use letters $f, g, \dots$ to stand for elements of $\mathcal{F}$ and $v, w, \dots$ for constant values. The *definition* of each function $f \in \mathcal{F}$ is a sentence like:

$$f : R \quad [= d]$$

where $R \subseteq \mathcal{V}$ is called the *range* of f, and the declaration '$= d$' is optional, representing a *default value* $d \in R$. We will use the notation, $range(f) = R$ and, when defined, $default(f) = d$. As usual, range *boolean* stands for the set $\{\texttt{true}, \texttt{false}\}$. A *functional literal* (*F-literal* for short) is any expression like $f = v$, satisfying $v \in range(f)$. For simplicity sake, when $range(f) = boolean$ we may omit the '$= v$' and use a "standard" logical literal instead, so that:

$$f \overset{\text{def}}{=} f = \texttt{true}$$
$$\neg f \overset{\text{def}}{=} f = \texttt{false}$$

A *functional logic program* (*F-program* for short) is a finite set of rules like:

$$H \leftarrow B_1, \dots, B_n$$

where H and all the $B_i's$ are now F-literals. Again, H is called the *head* and can also be the special symbol $\bot$ that denotes inconsistence, whereas $B_1, \dots, B_m$ are the *body*, which will be abbreviated as B. When convenient, B can also be seen as a set of F-literals.

In order to describe the correspondence with normal logic programs, we will always bear in mind the translation of each F-literal L with shape $f = v$ into a ground atom L' of shape $holds(f, v)$. We generalize the use of the prime operator for any construction (expressions, rules, sets, etc) having the expected meaning: it replaces each occurring F-literal L by atom L'. A first important observation in this sense is that given the F-program P, the corresponding normal program P' is *positive* (that is, it contains no default negation).

3.2 Semantics: stable and supported models

An *F-interpretation* I is defined as a (possibly partial) function $I : \mathcal{F} \rightarrow \mathcal{V}$ where $I(f)$ can be undefined only if f has no default value and, otherwise,

$I(f) \in \mathit{range}(f)$. We alternatively represent an F-interpretation as a consistent set of F-literals, where by *consistent* we mean containing no pair of literals $f = v$ and $f = w$ with $v \neq w$, or the symbol $\bot$. An useful definition is the idea of *default portion* of an F-interpretation I:

$$\mathit{default}(I) \stackrel{\text{def}}{=} \{(f = d) \in I \mid d = \mathit{default}(f)\}$$

that is, the F-literals in I that correspond to assignments of default values.

An F-interpretation I *satisfies* a rule $H \leftarrow B$ iff $H \in I$ whenever $B \subseteq I$. An *F-model* of an F-program P is any F-interpretation I satisfying all the rules of P. An F-program P is said to be *consistent* iff it has some model.

As we did with T_P for normal logic programs, we can easily define an analogous *direct consequences* operator, $t_P(I)$, for F-programs as follows:

$$t_P(I) = \{H \mid (H \leftarrow B) \in P \text{ and } B \subseteq I\}$$

Note that $t_P(I)$ is just a set of F-literals which could be inconsistent or partial, even for functions with default values. In this way, we actually have the straightforward correspondence: $T_{P'}(I') = (t_P(I))'$. Therefore, T_P properties are also applicable for t_P:

Proposition 1. *Any (consistent) F-program P has a least F-model, written F-least(P).*

In the same way, for any program P, operator t_P is monotonic and has a least fixpoint which can be computed by iteration on the least set of F-literals $\emptyset$. Again, by adapting T_P results, we get:

Proposition 2. *If program P is consistent, its least F-model corresponds to the least fixpoint of t_P.*

Now, we can extend the idea of stable and supported models for the case of F-programs.

Definition 1 (Functional supported model). *A* functional supported model *of an F-program P is any F-interpretation I satisfying: $I = t_P(I) \cup \mathit{default}(I)$.*

Definition 2 (Functional stable model). *A* functional stable model *of a program P is any F-interpretation I satisfying $I = \gamma(I)$, where:*

$$\gamma(I) \stackrel{\text{def}}{=} \mathit{F\text{-}least}(P \cup \mathit{default}(I))$$

3.3 Translation into normal logic programs

When we interpret the previous definitions for stable and supported models of F-programs, it is interesting to note that, in both cases, we deal with a positive program that is "completed" somehow with the default information in I. We will see that this effect can be captured inside normal logic programs by the addition of the axiom rule schemata:

$$\bot \leftarrow holds(f,v), holds(f,w) \tag{1}$$

$$holds(f,d) \leftarrow not\ holds(f,v_1), \ldots, not\ holds(f,v_n) \tag{2}$$

for all function f, values $v, w \in range(f)$ with $v \neq w$, and $d = default(f)$, $\{v_1, \ldots, v_n\} = range(f) - \{d\}$. Axiom (1) simply gets rid of models where a function takes two different values. Axiom (2) allows assuming the default value d for any function f, whenever the function does not take any of the rest of possible values. Any propositional interpretation I' that classically satisfies (1) and (2) can be seen as an F-interpretation I, since it will not contain an inconsistent pair of literals (due to (1)) and will not be partial for functions with default value (due to (2)). This is important because, since any stable (or supported) model is also a classical model of P', axioms (1) and (2) will guarantee that it has an associated F-interpretation.

Assume we write P^* to stand for the normal logic program $P' \cup (1) \cup (2)$ obtained from the F-program P.

Theorem 1. *An F-interpretation I is a functional supported model of P iff I' is a supported model of P^*.*

Proof. First, note that $T_{P^*}(I')$ contains $T_{P'}(I')$, which corresponds to the translation of $t_P(I)$, as we had seen. The remaining atoms in $T_{P^*}(I')$ come from those heads of axioms (1) and (2) for the cases in which their body is true in I'. Clearly, by consistence of I as an F-interpretation, the body of (1) cannot be true in I'. As for (2), we must collect the set of $holds(f,d)$ for which no other value for f is included in I'. As I' cannot be partial for f, this is equivalent to collect all the $holds(f,d)$ such that $holds(f,d) \in I'$. But this is exactly the translation into atoms of the set $default(I)$ of functional literals. $\square$

The proof suggests that, for the case of supported models, we can replace axiom (2) by the simpler expression:

$$holds(f,d) \leftarrow holds(f,d) \tag{3}$$

For the case of stable models, we first prove that there exists a one-to-one correspondence between operators γ for F-program P and Γ for P^*.

Theorem 2. *Let I, J be a pair of sets of F-literals and P an F-program. Then $J = \gamma(I)$ for P iff $J' = \Gamma(I')$ for P^*.*

Proof. As a proof sketch, we outline a quite obvious correspondence between the reduct $(P^*)^{I'}$ and the F-program $P \cup default(I)$. Consider rule (2) for each function f with $default(f) = d$. If $holds(f,d) \notin I'$, since I' is not partial for f, there must exist some $holds(f,v_i) \in I'$ with $v_i \neq d$, and so, the whole rule (2) will be deleted when computing the reduct. On the other hand, if $holds(f,d) \in I'$, since I' is consistent, no other different $holds(f,v_i)$ belongs to I, and so we can delete all the default literals in (2), what simply amounts to the fact $holds(f, d)$ in the reduct. As a result, the reduct $(P^*)^{I'}$ is exactly the same program than $(P \cup \mathit{default}(I))' \cup (1)$. Finally, note that computing the least model of $(P^*)^{I'}$ is completely analogous to computing the least functional model of $P \cup \mathit{default}(I)$ (for instance, using the direct consequences operator in both cases), where axiom (1) just rules out inconsistent results in the logic program. □

Corollary 1. *An F-interpretation I is a functional stable model of P iff I' is a stable model of P^*.*

3.4 Well-founded semantics

The third type of semantics we will consider is the generalization of WFS for the case of F-programs. As we saw in Section 2, the main difference of WFS with respect to the two previous semantics is that, instead of considering multiple models for a program, we get a single model which may leave some atoms undefined. When we move to the functional case, the well-founded model would now have the shape of a pair of sets of F-literals (I, J), $I \subseteq J$. This means, in principle, that each F-literal $f = v$ could be founded, unfounded or undefined regardless the rest of values for function f. However, it is clear that, as happened with WFS for ELP, we must impose the restriction of consistency[1] for the set of founded literals I.

Since ELP can be seen as a particular case of F-programs (where all ranges are fixed to *boolean*), it is easy to find similar examples of possible counterintuitive behavior due to the non-satisfaction of the coherence principle. For instance, assume we try to define WFS for any F-program P by correspondence with the standard WFS for P^*.

Example 1. Let P_1 be the F-program:

$$a \leftarrow \neg a \tag{4}$$

$$b \leftarrow a \tag{5}$$

$$c \leftarrow b \tag{6}$$

$$\neg b \tag{7}$$

where $a, b, c : \mathit{boolean} = \texttt{false}$.

The WFM of P_1^* leaves both values of a undefined (due to cycle (4)) and, as a consequence, this undefinedness is propagated to literals $b = \texttt{true}$ and $c = \texttt{true}$ through rules (5) and (6). This result, however, seems counterintuitive in the presence of fact (7) which makes $b = \texttt{false}$ founded. As a result, we should expect that condition of rule (6) became *unfounded*, leaving c false by default.

The generalization of Alferes and Pereira's coherence principle for the case of arbitrary function ranges would be:

[1] Note that, on the other hand, the possibility of an "inconsistent" set of non-unfounded literals J must be allowed, since we could simultaneously have different undefined values for a same function.

Definition 3 (Coherence). *A pair (I, J) of sets of literals with $I \subseteq J$ and I consistent is said to be* coherent *iff for each $(f = v) \in I$, we have that $(f = w) \notin J$ for all $w \in range(f) - \{v\}$.*

In other words, a coherent pair (I, J) satisfies that, if a function value is founded, then the rest of values for that function are unfounded. As shown with Example 1, using the WFM of P^* as a guide for defining a functional WFS is not adequate for dealing with coherence. Instead, we could think about using a translation of P into ELP interpreted under WFSX.

Definition 4. *Given F-program P we define the extended logic program P^e as the set of rules in P' together with the axiom rule schemata:*

$$\neg holds(f, v) \leftarrow holds(f, w) \tag{8}$$

$$holds(f, d) \leftarrow not\ \neg holds(f, d) \tag{9}$$

where $v, w \in range(f)$ with $v \neq w$, and $d = default(f)$.

That is, P^e corresponds to P^* where axioms (1) and (2) are now replaced by (8) and (9). It is not difficult to see that these two new axioms are an alternative way of representing (2), provided that (1) is not needed when we deal with explicit negation. An important remark at this point is that program P^e actually handles an extended Herbrand Base $\mathcal{H}$, containing atoms of shape $holds(f, v)$ or $\neg holds(f, v)$. Therefore, when translating an F-interpretation I into a propositional interpretation, we must also describe the truth values for atoms like $\neg holds(f, v)$. Although this information is not explicitly included in I', axiom (8) allows us to consider it as implicit in the following way. For all function f and value v: $\neg holds(f, v) \in I'$ iff exists some $holds(f, w) \in I'$ with $w \neq v$.

Bearing in mind this new translation, we proceed now to define the adapted WFS for the functional case. For any F-program P and any consistent set of literals I, the program $P_s(I)$ (the s stands for "seminormal," by analogy with WFSX) is defined as follows:

$$P_s(I) \stackrel{\text{def}}{=} \{(f = v \leftarrow B) \in P \mid \text{such that no } f = w \in I, \text{ with } v \neq w\}$$

that is, we get those rules of P where the head literal is not contradictory with respect to another literal in I. We write γ_s to stand for γ with respect to program $P_s(I)$, that is:

$$\gamma_s(I) \stackrel{\text{def}}{=} least(P_s(I) \cup default(I))$$

As the definition of $P_s(I)$ requires I to be consistent, γ_s is not defined for an inconsistent I. The following result relating operators γ_s and Γ_s will allow us to inherit properties from WFSX for the case of F-programs:

Theorem 3. *Let I, J be a pair of sets of F-literals (with I consistent) and P an F-program. Then $J = \gamma_s(I)$ for P iff $J' = \Gamma_s(I')$ for P^e.*

Proof. The seminormal program P^e_s contains a rule r'_s:

$$holds(f, v) \leftarrow B', not\ \neg holds(f, v) \tag{10}$$

per each rule $r = (f = v \leftarrow B)$ in P, plus the seminormal version of rule schemata (8):

$$\neg holds(f, v) \leftarrow holds(f, w), not\ holds(f, v) \tag{11}$$

and rule schemata (9) (which is already, in fact, a seminormal rule). Now note that the reduct $(P^e_s)^{I'}$ will contain a rule $holds(f, v) \leftarrow B'$ per each r'_s satisfying $\neg holds(f, v) \notin I'$. As we saw for explicitly negated atoms in I', this means that there is no other $w \neq v$ such that $holds(f, w) \in I'$. So, we take rules whose head is consistent in I', what corresponds exactly to $P_s(I)$ in the functional case. $\square$

Consider now the composed operator $\gamma\gamma_s$. The last theorem, together with Theorem 2, allows us to import the next property from operator $\Gamma\Gamma_s$:

Corollary 2. *When defined, operator $\gamma\gamma_s$ is monotonic.*

Thus, we can compute a least fixpoint of $\gamma\gamma_s$, written $lfp(\gamma\gamma_s)$, by iteration on the least consistent set of literals $\emptyset$, provided that this iteration keeps consistence in each step.

Definition 5 (Functional Well-Founded Model). *For any F-program P, if $lfp(\gamma\gamma_s)$ is defined, then the* well-founded model *(WFM) of P is a pair of sets of F-literals (I, J) where:*

$$I \overset{\text{def}}{=} lfp(\gamma\gamma_s) \qquad\qquad J \overset{\text{def}}{=} \gamma_s(I)$$

When $lfp(\gamma\gamma_s)$ is not defined, we say that P is inconsistent.

As this definition is completely analogous to the WFM under WFSX, Theorem 3 also allows us to derive the following results:

Corollary 3. *The pair (I, J) is the WFM of a program P iff (I', J') is the WFM of P^e under WFSX.*

Corollary 4. *The WFM (I, J) of a F-program P is coherent.*

An alternative way of computing the functional WFM (perhaps more intuitive in operational terms) can be described by rewriting operations, by a direct analogy to Brass et al's method for WFS [2], further adapted for the case of WFSX in [3].

Definition 6 (Program Transformations). *Given an F-program P we define the following transformations:*

1. Fact simplification $\overset{\mathtt{F}}{\mapsto}$
For each fact $f = v$ in P:

   (a) remove all occurrences of $f = v$ from all bodies in P and
   (b) remove all rules with $f = w$ in their body, where $v \neq w$.
2. Default assumption $\overset{\mathtt{D}}{\mapsto}$
   For each function f with default$(f) = d$ and not occurring in any head of P, add the fact $f = d$.
3. Loop detection $\overset{\mathtt{L}}{\mapsto}$
   Let $P_{opt} \overset{\text{def}}{=} P \cup \{f = d \mid f \in \mathcal{F}$ and default$(f) = d\}$. Remove all rules in P containing a body literal $f = v$ not included in $lfp(t_{P_{opt}})$.

The meaning of the first two transformations is quite obvious. The loop detection step is needed to avoid deriving undefinedness from simple loops like $f = v \leftarrow f = v$. The intuitive behavior for this transformation is the following one. We construct an *optimistic* program P_{opt} where we assume that *all default values* can be added as facts. Then, we compute the consequences $lfp(t_{P_{opt}})$ of this program, ignoring possible inconsistencies (after all, we are being optimistic). If a literal is not among these optimistic consequences, it will be *unaccessible* at all, and so, we can remove rules that depend on that literal.

Given program P, we call *reminder* program, P_{rem}, to the result of the exhaustive application of transformations $\overset{\mathtt{F}}{\mapsto}$, $\overset{\mathtt{D}}{\mapsto}$ and $\overset{\mathtt{L}}{\mapsto}$ on P. For any F-program P we define its *trivial model* as a pair of F-literals (I, J) where J is the set of heads in P and $I \subseteq J$ the set of facts.

Proposition 3. *The functional WFM of a program P is the trivial model of the reminder program P_{rem}.*

4 Expressiveness of functional programs

The correspondence in the shape of F-programs with positive logic programs may incorrectly lead us to think that the expressive power of the current proposal is lower than full logic programming with default negation. In this section we show that this impression is wrong – the use of default values constitutes an alternative to default negation.

Contrarily to the previous translations, that showed how to convert functional programs into normal or extended logic programs, we will do here the opposite operation. In the case of a normal program, the conversion is quite straightforward. Assume we have a (ground) normal logic program P with Herbrand Base $\mathcal{H}$. Then, we would just declare each ground atom $p \in \mathcal{H}$ as a 0-ary boolean function $p : boolean = \mathtt{false}$, so that it is assumed to be false by default. Then, the only transformation needed in program rules would be replacing each default literal (*not* p) by the F-literal $\neg p$ (that is, $p = \mathtt{false}$). Notice how, explicit negation behaves as default negation when we have default value $\mathtt{false}$.

As for extended logic programs, the translation is slightly more complicated, since we need handling simultaneously default and explicit negation for each symbol p. This can be accomplished by the inclusion of extra atoms for representing default negation. In this case, we would declare all atoms $p \in \mathcal{H}$ as boolean functions p : *boolean* but *without* default value. Then, we add a new special function declared as:

$$know : \mathcal{H} \times boolean \longrightarrow boolean = \texttt{false}$$

which has two arguments, so that $know(p, v)$ is used to assert that we know that atom p takes value $v \in \{\texttt{true}, \texttt{false}\}$. Of course, we must also add the rule schemata:

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

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

$$\begin{aligned} not\ p &\overset{\text{def}}{=} \neg know(p, \texttt{true}) \\ not\ \neg p &\overset{\text{def}}{=} \neg know(p, \texttt{false}) \end{aligned}$$

At a first sight, the inclusion of this special function *know* as a predefined part of the language could seem interesting. 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, for instance, $\neg know(age, 10)$ pointing out that we do not have evidence that function *age* takes value 10. 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 $f = v$; or (b) we have that for all $v \in range(f)$, $\neg know(f, v)$. Thus, an operator like $unknown(f)$ for representing case (b) seems more convenient for practical purposes. This operator can be defined as:

$$unknown : \mathcal{H} \longrightarrow boolean = \texttt{true}$$

so that any $f \in \mathcal{F}$ will be unknown by default. Again, we would also handle the implicit rule schemata:

$$\neg unknown(f) \leftarrow f = v$$

5 Non-ground programs and nested functions

When we consider the use of variables, we will naturally require function arities greater than zero. Although, in principle, the same function name and arity could be used for an arbitrary set of ground functional terms, it will usually be more convenient to define a function domain, that specifies the types of all the possible arguments. The *definition* of a function is now a sentence like:

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

where the new $D_1 \times D_2 \times \cdots \times D_n$, with $n \geq 0$, is called the *domain* of f, written $domain(f)$, and being each D_i a finite set of constant values. Under this extension, a (ground) literal would simply have the shape $f(\overline{w}) = v$ where $v \in range(f)$ and $\overline{w}$ is a tuple of values $\overline{w} \in domain(f)$.

Consider the following program P_2 with the function definitions:

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

for some finite ranges *person*, *object*, and the set of rules (we omit the irrelevant facts database):

$$
\begin{aligned}
father(X) = Y &\leftarrow parent(Y, X), sex(Y) = \mathtt{male} && (12) \\
mother(X) = Y &\leftarrow parent(Y, X), sex(Y) = \mathtt{female} && (13) \\
offspring(X, Y) &\leftarrow parent(X, Y) && (14) \\
offspring(X, Y) &\leftarrow parent(X, Z), Z \neq Y,\ offspring(Z, Y) && (15) \\
grandpa(X, Y) &\leftarrow parent(Z, Y), father(Z) = X && (16) \\
likes(X, Y) &\leftarrow mother(X) = Y && (17) \\
\neg likes(X, Y) &\leftarrow mother(X) = M, mother(Y) = M, \\
&\qquad father(X) = F, father(Y) = G, \\
&\qquad nationality(A) = R, nationality(Y) = S, R \neq S && (18) \\
older(X, Y) &\leftarrow birth(X) = A, birth(Y) = B, A < B && (19) \\
older(X, Y) &\leftarrow offspring(X, Y) && (20) \\
\bot &\leftarrow older(X, Y), older(Y, X), X \neq Y && (21)
\end{aligned}
$$

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 (what 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 (17) says that any person likes his/her mother, whereas rule (18) says that X dislikes Y if they have the same mother, but their fathers are of different nationality. Using a default French nationality (`fr`) can be useful when dealing with inhabitants of Saint Malo, for instance. Relation *older* is partial, since it may be the case that we ignore the birth date of some ancestors.

As for the rules shape, variables are understood as abbreviations of all possible values and, as it can be observed, we allow arbitrary expressions relating variables (with arithmetic and relational operators) so that they describe the final combinations that generate a ground instance.

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 (16). Clearly, variable X is exclusively used for representing the value of $father(Z)$. Thus, it seems natural to replace this auxiliary variable by the functional term $father(Z)$, writing instead:

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

Similar steps could be applied to rules (17) and (19), respectively leading to:

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

However, the most interesting example would be rule (18) where we can save many unnecessary variables:

$$\begin{aligned} \neg likes(X,Y) \leftarrow\ & mother(X) = mother(Y), \\ & nationality(father(X)) \neq nationality(father(Y)) \end{aligned} \qquad (22)$$

Allowing this nested use of functions does not introduce any special difficulty, since a nested rule can always be easily unfolded back into the unnested version by a successive introduction auxiliary variables. Without entering into a formal description, consider instead as an example a rule like (22). We can go replacing each inner subexpression by a fresh variable, generating the sequence of transformations:

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

that ends up with a rule equivalent to (18).

6 Conclusion and Related Work

We have presented an extension of logic programs with functional terms for their use in Knowledge Representation and Nonmonotonic Reasoning. This extension provides a common framework for default reasoning with functions, declaring the concept of default values of functions under three different semantics adapted from Clark's completion, stable models and WFS.

There exist many connections to related work that deserve to be formally studied in future work. The closer approach inside Nonmonotonic Reasoning is probably the so-called formalism of *Causal Theories* [9] inspired by the causal logic in [13]. As a matter of fact, our description of the supported models semantics for functional programs is just a rephrasing of the idea of *causally explained models* previously introduced in that approach. Furthermore, the use of *multi-valued* symbols does not suppose a real novelty in Causal Theories and, in fact, the definition of default values is something usually done by the addition of expressions like rule (3). The only part of our proposal (when restricted to supported models) that would mean a contribution in this sense is the possibility of nesting functional terms described in Section 5, which is directly applicable to Causal Theories.

As for the relation to Functional LP, much work remain to be done yet. For instance, the use of default rules for FLP has already been studied in [14], although mostly analyzed from an operational perspective with respect to narrowing. It would be very interesting to establish a formal relationship between that work and some or all the semantics we propose in this paper (perhaps, due to the kind of programming paradigm, especially with WFS).

Other topics for future work include the extension of this framework for its use for Reasoning about Actions and Change. We expect that the definition of functions will allow efficiency improvements by restricting the grounding process, as happens for instance, with the functional extension [6] of the classical planning language STRIPS.

References

1. J. J. Alferes, L. M. Pereira, and T. C. Przymusinski. Classical negation in non-monotonic reasoning and logic programming. *Journal of Automated Reasoning*, 20(1):107–142, 1998.
2. S. Brass, J. Dix, B. Freitag, and U. Zukowski. Transformation-based bottom-up computation of the well-founded model. *Theory and Practice of Logic Programming*, 2001.
3. P. Cabalar. A rewriting method for well-founded semantics with explicit negation. In *International Conference on Logic Programming (ICLP'02)*, Copenhagen, Denmark, July 2002. Lecture Notes in Computer Science 2401:378-392.
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. In *Logic-Based Artificial Intelligence*. 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. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence Journal*, (to appear), 2003.
10. M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19,20:583–628, 1994.
11. V. Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
12. V. W. Marek and M. Truszczyński. Stable logic programming - an alternative logic programming paradigm. In K. R. Apt, V. W. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*. Springer-Verlag, 1999.
13. N. McCain and H. Turner. Causal theories of action and change. In *Proc. of the AAAI-97*, pages 460–465, 1997.
14. J. J. Moreno-Navarro. Extending constructive negation for partial functions in lazy functional-logic languages. In *Extensions of Logic Programming*, pages 213–227, 1996.
15. 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.
16. L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. In *Proceedings of the European Conference on Artificial Intelligence (ECAI'92)*, pages 102–106, Montreal, Canada, 1992. John Wiley & Sons.
17. J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.
18. M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.
19. 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.