TLP **11** (2–3): 203–233, 2011. © Cambridge University Press 2011
doi:10.1017/S1471068410000517 First published online 4 February 2011

Functional answer set programming

PEDRO CABALAR
Department of Computer Science, University of Corunna, Corunna, Spain
(*e-mail:* cabalar@udc.es)

submitted 21 June 2009; revised 17 October 2009; accepted 8 December 2009

Abstract

In this paper we propose an extension of Answer Set Programming (ASP) to deal with (possibly partial) evaluable functions. To this aim, we start from the most general logical counterpart of ASP, Quantified Equilibrium Logic (QEL), and propose a variant $\mathrm{QEL}^{=}_{\mathcal{F}}$ where the set of functions is partitioned into Herbrand functions (or *constructors*) and evaluable functions (or *operations*). We show how this extension has a direct connection to Scott's *Logic of Existence*, and introduce several useful derived operators, some of them directly borrowed from Scott's formalisation. Using this general framework for arbitrary theories, we proceed to focus on a syntactic subclass that corresponds to normal logic programs with evaluable functions and equality. We provide a translation of this class into function-free normal programs and consider a safety condition so that the resulting program is also safe, under the usual meaning in ASP. Finally, we also establish a formal comparison to Lin and Wang's approach (FASP) dealing with evaluable total functions.

KEYWORDS: answer set programming, equilibrium logic, partial functions, functional logic programming

1 Introduction

Since its introduction two decades ago (Gelfond and Lifschitz 1988), the paradigm of *answer set programming* (ASP) (Marek and Truszczyński 1999) has gradually become one of the most successful and practical formalisms for knowledge representation due to its flexibility, expressiveness and current availability of efficient solvers. This success can be easily checked by the continuous and plentiful presence of papers on ASP in the main conferences and journals on logic programming, knowledge representation and artificial intelligence during the last years. The declarative semantics of ASP has allowed many syntactic extensions that have simplified the formalisation of complex domains in different application areas like constraint satisfaction problems, planning or diagnosis.

In this paper, we consider one more syntactic extension that is an underlying feature in most application domains: the use of *(partial) evaluable functions*. Most ASP programs include some predicates that are nothing else than relational representations of functions from the original domain being modelled. For instance,

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

when modelling the typical educational example of family relationships, we may use a predicate $mother(X, Y)$ to express that X's mother is Y, but of course, we must add an additional constraint to ensure that Y is unique with respect to X, i.e. that the predicate actually acts as the function $mother(X) = Y$. In fact, it is quite common that first time Prolog students use this last notation as their first attempt. Functions are not only a natural element for knowledge representation but can also simplify in a considerable way ASP programs. Apart from avoiding constraints for uniqueness of value, the possibility of nesting functional terms like in $W = mother(father(mother(X)))$ allows a more compact and readable representation than the relational version $mother(X, Y), father(Y, Z), mother(Z, W)$ involving extra variables, which may easily mean a source of formalisation errors. Similarly, as we will see later, the use of partial functions can also save the programmer from including explicit conditions in the rule bodies to check that the rule head is actually defined.

The addition of functions to ASP is not new at all. In fact, there exist two different ways in which functions are actually understood. The first way of treating functions is followed by most of the existing work in the topic (like the general approaches, Syrjänen 2001; Bonatti 2004; Šimkus and Eiter 2007, or the older use of function *Result* for Situation Calculus inside ASP, Gelfond and Lifschitz 1993). These approaches treat functions in the same way as Prolog, that is they are just a way for *constructing* the Herbrand universe, and so they satisfy the unique names assumption – e.g. $mother(john) = mary$ is always false. A second way of treating functions is dealing with them as in predicate calculus, as done for instance in *functional logic programming* (Hanus 1994). The first and most general approach in this direction is due to the logical characterisation of ASP in terms of *Equilibrium Logic* (Pearce 1996) and, in particular, to its extension to first-order theories, *quantified equilibrium logic* (QEL) (Pearce and Valverde 2004). As a result of this characterisation, the concept of stable model is now defined for any theory from predicate calculus with equality. In fact, stable models can be alternatively described by a second-order logic operator (Ferraris *et al.* 2007) quite close to circumscription (McCarthy 1980), something that has been already used, for instance, to study strong equivalence for programs with variables (Lifschitz *et al.* 2007). Another alternative for ASP with (non-Herbrand) functions has been very recently presented in Lin and Wang (2008) and, as we will show later, can be seen as a particular case of the current approach, when we restrict to total functions.

As we will explain in the next section, we claim that the exclusive use of Herbrand functions and the currently proposed interpretation of equality in QEL or the requirement for functions to be total, as in Lin and Wang (2008), yield some counter-intuitive results when introducing functions for knowledge representation. In order to overcome these problems, we propose a variation of QEL that separates Herbrand functions (or constructors) from evaluable functions, as also done in logical characterisations (González-Moreno *et al.* 1999; Rodríguez-Artalejo 2001; Hanus 2007) of *Functional Logic Programming*. We further show how our semantics for partial functions has a direct relation to the *Logic of Existence* (or *E*-logic) proposed by Scott (1979).

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

The rest of the paper[1] is organised as follows. In the next section, we informally consider some examples of knowledge representation with functions in ASP, commenting the apparently expected behaviour and the problems that arise when using the current proposal for QEL. In Section 3, we introduce our variant called $\text{QEL}^{=}_{\mathscr{F}}$. Section 4 defines some useful derived operators, many of them directly extracted from E-logic and showing the same behaviour. In Section 5, we consider a syntactic subclass of logic programs with evaluable functions and Herbrand constants and show how they can be translated into (non-functional) normal logic programs afterwards. This includes a definition of safety that guarantees that the final translation results in a safe program, something crucial for the current ASP grounders. Section 6 establishes a formal comparison showing how Lin and Wang (2008) can be encoded into our functional logic programs by forcing functions to be total and also includes a discussion showing that $\text{QEL}^{=}_{\mathscr{F}}$ is more suitable for non-monotonic reasoning with functions. Finally, Section 7 contains a brief discussion about other related work, and Section 8 concludes the paper.

2 A motivating example

Consider the following simple scenario with a pair of rules.

Example 1
When deciding the second course of a given meal once the first course is fixed, we want to apply the following criterion: on Fridays, we repeat the first course as second one; the rest of week days, we choose $fish$ if the first was *pasta*. □

A straightforward encoding of these rules[2] into ASP would correspond to the program Π_1:

$$second(fish) \leftarrow first(pasta) \wedge \neg friday, \tag{1}$$

$$second(X) \leftarrow first(X) \wedge friday, \tag{2}$$

$$\bot \leftarrow first(X) \wedge first(Y) \wedge X \neq Y, \tag{3}$$

$$\bot \leftarrow second(X) \wedge second(Y) \wedge X \neq Y, \tag{4}$$

where the last two rules just represent that each course is unique, i.e. $first(salad)$ and $first(pasta)$ cannot be simultaneously true, for instance. In fact, these constraints immediately point out that $first$ and $second$ are 0-ary functions. A very naive attempt to use these functions for representing our example problem could be the pair of formulas Π_2:

$$second = fish \leftarrow first = pasta \wedge \neg friday, \tag{5}$$

$$second = first \leftarrow friday. \tag{6}$$

[1] This paper extends (Cabalar 2008) and improves it in many different ways. The most significant is, firstly, the inclusion of Section 6 with a complete formal comparison with Lin and Wang (2008) plus a small discussion on expressiveness. Secondly, the safety condition has been corrected (some cases dealing with equality were wrong) and improved to cover more cases. Thirdly, all proofs have been completed now and included in the Appendix.

[2] As a difference with respect to the typical ASP notation, we use $\neg$ to represent default negation, and instead of a comma, we use $\wedge$ to separate literals in the body.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Of course, Π_2 is not a logic program, but it can still be given a logic programming meaning by interpreting it under Herbrand models of QEL or the equivalent recent characterisation of stable models for first-order theories (Ferraris *et al.* 2007). Unfortunately, the behaviour of Π_2 in QEL with Herbrand models (and decidable equality) will be quite different from that of Π_1 by several reasons that can be easily foreseen. First of all, there exists now a qualitative difference between functions $first$ and *second* with respect to $fish$ and *pasta*. For instance, while it is clear that $fish = pasta$ must be false, we should allow $second = first$ to cope with our Fridays criterion. If we deal with Herbrand models or unique names assumption, the four constants would be pairwise different and (5) would be equivalent to $\bot \leftarrow \bot$, that is a tautology, whereas (6) would become the constraint $\bot \leftarrow friday$.

Even after limiting the unique names assumption only to constants $fish$ and *pasta*, new problems arise. For instance, the approaches in Ferraris *et al.* (2007), Pearce and Valverde (2004), Lifschitz *et al.* (2007) and Lin and Wang (2008) deal with complete functions and the axiom of *decidable equality*:

$$x = y \vee \neg(x = y). \tag{7}$$

This axiom is equivalent to $x = y \leftarrow \neg\neg(x = y)$ which informally implies that we always have a justification to assign any value to any function. Thus, for instance, if it is not Friday and we do not provide any information about the first course, i.e. no atom $first(X)$ holds, then Π_1 will not derive any information about the second course, that is no atom $second(X)$ is derived. In Π_2, however, functions $first$ and *second* must *always* have a value, which is further justified in any stable model by (7). As a result, we get that a possible stable model is, for instance, $first = fish$ and $second = pasta$. A related problem of axiom (7) is that it allows rewriting a rule like (5) as the constraint

$$\bot \leftarrow first = pasta \wedge \neg friday \wedge \neg(second = fish)$$

whose relational counterpart would be

$$\bot \leftarrow first(pasta) \wedge \neg friday \wedge \neg second(fish) \tag{8}$$

and whose behaviour in logic programming is very different from the original rule (1). As an example, while $\Pi_1 \cup \{first(pasta)\}$ entails $second(fish)$, the same program after replacing (1) by (8) has no stable models.

Finally, even after removing decidable equality, we face a new problem that has to do with directionality in the equality symbol when used in the rule heads. The symmetry of '=' allows rewriting (6) as

$$first = second \leftarrow friday \tag{9}$$

that in a relational notation would be the rule

$$first(X) \leftarrow second(X) \wedge friday, \tag{10}$$

which, again, has a very different meaning from the original (2). For instance, $\Pi_1 \cup \{friday, second(fish)\}$ does not entail anything about the first course, whereas if we replace (2) by (10) in this program, we obtain $first(fish)$. This is counter-intuitive, since our program was intended to derive facts about the second course and not

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

about the first one. To sum up, we will need some kind of new directional operator to specify the function value in a rule head.

3 Quantified equilibrium logic with evaluable functions

The definition of propositional equilibrium logic (Pearce 1996) relied on establishing a selection criterion on models of the intermediate logic called the logic of *here-and-there* (HT) (Heyting 1930). The first-order case (Pearce and Valverde 2004) followed similar steps, introducing a quantified version of HT called SQHT$^{=}$ that stands for *quantified HT with static domains*[3] *and equality*. In this section, we describe the syntax and semantics of a variant, called SQHT$^{=}_{\mathscr{F}}$, for dealing with evaluable functions.

We begin by defining a first-order language by its *signature*, a tuple $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ of disjoint sets, where $\mathscr{C}$ and $\mathscr{F}$ are sets of *function names* and $\mathscr{P}$ is a set of *predicate names*. We assume that each function (resp. predicate) name has the form f/n, where f is the function (resp. predicate) symbol and $n \geqslant 0$ is an integer denoting the number of arguments (or *arity*). Elements in $\mathscr{C}$ will be called *Herbrand functions* (or *constructors*), whereas elements in $\mathscr{F}$ will receive the name of *evaluable*[4] *functions* (or *operations*). The sets $\mathscr{C}_0$ (Herbrand constants) and $\mathscr{F}_0$ (evaluable constants) respectively represent the elements of $\mathscr{C}$ and $\mathscr{F}$ with arity 0. We assume that $\mathscr{C}_0$ contains at least one element.

First-order formulas are built up in the usual way, with the same syntax of classical predicate calculus with equality $=$. We assume that $\neg\varphi$ is defined as $\varphi \rightarrow \bot$, whereas $x \neq y$ just stands[5] for $\neg(x = y)$. An atom like $t = t'$ is called an *equality atom*, whereas an atom like $p(t_1, \ldots, t_n)$ with $n \geqslant 0$ for any predicate p different from equality receives the name of *predicate atom*. Given any set of functions $\mathscr{A}$, we write $Terms(\mathscr{A})$ to stand for the set of ground terms built from functions (and constants) in $\mathscr{A}$. In particular, the set of all possible ground terms for signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ would be $Terms(\mathscr{C} \cup \mathscr{F})$, whereas the subset $Terms(\mathscr{C})$ will be called the *Herbrand universe* of $\mathscr{L}$. The *Herbrand base* $HB(\mathscr{C}, \mathscr{P})$ is a set containing all atoms that can be formed with predicates in $\mathscr{P}$ and terms in the Herbrand universe, $Terms(\mathscr{C})$.

From now on, we assume that all free variables are implicitly universally quantified. We use letters x, y, z and their capital versions to denote variables, t to denote terms and letters c, d to denote ground terms. Boldface letters like $\mathbf{x}, \mathbf{t}, \mathbf{c}, \ldots$ represent tuples (in this case of variables, terms and ground terms, respectively). The corresponding semantics for SQHT$^{=}_{\mathscr{F}}$ is described as follows.

Definition 1 (*state*)
A *state* for a signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ is a pair (σ, A), where $A \subseteq HB(\mathscr{C}, \mathscr{P})$ is a set of atoms from the Herbrand base and $\sigma : Terms(\mathscr{C} \cup \mathscr{F}) \rightarrow Terms(\mathscr{C}) \cup \{\mathtt{u}\}$ is a function assigning to any ground term in the language some ground term in the

[3] The term *static domain* refers to the fact that the universe is shared among all worlds in the Kripke frame.

[4] In Hanus (2007), elements of $\mathscr{F}$ are called *defined functions* instead – we avoid this terminology because it could be mistakenly understood as the opposite of being *undefined* or partial.

[5] We hope that, depending on the context, the reader will be aware of the different use of symbols '$=$' and '$\neq$' as formulas in the language from their standard use in the semantic metalanguage.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Herbrand universe or the special value $\mathtt{u} \notin Terms(\mathscr{C} \cup \mathscr{F})$ (standing for *undefined*). Function σ must satisfy:

(i) $\sigma(c) = c$ for all $c \in Terms(\mathscr{C})$.

(ii) $$\sigma(f(t_1,\ldots,t_n)) = \begin{cases} \mathtt{u}, & \text{if} \quad \sigma(t_i) = \mathtt{u} \text{ for some } i = 1,\ldots,n, \\ \sigma(f(\sigma(t_1),\ldots,\sigma(t_n))), & \text{otherwise.} \end{cases}$$

□

As can be seen, our domain is exclusively formed by the terms from the Herbrand universe, $Terms(\mathscr{C})$. These elements are used as arguments of ground atoms in the set A that collects the *true* atoms in the state. Similarly, the value of any functional term is an element from $Terms(\mathscr{C})$, excepting the cases in which operations are left undefined (i.e. they are *partial* functions) – if so, then they are assigned the special element $\mathtt{u}$ (different from any syntactic symbol) instead. Condition (i) asserts, as expected, that any term c from the Herbrand universe has the fixed valuation $\sigma(c) = c$. Condition (ii) establishes two important restrictions. On the one hand, it guarantees that a functional term with an undefined argument becomes undefined in its turn[6]. On the other hand, Condition (ii) also guarantees that functions preserve their interpretation through subterms – for instance, if we have $\sigma(f(a)) = c$, we expect that $\sigma(g(f(a))$ and $\sigma(g(c))$ coincide. It is easy to see that (ii) implies that σ is completely determined by the values it assigns to all terms like $f(\mathbf{c})$, where f is any operation and $\mathbf{c}$ is a tuple of elements in $Terms(\mathscr{C})$.

Definition 2 (*Ordering $\preceq$ among states*)
We say that state $S = (\sigma, A)$ is *smaller* than state $S' = (\sigma', A')$, written $S \preceq S'$, when both

(i) $A \subseteq A'$,
(ii) $\sigma(d) = \sigma'(d)$ or $\sigma(d) = \mathtt{u}$, for all $d \in Terms(\mathscr{C} \cup \mathscr{F})$. □

We write $S \prec S'$ when the relation is strict, that is $S \preceq S'$ and $S \neq S'$. The intuitive meaning of $S \preceq S'$ is that the former contains *less information* than the latter so that any true atom or defined function value in S must hold in S'.

Definition 3 (*HT-interpretation*)
An *HT interpretation* I for a signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ is a pair of states $I = \langle S^h, S^t \rangle$ with $S^h \preceq S^t$. □

The superindices h, t represent two worlds (respectively standing for *HT*) with a reflexive ordering relation further satisfying $h \leqslant t$. An interpretation like $\langle S^t, S^t \rangle$ is said to be *total*, referring to the fact that both states contain the same information[7].

Given an interpretation $I = \langle S^h, S^t \rangle$, with $S^h = (\sigma^h, I^h)$ and $S^t = (\sigma^t, I^t)$, we define when I *satisfies* a formula φ at some world $w \in \{h, t\}$, written $I, w \models \varphi$, inductively as follows:

[6] Using functional logic programming terminology, this means that functions are *strict*, as opposed to non-strict functions with lazy evaluation.

[7] Note that by *total* we do not mean that functions cannot be left undefined. We may still have some term d for which $\sigma^t(d) = \mathtt{u}$.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

- $I, w \models p(t_1, \ldots, t_n)$ if $p(\sigma^w(t_1), \ldots, \sigma^w(t_n)) \in I^w$,
- $I, w \models t_1 = t_2$ if $\sigma^w(t_1) = \sigma^w(t_2) \neq \mathfrak{u}$,
- $I, w \not\models \bot$; $I, w \models \top$,
- $I, w \models \alpha \wedge \beta$ if $I, w \models \alpha$ and $I, w \models \beta$,
- $I, w \models \alpha \vee \beta$ if $I, w \models \alpha$ or $I, w \models \beta$,
- $I, w \models \alpha \rightarrow \beta$ if for all $w' \geqslant w$: $I, w' \not\models \alpha$ or $I, w' \models \beta$,
- $I, w \models \forall x\ \alpha(x)$ if for each $w' \geqslant w$ and each $c \in Terms(\mathscr{C})$: $I, w' \models \alpha(c)$,
- $I, w \models \exists x\ \alpha(x)$ if for some $c \in Terms(\mathscr{C})$: $I, w \models \alpha(c)$. □

An important observation is that the first condition above implies that an atom with an undefined argument will always be valuated as false since, by definition, $\mathfrak{u}$ never occurs in ground atoms of I^h or I^t. Something similar happens with equality: $t_1 = t_2$ will be false if any of the two operands, or even both, is undefined. As usual, we say that I is a *model* of a formula φ, written $I \models \varphi$, when $I, h \models \varphi$. Similarly, I is a *model* of a theory Γ when it is a model of all of its formulas.

From the definition of $\neg$ as derived operator, we can easily check as follows.

Proposition 1
$I, w \models \neg\varphi$ iff $I, t \not\models \varphi$. □

Non-monotonicity is obtained by the next definition, which introduces the idea of equilibrium models for $\mathrm{SQHT}^{=}_{\mathscr{F}}$.

Definition 4 (*Equilibrium model*)
A model $\langle S^t, S^t \rangle$ of a theory Γ is an *equilibrium model* if there is no strictly smaller state $S^h \prec S^t$ that $\langle S^h, S^t \rangle$ is also model of Γ. □

The QEL with evaluable functions ($\mathrm{QEL}^{=}_{\mathscr{F}}$) is the logic induced by the $\mathrm{SQHT}^{=}_{\mathscr{F}}$ equilibrium models.

For space reasons, we describe $\mathrm{SQHT}^{=}$ (resp. QEL) as a particular instance of $\mathrm{SQHT}^{=}_{\mathscr{F}}$ (resp. $\mathrm{QEL}^{=}_{\mathscr{F}}$). It can be easily checked that this description is equivalent to the one in (Pearce and Valverde 2008). The syntax for $\mathrm{SQHT}^{=}$ is the same as for $\mathrm{SQHT}^{=}_{\mathscr{F}}$ (that is predicate calculus with equality) but starting from a signature $\langle \mathscr{F}, \mathscr{P} \rangle$ where no distinction is made among functions in set $\mathscr{F}$. Each $\mathrm{SQHT}^{=}$ interpretation for signature $\langle \mathscr{F}, \mathscr{P} \rangle$ further deals with a universe domain, a set $U \neq \emptyset$ which is said to be *static*, that is common to both worlds h and t. To capture this in $\mathrm{SQHT}^{=}_{\mathscr{F}}$, we can just use signature $\langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ and define $\mathscr{C}$ as a set of constant names, one c' per each individual $c \in U$. The most important feature of $\mathrm{SQHT}^{=}$ interpretations is that they satisfy the axiom $t = t$ for any term t. In other words, any ground term $d \in Terms(\mathscr{C} \cup \mathscr{F})$ is defined $\sigma^h(d) \neq \mathfrak{u}$ and, in fact, by construction of interpretations, this also means $\sigma^h(d) = \sigma^t(d)$. As a result, $\mathrm{SQHT}^{=}$ actually uses a unique σ function for both worlds h and t and interpretations can be represented instead as $\langle \sigma, I^h, I^t \rangle$. Under this restriction, it is easy to see that decidable equality $t_1 = t_2 \vee t_1 \neq t_2$ is a valid formula.

Herbrand models from $\mathrm{SQHT}^{=}$ and signature $\langle \mathscr{C}, \mathscr{P} \rangle$ can be easily captured by just considering $\mathrm{SQHT}^{=}_{\mathscr{F}}$ interpretations for signature $\langle \mathscr{C}, \emptyset, \mathscr{P} \rangle$. Finally, the models' selection criterion in the definition of equilibrium models need not be modified. Since $\sigma^h = \sigma^t = \sigma$ and all terms are defined, the $\preceq$ ordering relation among states in QEL actually amounts to a simple inclusion of sets of ground atoms.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

4 Useful derived operators

From the SQHT$^{=}_{\mathscr{F}}$ semantics, it is easy to see that the formula $(t = t)$, usually included as an axiom for equality, is not valid in SQHT$^{=}_{\mathscr{F}}$. In fact, $I, w \models (t = t)$ iff $\sigma^w(t) \neq \mathfrak{u}$, that is term t is defined. In this way, we can introduce *existence* operator[8] of Scott (1979) in a standard way: $E\ t \stackrel{\text{def}}{=} (t = t)$. Condition (ii) in Definition 1 implies the *strictness* condition of E-logic, formulated by the axiom $E\ f(t) \rightarrow E\ t$. As happens with $(t = t)$, the substitution axiom for functions

$$t_1 = t_2 \rightarrow f(t_1) = f(t_2)$$

is not valid, since it may be the case that the function is undefined. However, the following weaker version is an SQHT$^{=}_{\mathscr{F}}$ tautology:

$$t_1 = t_2 \wedge E\ f(t_1) \rightarrow f(t_1) = f(t_2).$$

Usual axioms for equality that are valid in SQHT$^{=}_{\mathscr{F}}$ are, for any predicate P, as follows:

$$t_1 = t_2 \rightarrow t_2 = t_1,$$
$$t_1 = t_2 \wedge t_2 = t_3 \rightarrow t_1 = t_3,$$
$$t_1 = t_2 \wedge P(t_1) \rightarrow P(t_2).$$

At this point, it is perhaps convenient to introduce a few terms to talk about particular types of functions. We say that an evaluable function f is *decidable* under a given interpretation I, when I satisfies the excluded middle axiom

$$f(\mathbf{t}) = t' \vee f(\mathbf{t}) \neq t', \tag{11}$$

and we say that f is *decidable* in a given theory when it is decidable under any of its models. The models of (11) correspond to interpretations where $\sigma^h(f(\mathbf{c})) = \sigma^t(f(\mathbf{c}))$ for any tuple of elements $\mathbf{c} \in Terms(\mathscr{C})$, that is function f has the *same* interpretation in both worlds, and so it somehow behaves 'classically'. As example of decidable functions, think about integer arithmetic operations like $+$, $-$, $\times$ or $\div$, for which we expect a fixed interpretation in worlds h and t with their usual meaning. Of course, an evaluable function is always decidable under any equilibrium model I of a theory Γ, since $\sigma^h = \sigma^t$ in that case, but this does not necessarily mean that (11) holds for all SQHT$^{=}_{\mathscr{F}}$ models of Γ.

A function is said to be *total* under an interpretation I when I satisfies

$$E\ \mathbf{t} \rightarrow E\ f(\mathbf{t}) \tag{12}$$

and called *partial* under I otherwise. We say that a decidable function f is *total* in a theory Γ if it is total under any of its models; otherwise it is *partial* in Γ. Semantically, total functions satisfy $\sigma^w(f(\mathbf{c})) \neq \mathfrak{u}$ for any tuple of terms $\mathbf{c} \in Terms(\mathscr{C})$ and any world w, while partial functions do not. From this, and the SQHT$^{=}_{\mathscr{F}}$ semantics, it

[8] Contrarily to the original Scott's E-logic, variables in SQHT$^{=}_{\mathscr{F}}$ are always defined. This is not an essential difference: terms may be left undefined instead, and so most theorems, like $(x = y) \rightarrow (y = x)$, are expressed here using metavariables for terms $(t_1 = t_2) \rightarrow (t_2 = t_1)$.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

can be observed that (12) actually implies (11), that is a total function is always decidable. The opposite does not necessarily hold. Back to the example, under their usual interpretation, $+$, $-$ and $\times$ are total functions, whereas $\div$ is partial, although, as stated earlier, still decidable; in particular the formula $E\ x \rightarrow E\ x \div 0$ is always false, since a variable x is always defined, $E\ x$, whereas $\sigma^w(x \div 0) = \mathfrak{u}$. Again, a function may be total under equilibrium models of a theory Γ without being so under any $\mathrm{SQHT}^{=}_{\mathscr{F}}$ model of Γ.

Similarly to these distinctions among types of functions, we have now several types of equalities and inequalities. In E-logic, there exists a second and weaker equality (which Scott called *equivalence*) that can be defined as $t_1 \equiv t_2 \stackrel{\text{def}}{=} (E\ t_1 \vee E\ t_2) \rightarrow t_1 = t_2$. In other words, t_1 and t_2 have the same defined value, provided that any of them is defined. This equality is perhaps not so interesting for knowledge representation and its inclusion in logic programs but may have a crucial importance when studying properties of programs, like for instance, strongly equivalent[9] transformations. In this sense, $t_1 \equiv t_2$ means that t_1 can be replaced by t_2 and vice versa, that is they have the *same behaviour*. For instance, the following valid formula

$$t_1 \equiv t_2 \rightarrow f(t_1) \equiv f(t_2)$$

allows us replacing $f(t_1)$ by $f(t_2)$ when we know that t_1 can be replaced by t_2. Note that if we just use equality for that purpose

$$t_1 = t_2 \rightarrow f(t_1) = f(t_2),$$

we would be forcing f to become a total function, since taking t_2 to be t_1 above we actually get $E\ t_1 \rightarrow E\ f(t_1)$, and this is not what we want. To understand the difference, note that $t = 0 \rightarrow (1 \div t) = (1 \div 0)$ is always false because $(1 \div 0)$ is undefined, whereas $t \equiv 0 \rightarrow (1 \div t) \equiv (1 \div 0)$ is valid.

To represent the *difference* between two terms, we may also have several alternatives. The straightforward one is just $\neg(t_1 = t_2)$ or abbreviated $t_1 \neq t_2$. However, this formula can be satisfied when any of the two operands is undefined. We may sometimes want to express a stronger notion of difference that behaves as a positive formula (this is usually called *apartness* in the intuitionistic literature; Heyting 1956). In our case, we are especially interested in an apartness operator $t_1 \# t_2$ where both arguments are required to be defined:

$$t_1 \# t_2 \stackrel{\text{def}}{=} E\ t_1 \wedge E\ t_2 \wedge \neg(t_1 = t_2).$$

The semantic effect of this operator is that $I, w \models t_1 \# t_2$ iff $\sigma^w(t_1) \neq \mathfrak{u}$, $\sigma^w(t_2) \neq \mathfrak{u}$ and $\sigma^w(t_1) \neq \sigma^w(t_2)$. To understand its meaning, consider the difference between $\neg(King(France) = LouisXIV)$ and $King(Spain) \# LouisXIV$. The first expression means that we cannot prove that the King of France is Louis XIV, but can mean that France does not have a king. The second expression means that we can prove that the King of Spain (and so such a concept exists) is not Louis XIV.

[9] Two theories Γ_1, Γ_2 are said to be *strongly equivalent* when, for any theory Γ, the equilibrium models of $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ coincide.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

The next operator we introduce has to do with definedness of rule heads in logic programs (as far as we know, it has not been considered in the literature). The inclusion of a formula in the consequent of an implication may have an undesired effect when thinking about its use as a rule head. For instance, consider the rule $visited(next(x)) \leftarrow visited(x)$ and assume that we have the fact $visited(1)$, but there is no additional information about $next(1)$. We would expect that the rule above does not yield any particular effect on $next(1)$. Unfortunately, as $visited(next(1))$ must be true, the function $next(1)$ must become defined and, as a collateral effect, it will be assigned some arbitrary value, say $next(1) = 10$, so that $visited(10)$ is made true. To avoid this problem, we will use a new operator $:\!-$ to define a different type of implication where the consequent is only forced to be true when all the functional terms that are 'necessary to build' the atoms in the consequent are defined. Given a term t, we define its set of *structural arguments* $Args(t)$ as follows:

- $Args(t) \stackrel{\text{def}}{=} \{t_1, \ldots, t_n\}$ if t has the form $f(t_1, \ldots, t_n)$ for any evaluable function $f/n \in \mathscr{F}$.
- $Args(t) \stackrel{\text{def}}{=} t$ otherwise.

We extend this definition for any atom A so that its set of structural arguments $Args(A)$ corresponds to

$$Args(P(t_1, \ldots, t_n)) \stackrel{\text{def}}{=} \{t_1, \ldots, t_n\},$$
$$Args(t = t') \stackrel{\text{def}}{=} Args(t) \cup Args(t').$$

In our previous example, $Args(visited(next(x))) = \{next(x)\}$. Notice that for an equality atom $t = t'$, we do not consider $\{t, t'\}$ as arguments as done for the rest of predicates but go down one level instead, considering $Args(t) \cup Args(t')$ in its turn. For instance, if A is the atom $friends(mother(x), mother(y))$, then $Args(A)$ would be $\{mother(x), mother(y)\}$, whereas for an equality atom A' like $mother(x) = mother(y)$, $Args(A') = \{x, y\}$. We define $[\varphi]$ as the result of replacing each atom A in φ by the conjunction of all $E\ t \rightarrow A$ for each $t \in Args(A)$. We can now define the new implication operator as $\varphi :\!- \psi \stackrel{\text{def}}{=} \psi \rightarrow [\varphi]$. Back to the example, if we use now $visited(next(x)) :\!- visited(x)$, we obtain, after applying the previous definitions, that it is equivalent to

$$\begin{aligned} & visited(x) \rightarrow [visited(next(x))], \\ \leftrightarrow\ & visited(x) \rightarrow (E\ next(x) \rightarrow visited(next(x))), \\ \leftrightarrow\ & visited(x) \wedge E\ next(x) \rightarrow visited(next(x)). \end{aligned}$$

Another important operator will allow us to establish a direction in a rule head assignment – remember the discussion about distinguishing between (6) and (9) in Section 2. We define this *assignment* operator as follows:

$$f(\mathbf{t}) := t' \stackrel{\text{def}}{=} E\ t' \rightarrow f(\mathbf{t}) = t'.$$

Now, our Example 1 would be encoded with the pair of formulas

$$second := fish :\!- first = pasta \wedge \neg friday, \qquad second := first :\!- friday,$$

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

which, after some elementary transformations, lead to

$$second = fish \leftarrow first = pasta \wedge \neg friday,$$
$$second = first \leftarrow E\ first \wedge friday.$$

Using these operators, a compact way to fix a default value t' for a function $f(\mathbf{t})$ would be $f(\mathbf{t}) := t' \ :\!-\ \neg(f(\mathbf{t}) \# t')$. Finally, we introduce a non-deterministic choice assignment with the following set-like expression:

$$f(\mathbf{t}) \in \{x \mid \varphi(x)\}, \tag{13}$$

where $\varphi(x)$ is a formula (called the set *condition*) that contains the free variable x. The intuitive meaning of (13) is self-explanatory. As an example, the formula $a \in \{x \mid \exists y\ Parent(x, y)\}$ means that a should take a value among those x that are parents of some y. Expression (13) is defined as the conjunction of

$$\forall x\ (\varphi(x) \rightarrow f(\mathbf{t}) = x \vee f(\mathbf{t}) \neq x), \tag{14}$$
$$\neg\exists x\ (\varphi(x) \wedge f(\mathbf{t}) = x) \rightarrow \bot. \tag{15}$$

Other typical set constructions can be defined in terms of (13):

$$f(\mathbf{t}) \in \{t'(\mathbf{y}) \mid \exists \mathbf{y}\ \varphi(\mathbf{y})\} \stackrel{\text{def}}{=} f(\mathbf{t}) \in \{x \mid \exists \mathbf{y}\ (\varphi(\mathbf{y}) \wedge t'(\mathbf{y}) = x)\},$$
$$f(\mathbf{t}) \in \{t'_1, \ldots, t'_n\} \stackrel{\text{def}}{=} f(\mathbf{t}) \in \{x \mid t'_1 = x \vee \cdots \vee t'_n = x\}.$$

It must be noticed that variable x in (13) is not free but implicitly quantified and local to this expression. Note that $\varphi(x)$ may contain other quantified and/or free variables. For instance, observe the difference between

$$Person(y) \rightarrow a(y) \in \{x \mid Parent(x, y)\}, \tag{16}$$
$$Person(y) \rightarrow a(y) \in \{x \mid \exists y\ Parent(x, y)\}. \tag{17}$$

In (16), we assign, per each person y, one of her parents to $a(y)$, whereas in (17), we assign *any* parent, as, in fact, we could change the set condition to $\exists z\ Parent(x, z)$.

At a first sight, it could seem that the formula $\exists x(\varphi(x) \wedge f(\mathbf{t}) = x)$ could capture the expected meaning of $f(\mathbf{t}) \in \{x \mid \varphi(x)\}$ in a more direct way. Unfortunately, such a formula would not 'pick' a value x among those that satisfy $\varphi(x)$. For instance, if we translate $a \in \{x \mid \exists y\ Parent(x, y)\}$ as $\exists x(\exists y\ Parent(x, y) \wedge a = x)$ would allow the free addition of facts for $Parent(x, y)$. Notice also that a formula like $a \in \{t\}$ is stronger than an assignment $a := t$ since when t is undefined, the former is always false, regardless of the value of a (it would informally correspond to an expression like $a \in \emptyset$).

5 Logic programs with evaluable functions

In this section, we consider a subset of $\mathrm{QEL}^{=}_{\mathscr{F}}$ which corresponds to a certain kind of logic programs that allow evaluable functions but not constructors other than a finite set of Herbrand constants $\mathscr{C} = \mathscr{C}_0$. The interest of this syntactic class is that it can be translated into ground normal logic programs, and so equilibrium models can be computed by any of the currently available answer set provers. From now

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

on, we assume that any function f/n with arity $n > 0$ is evaluable, $f/n \in \mathscr{F}$, and any constant c is a constructor, $c \in \mathscr{C}$, unless we include a declaration $c/0 \in \mathscr{F}$. As usual in logic programming notation, we use in this section capital letters to represent variables.

In what follows, we will use the tag 'FLP' to refer to functional logic programming definitions and 'LP' to talk about the more restrictive syntax of normal logic programs (without functions). An FLP-*atom* has the form $p(\mathbf{t})$, $t_1 = t_2$ or $t_1 \# t_2$, where p is a predicate name, $\mathbf{t}$ is a tuple of terms and t_1, t_2 are a pair of terms. An FLP-*literal* is an FLP-atom A (called *positive* literal) or its default negation $\neg A$ (called *negative* literal). We call LP-*terms* (resp. LP-*atoms*, resp. LP-*literals*) to those not containing evaluable function symbols.

Definition 5 (*FLP -rule*)
An FLP-*rule* is an implication α :- β, where β (called *body*) is a conjunction of literals and α (called *head*) has the form of one the following expressions:

(i) an FLP-atom $p(\mathbf{t})$,
(ii) the truth constant $\bot$,
(iii) an *assignment* $f(\mathbf{t}) := t'$ with $f \in \mathscr{F}$ or
(iv) a *choice* like $f(\mathbf{t}) \in \{x \mid \varphi(x)\}$ with $f \in \mathscr{F}$ and $\varphi(x)$ a conjunction of literals; we call x the *choice variable* and $\varphi(x)$ the choice condition. □

Function f in (iii) and (iv) is called the *head function*. A *choice rule* is a rule with a *choice* head. A *functional logic program* is a set of FLP-rules. The following is an example of a program in FLP syntax:

Example 2 (*Hamiltonian cycles*)
A *Hamiltonian cycle* is a cyclic path in a graph that visits all its nodes exactly once. We encode this problem by using a function $next(X)$ that specifies which is the next node in the path for node X and $visited(X)$ that keeps track of visited nodes. The program Π_2 consists of the following rules:

$$next(X) \in \{Z \mid arc(X,Z)\} \text{ :- } node(X), \tag{18}$$

$$visited(next(0)), \tag{19}$$

$$visited(next(X)) \text{ :- } visited(X), \tag{20}$$

$$\bot \text{ :- } node(X) \wedge \neg visited(X), \tag{21}$$

where we assume that we always have some node 0, called the 'initial' one. □

An LP-*rule* is such that its body exclusively contains LP-literals and its head is either $\bot$ or an LP-atom $p(\mathbf{t})$. An LP-*program* is a set of LP-rules. As LP-rules do not contain evaluable functions, any LP-rule α :- β is simply equivalent to $\beta \rightarrow \alpha$. Thus, an LP-program has the form of a (standard) normal logic program with constraints and without evaluable functions. The absence of evaluable functions guarantees that $\mathrm{QEL}^{=}_{\mathscr{F}}$ and QEL coincide for this kind of program:

Proposition 2
$\mathrm{QEL}^{=}_{\mathscr{F}}$ equilibrium models of an LP-program Π correspond to QEL equilibrium models of Π. □

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

5.1 Translation to programs without functions

The translation of an FLP-program Π will be done in two steps. In the first step, we will define a QEL theory $\Gamma(\Pi)$ for a different signature and prove that it is $\mathrm{SQHT}^{=}_{\mathscr{F}}$ equivalent modulo the original signature. This theory $\Gamma(\Pi)$ is not an LP-program yet, as it will allow existential quantifiers and double negations in the rule bodies. However, these features can be removed in the second step by introducing auxiliary predicates so that the resulting LP-program preserves strong equivalence with respect to equilibrium models (modulo the original signature). We will focus here on the first translation step – for a description on the transformations removing existential quantifiers and double negations and its complete proof of correctness; see Cabalar (2009).

The main idea of the translation is that for each evaluable function $f/n \in \mathscr{F}$ occurring in Π, we will introduce a predicate like $holds_f(X_1, \ldots, X_n, V)$ in Π^* or $holds_f(\mathbf{X}, V)$ for short. The technique of converting a function into a predicate and shifting the function value as an extra argument is well known in FLP and has received the name of *flattening* (Naish 1991; Rouveirol 1994). Flattening in ASP was also applied for translating the languages in Cabalar and Lorenzo (2004) and Cabalar (2005) into (function-free) logic programs and in Lin and Wang (2008) to show that total functions can be removed in favour of predicates. Obviously, once we deal with a predicate, we will need that no two different values are assigned to the same function. This can be simply captured by

$$\bot \leftarrow holds_f(\mathbf{X}, V) \wedge holds_f(\mathbf{X}, W) \wedge \neg(V = W), \tag{22}$$

with variables V, W not included in $\mathbf{X}$.

Given the original signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ for program Π, the theory $\Gamma(\Pi)$ will deal with a new signature $\Sigma^* = \langle \mathscr{C}, \emptyset, \mathscr{P}^* \rangle$, where $\mathscr{P}^*$ consists of $\mathscr{P}$ plus a new predicate $holds_f/(n+1)$ per each evaluable function $f/n \in \mathscr{F}$.

Definition 6 (*Correspondence of interpretations*)
Given an HT interpretation $I = \langle S^h, S^t \rangle$ for signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ we define a *corresponding interpretation* $I^* = \langle (\sigma^h, J^h), (\sigma^t, J^t) \rangle$ for signature $\Sigma^* = \langle \mathscr{C}, \emptyset, \mathscr{P}^* \rangle$ so that for any $f/n \in \mathscr{F}$, any tuple $\mathbf{c}$ of n elements from $\mathscr{C}$, any predicate $p/n \in \mathscr{P}$ and any $w \in \{h, t\}$:
(1) $holds_f(\mathbf{c}, d) \in J^w$ iff $\sigma^w(\mathbf{c}) = d$ with $d \in \mathscr{C}$,
(2) $p(\mathbf{c}) \in J^w$ iff $p(\mathbf{c}) \in I^w$. □

Once (22) is fixed, the correspondence between I and I^* is bidirectional:

Proposition 3
Given signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ and an interpretation J for Σ^* satisfying (22), then there exists an interpretation I for Σ such that $I^* = J$.

Definition 7 (*Translation of terms*)
We define the translation of a term t as the triple $\langle t^*, \Phi(t) \rangle$, where t^* is an LP-term and $\Phi(t)$ is a formula such that

(1) for an LP-term t, $t^* \stackrel{\text{def}}{=} t$ and $\Phi(t) \stackrel{\text{def}}{=} \top$,

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

(2) when t is like $f(\mathbf{t})$ with f an evaluable function, then $t^* \stackrel{\text{def}}{=} X$ and $\Phi(t) \stackrel{\text{def}}{=} \Phi(\mathbf{t}) \wedge holds_f(\mathbf{t}^*, X)$, where X is a new fresh variable and $\Phi(\mathbf{t})$ stands for the conjunction of all $\Phi(t_i)$ for all terms t_i in the tuple $\mathbf{t}$. □

For 0-ary evaluable functions, we would have that $\mathbf{t}$ is empty – in this case, we just assume that $\Phi(\mathbf{t}) = \top$. We introduce now some additional notation. Given a term t, $subterms(t)$ denotes all its subterms, including t itself. Given a set of terms T, by T^* we mean $\{t^* \mid t \in T\}$. If ρ is a replacement of variables by Herbrand constants $[\mathbf{X}/\mathbf{c}]$, we write $I, w, \rho \models \varphi$ to stand for $I, w \models \varphi[\mathbf{X}/\mathbf{c}]$. Given a conjunction of literals $B = L_1 \wedge \cdots \wedge L_n$, we denote $B^* \stackrel{\text{def}}{=} L_1^* \wedge \cdots \wedge L_n^*$.

Definition 8 (*Translation of literals*)
The *translation of an atom* (or positive literal) A is a formula A^* defined as follows:

(1) If A has the form $p(\mathbf{t})$, then $A^* \stackrel{\text{def}}{=} \exists \mathbf{X}(p(\mathbf{t}^*) \wedge \Phi(\mathbf{t}))$, where $\mathbf{X}$ is the set of new fresh variables in $subterms(\mathbf{t})^*$ (those not occurring in the original literal).
(2) If A is like $(t_1 = t_2)$, then $A^* \stackrel{\text{def}}{=} \exists \mathbf{X}(t_1^* = t_2^* \wedge \Phi(t_1) \wedge \Phi(t_2))$, where $\mathbf{X}$ is the set of new fresh variables in $subterms(t_1)^* \cup subterms(t_1)^*$.
(3) If A is like $(t_1 \# t_2)$, then $A^* \stackrel{\text{def}}{=} \exists \mathbf{X}(t_1^* \neq t_2^* \wedge \Phi(t_1) \wedge \Phi(t_2))$, where $\mathbf{X}$ is the set of new fresh variables in $subterms(t_1)^* \cup subterms(t_1)^*$.

The *translation of a negative literal* $L = \neg A$ is the formula $L^* \stackrel{\text{def}}{=} \neg A^*$. □

Notice the difference in the translation of the two kinds of inequalities $\neq$ and $\#$. For instance, while $f(X) = 0$ becomes the negated formula $\neg \exists X' \ (X' = 0 \wedge holds_f(X, X'))$, i.e. either f has no value or it has a non-zero value, the literal $f(X) \# 0$ becomes the formula $\exists X' \ (X' \neq 0 \wedge holds_f(X, X'))$ which can be seen as 'positive', as negation does not affect to any predicate apart from equality.

Definition 9 (*Translation of rules*)
The *translation of an (FLP) rule* r like H :- B is a conjunction of formulas $\Gamma(r)$ defined as follows:

(1) If $H = \bot$, then $\Gamma(r)$ is the formula $\bot \leftarrow B^*$.
(2) If H is like $p(\mathbf{t})$, then $\Gamma(r)$ is the formula $p(\mathbf{t}^*) \leftarrow \Phi(\mathbf{t}) \wedge B^*$.
(3) If H has the form $f(\mathbf{t}) := t'$, then $\Gamma(r)$ is the formula $holds_f(\mathbf{t}^*, t'^*) \leftarrow \Phi(\mathbf{t}) \wedge \Phi(t') \wedge B^*$.
(4) If H has the form $f(\mathbf{t}) \in \{X \mid \varphi(X)\}$, then $\Gamma(r)$ is the conjunction of

$$holds_f(\mathbf{t}^*, X) \vee \neg holds_f(\mathbf{t}^*, X) \leftarrow \Phi(\mathbf{t}) \wedge B^* \wedge \varphi(X)^*, \tag{23}$$

$$\bot \leftarrow \neg \exists X(holds_f(\mathbf{t}^*, X) \wedge \varphi(X)^*) \wedge \Phi(\mathbf{t}) \wedge B^*, \tag{24}$$

where we assume that if X happened to occur in B, we have previously replaced it in the choice by a new fresh variable symbol, say $\{Y \mid \varphi(Y)\}$.

Definition 10 (*Translation of a program* $\Gamma(\Pi)$)
The *translation of an FLP program* Π is a theory $\Gamma(\Pi)$ consisting of the union of all $\Gamma(r)$ per each rule $r \in \Pi$ plus, for each evaluable function f/n, the schemata (22). □

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Theorem 1 (*Correctness of* $\Gamma(\Pi)$)
Let Π be an FLP-program with signature $\Sigma = \langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$, I am interpretation for signature Σ and $J = I^*$ an interpretation for signature Σ^* then: $I, w \models \Pi$ iff $I^*, w \models \Gamma(\Pi)$. □

As an example, the translation of Π_2 is the theory $\Gamma(\Pi_2)$:

$$holds_next(X,Z) \vee \neg holds_next(X,Z) \leftarrow arc(X,Z) \wedge node(X), \tag{25}$$

$$\bot \leftarrow \neg \exists Z (holds_next(X,Z) \wedge arc(X,Z)) \wedge node(X), \tag{26}$$

$$visited(X) \leftarrow holds_next(0,X), \tag{27}$$

$$visited(X_2) \leftarrow holds_next(X,X_2) \wedge visited(X), \tag{28}$$

$$\bot \leftarrow node(X) \wedge \neg visited(X), \tag{29}$$

$$\bot \leftarrow holds_next(X,V) \wedge holds_next(X,W) \wedge \neg(V = W). \tag{30}$$

Of course, $\Gamma(\Pi)$ is not a normal logic program, since in the general case, it contains rule heads like $\varphi \vee \neg\varphi$ for some atom φ, as in (46), or expressions in the body like $\exists X\ \varphi(X)$, with $\varphi(X)$ being a conjunction of literals, as in (47). However, as stated earlier and is detailed in Cabalar (2009) and Lee and Palla (2009), we can always build an LP-program Π^* by removing these constructions and introducing new auxiliary predicates. For instance, a formula like $\varphi \vee \neg\varphi \leftarrow \beta$, which is strongly equivalent to a double negation in the body $\varphi \leftarrow \neg\neg\varphi \wedge \beta$, can be replaced by the pair of rules $(\varphi \leftarrow \neg aux \wedge \beta)$ and $(aux \leftarrow \neg\varphi \wedge \beta)$, where aux is a new auxiliary predicate. Similarly, we can replace a formula $\exists X\ \varphi(X)$ in a rule body by a new auxiliary predicate aux' and include a rule $(aux' \leftarrow \varphi(X))$ for its definition. Of course, the auxiliary predicates must incorporate as arguments all the free variables of the original expression they replace. In our example, the final program Π_2^* would result from replacing in $\Gamma(\Pi_2)$ the formula (46) by rules

$$holds_next(X,Z) \leftarrow \neg aux(X,Z) \wedge arc(X,Z) \wedge node(X),$$
$$aux(X,Z) \leftarrow \neg holds_next(X,Z) \wedge arc(X,Z) \wedge node(X),$$

and (47) by rules

$$aux'(X) \leftarrow holds_next(X,Z) \wedge arc(X,Z) \wedge node(X),$$
$$\bot \leftarrow \neg aux'(X) \wedge node(X).$$

5.2 *Safety*

As we will translate a set of FLP-rules into a set of LP-rules, when trying to ground the latter, we will need to guarantee their domain independence, i.e. that the set of stable models are not affected by extending the set of constants. To this aim, we introduce a notion of *safety* for FLP-rules that guarantees the safety of the resulting LP program.

Definition 11 (*Restricted variable*)
A variable X is said to be *restricted* in a conjunction of literals β by a positive literal A in β when X occurs in A and one of the following holds:

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

(1) A has the form $p(\mathbf{t})$,
(2) A contains a term $f(\mathbf{t})$ and X is one of the arguments in $\mathbf{t}$,
(3) A has the form $f(\mathbf{t}) = X$ or $X = f(\mathbf{t})$,
(4) A has the form $X = Y$ or $Y = X$ and, in its turn, Y is restricted by a different positive literal A' in β.

We just say that X is *restricted* in β if it is restricted by some A in β. □

As an example, given the conjunction of literals

$$p(X, f(Y)) \wedge f(Z) \# W \wedge \neg q(V) \wedge V = Y, \tag{31}$$

X and Y are restricted by the first literal in (31), Z is restricted by the second literal and V is restricted by $V = Y$, since Y is already restricted by the first literal. Similarly, in Example 2, observe that X is restricted in the bodies of (18), (20) and (21).

Definition 12 (*Safe rule*)
A rule r, of one of the forms in Definition 5, is said to be *safe* when each variable X occurring in r satisfies the following:

(1) If X is not a choice variable and is not restricted in the body of r, then
   - X does not occur in the scope of negation and
   - X is not t' nor one of the arguments in $\mathbf{t}$, in any of the possible forms of the head of r.
(2) If X is a choice variable, then it is restricted in the choice condition $\varphi(x)$. □

For instance, the rules $p(f(X), Y) \text{ :- } q(Y)$ and $g \in \{Y \mid p(Y)\}$ are safe, whereas the rules $f(Z) := 0$ or $g \in \{Y \mid \neg p(Y)\}$ are not safe. A safe program is a set of safe rules. It can be easily checked that the FLP-program Π_2 in Example 2 is safe.

When we restrict our definition of safety to the case of LP-programs, Definition 12 trivially amounts to the standard concept of safe rule, with a minor exception due to our slightly weakened concept of restricted variable. In particular, when we do not have functions, the standard concept of restricted variable X requires that X is the argument of some positive literal formed with a non-equality predicate (Item 1 of Definition 11). In our case, we also allow that a variable becomes restricted by an equality atom $X = Y$ (Item 2 of Definition 11) in the trivial case where Y is restricted by another positive literal, like in the example

$$p(X) \leftarrow q(Y) \wedge X = Y. \tag{32}$$

This rule, for instance, is considered unsafe by current implementations of grounders `DLV` (Leone *et al.* 2006) and `GrinGo` (Gebser *et al.* 2007), although it is strongly equivalent to $p(X) \leftarrow q(X)$ which is obviously safe. Of course, rejecting a rule like (32) is not a great restriction, since a programmer would rarely write this kind of redundant code. In our case, however, accepting (32) as safe gets a relative importance since bodies like this may easily arise from our automated translation from FLP-programs to LP-programs. Anyway, in order to make rules like (32) acceptable by current ASP grounders, we assume that our final LP-program Π^* can

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

be post-processed to remove these redundant variables by exhausting the rewriting rule:

$$\frac{\beta \wedge X = Y \rightarrow \alpha}{\beta[Y/X] \rightarrow \alpha[Y/X]}.$$

As Π^* is free of functions, its set of $\mathrm{QEL}^{=}_{\mathscr{F}}$ equilibrium models coincides with its QEL equilibrium models, and these in their turn, in the case of safe programs, coincide with the set of stable models of the grounded version of the program Π^*. So, given the correspondence between Π^* and Π established in Theorem 1, we just remain to prove the following.

Theorem 2
If Π is safe, then Π^* is safe. □

6 Lin and Wang's evaluable total functions

As commented in the Introduction, (Lin and Wang 2008) introduced a closely related approach for dealing with (evaluable) functions in ASP. We will refer to this approach as *FASP*, taking the name of its associated implementation. In what follows, we show that FASP can be embedded into a subclass of $\mathrm{QEL}^{=}_{\mathscr{F}}$ where all functions are *total*, that is they satisfy axiom (12).

FASP formalism is a many-sorted first-order language so that all constants, variables, predicate and function arguments and values belong to a predefined type or sort, containing a finite and non-empty set of elements.

Definition 13 (*FASP-signature*)
An FASP-*signature* has the form $\langle \mathscr{C}, \mathscr{F}, \mathscr{P}, \mathscr{T}, \rho \rangle$, where $\mathscr{C}$, $\mathscr{F}$ and $\mathscr{P}$ have the same meaning as before and

- $\mathscr{C} = \mathscr{C}_0$, that is all constructors are 0-ary (like in our FLP-programs),
- $\mathscr{F}_0 = \emptyset$, that is there are no 0-ary evaluable functions,
- $\mathscr{T}$ is a non-empty finite set of *type names*, at least including *bool*,
- ρ is a *rank function* that assigns a pair $(\mathbf{T}, \tau)$ to each predicate, function and variable in the language so that $\mathbf{T}$ is a (possibly empty) tuple of type names called the *domain*[10] and τ is a type name called the *range*, so that, for predicates, $\tau = bool$, and for variables, $\mathbf{T} = \epsilon$ (the empty tuple). □

We will use the following abbreviations for rank declarations:

$$p \subseteq \tau_1 \times \cdots \times \tau_n, \tag{33}$$

$$f : \tau_1 \times \cdots \times \tau_n \longrightarrow \tau_{n+1}, \tag{34}$$

$$X : \tau, \tag{35}$$

that respectively stand for $\rho(p) = ((\tau_1, \ldots, \tau_n), bool)$, $\rho(f) = ((\tau_1, \ldots, \tau_n), \tau_{n+1})$ and $\rho(X) = (\epsilon, \tau)$.

[10] Sometimes also called *arity*, but we prefer here to maintain this name for the *number* of arguments in the tuple.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

An FASP-*rule* is an expression of the form

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \neg C_1 \wedge \cdots \wedge \neg C_m,$$

where A is $\bot$ (empty) or a predicate atom and B_i, $1 \leqslant i \leqslant m$, and C_j, $1 \leqslant j \leqslant n$, are atoms. As a many-sorted formalism, all terms and atoms occurring in the rule are supposed to additionally satisfy a *type coherence* restriction: for short, all arguments of predicates and functions must be of a compatible sort with respect to their rank. In the case of equality, $t_1 = t_2$ where both t_1 and t_2 must belong to the same sort.

An FASP-*program* Π is a set of FASP-rules together with a set of *type definitions*, one for each type τ used in the rules of Π and having the form $\tau : \{c_1, \ldots, c_n\}$, where c_i is an enumeration of constant names with $n > 0$. The following are a pair of examples extracted from (Lin and Wang 2008).

Example 3 (*Graph colouring problem*)
We must assign a colour to each node of a graph so that no two adjacent nodes have the same colour. A possible formalisation in FASP uses a function $clr : node \longrightarrow colour$, a predicate $arc \subseteq node \times node$, a pair of variables $X, Y : node$ and the program Π_3 containing the single rule

$$\bot \leftarrow arc(X, Y) \wedge clr(X) = clr(Y). \tag{36}$$

□

Example 4 (*Hamiltonian cycles in FASP*)
For instance, the Hamiltonian cycles are encoded in FASP with the program Π_4 consisting of the following rules:

$$\bot \leftarrow \neg arc(X, next(X)), \tag{37}$$

$$visited(next(0)), \tag{38}$$

$$visited(next(X)) \leftarrow visited(X), \tag{39}$$

$$\bot \leftarrow \neg visited(X), \tag{40}$$

together with the following domain and range declarations

$$arc \subseteq node \times node, \qquad next : node \longrightarrow node,$$

$$visited \subseteq node, \qquad X : node.$$

□

Definition 14 (*FASP-Interpretation*)
Given a signature $\langle \mathscr{C}, \mathscr{F}, \mathscr{P}, \mathscr{T}, \rho \rangle$, an *FASP-interpretation* S is a state (σ, A) for $\langle \mathscr{C}, \mathscr{F}, \mathscr{P} \rangle$ additionally satisfying the following:

- $\sigma(f(\mathbf{c})) \neq \mathtt{u}$ (functions are total).
- For each predicate p with domain $p \subseteq \tau_1 \times \cdots \times \tau_n$ then, each atom $p(\mathbf{c}) \in A$ satisfies $\mathbf{c} \in \tau_1 \times \cdots \times \tau_n$.
- For each funcion f with domain and range $f : \tau_1 \times \cdots \times \tau_n \longrightarrow \tau_{n+1}$, $n > 0$, then for any $\mathbf{c} \in \tau_1 \times \cdots \times \tau_n$, $\sigma(f(\mathbf{c})) \in \tau_{n+1}$. □

Given an FASP program Π, its grounding contains all type definitions in Π plus the rules that are obtained by replacing all variables in the rules of Π by the elements

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

in their respective ranges in all the possible ways. Notice that the grounding of Π may introduce constant symbols that did not occur in the non-ground rules but were elements of some type $\tau_i \subseteq \mathscr{C}$ in the signature.

Definition 15 (*Reduction* Π^S)
We define the *reduction* of a (ground) FASP-program Π under an FASP-interpretation $S = (\sigma, A)$, written Π^S, as the set of rules obtained from Π by iterating the following transformations:

- replace each functional term $f(\mathbf{c})$ in a rule by $\sigma(f(\mathbf{c}))$,
- replace by $\bot$ any equality literal like $c \neq c$ or $c = d$ with constants c and d syntactically different,
- replace by $\bot$ any body literal $\neg p(\mathbf{c})$ such that $p(\mathbf{c}) \in A$,
- replace by $\top$ the rest of literals $\neg p(\mathbf{c})$ and the rest of equality literals from the bodies of the remaining rules. □

We further assume that rules containing $\bot$ in their body are removed, whereas all $\top$ constants are removed from rule bodies.

It is easy to see that the ground program Π^S does not contain negation, equality or functions, although it may contain constraints. Let Π^S_{nc} be the set of non-constraint rules in Π^S. This program has a propositional least model, a set of ground atoms we denote as $LM(\Pi^S_{nc})$.

Definition 16 (*Answer set*)
We say that an FASP-interpretation $S = (\sigma, A)$ is an *answer set* of a (ground) FASP-program Π if $A = LM(\Pi^S_{nc})$ and A satisfies all the constraints in Π^S. □

6.1 Correspondence to FLP-programs

It may be noticed that the main syntactic difference between FASP and FLP-programs relies in that the former are many sorted. To overcome this difficulty, we will introduce sorts in FLP-programs as abbreviations of additional conditions and constraints. To this aim, given an FASP-program, we define the corresponding FLP-program $\hat{\Pi}$ as follows.

For each type declaration $\tau : \{c_1, \ldots, c_n\}$ in an FASP-program Π, we include a new fresh predicate with the same name τ in the signature of $\hat{\Pi}$ plus the set of FLP-atoms $\tau(c_i)$ for $1 \leqslant i \leqslant n$. For each function rank declaration like (34), we include in $\hat{\Pi}$ the rule

$$f(X_1, \ldots, X_n) \in \{X' \mid \tau_{n+1}(X')\} \leftarrow \tau_1(X_1) \wedge \cdots \wedge \tau_n(X_n), \tag{41}$$

and for any FASP-rule $\alpha \leftarrow \beta$ containing variables $X_1, \ldots, X_n$ with their respective ranges $\tau_1, \ldots, \tau_n$, we include in $\hat{\Pi}$ the FLP-rule

$$\alpha \leftarrow \beta \wedge \tau_1(X_1) \wedge \cdots \wedge \tau_n(X_n). \tag{42}$$

For instance, program $\hat{\Pi}_3$ would correspond to

$$clr(X) \in \{Y \mid colour(Y)\} \leftarrow node(X), \tag{43}$$

$$\bot \leftarrow arc(X, Y) \wedge clr(X) = clr(Y) \wedge node(X) \wedge node(Y), \tag{44}$$

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

plus a set of facts for unary predicates *node* and *colour*. Similarly, $\hat{\Pi}_4$ would consist of

$$next(X) \in \{Y \mid node(Y)\} \leftarrow node(X), \tag{45}$$

$$\bot \leftarrow \neg arc(X, next(X)) \wedge node(X), \tag{46}$$

$$visited(next(0)), \tag{47}$$

$$visited(next(X)) \leftarrow visited(X) \wedge node(X), \tag{48}$$

$$\bot \leftarrow \neg visited(X) \wedge node(X). \tag{49}$$

It can be noticed that for translating FASP-programs, we do not actually need the operator :- because functions are total (when applied to arguments in their domain). As a second observation, it is easy to see that since all variables in any FASP program Π are sorted, the resulting program $\hat{\Pi}$ will be safe, since any rule where a variable X occurs will include in its body a predicate atom[11] $\tau(X)$. As a result, we can just focus the comparison on the ground versions of Π and $\hat{\Pi}$, denoted $grnd(\Pi)$ and $grnd(\hat{\Pi})$ respectively. Note that the grounding of a rule like (41) corresponds to its definition as derived operator in terms of (14) and (15). Furthermore, as types have a finite extension $\tau : \{c_1, \ldots, c_n\}$, a formula like $\exists X(\tau(X) \wedge \alpha(X))$ can be unfolded as a finite disjunction $(\tau(c_1) \wedge \alpha(c_1)) \vee \cdots \vee (\tau(c_n) \wedge \alpha(c_n))$.

For instance, the grounding of (43) for $node : \{1, 2\}$ and $colour : \{r, g\}$ would contain (among other with false body) the following set of rules:

$$clr(1) = r \vee clr(1) \neq r \leftarrow node(1) \wedge colour(r),$$

$$clr(1) = g \vee clr(1) \neq g \leftarrow node(1) \wedge colour(g),$$

$$\bot \leftarrow node(1) \wedge \neg(colour(g) \wedge f(1) = g \vee colour(r) \wedge f(1) = r),$$

$$clr(2) = r \vee clr(2) \neq r \leftarrow node(2) \wedge colour(r),$$

$$clr(2) = g \vee clr(2) \neq g \leftarrow node(2) \wedge colour(g),$$

$$\bot \leftarrow node(2) \wedge \neg(colour(g) \wedge f(2) = g \vee colour(r) \wedge f(2) = r),$$

which, since the extent of *node* and *colour* is fixed, can be further simplified into the equivalent program

$$clr(1) = r \vee clr(1) \neq r,$$

$$clr(1) = g \vee clr(1) \neq g,$$

$$\bot \leftarrow \neg f(1) = g \wedge \neg f(1) = r,$$

$$clr(2) = r \vee clr(2) \neq r,$$

$$clr(2) = g \vee clr(2) \neq g,$$

$$\bot \leftarrow \neg f(2) = g \wedge \neg f(2) = r.$$

Generalising this process, the following lemma is relatively simple to check.

[11] In fact, this works in the same way as directive `#domain` directive in `lparse`, Section 5.5 in Syrjänen (2007), for declaring sorted variables.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Lemma 1
The grounding in $\hat{\Pi}$ of a choice rule like (41) with respect to FASP program Π and signature $\langle \mathscr{C}, \mathscr{F}, \mathscr{P}, \mathscr{T}, \rho \rangle$ is equivalent to the set of ground formulas

$$f(\mathbf{d}) = c_i \vee f(\mathbf{d}) \neq c_i, \tag{50}$$

$$\bot \leftarrow f(\mathbf{d}) \neq c_1 \wedge \cdots \wedge f(\mathbf{d}) \neq c_n, \tag{51}$$

for any $1 \leqslant i \leqslant n$, being $\tau : \{c_1, \ldots, c_n\}$ the range of f, and for any $\mathbf{d}$ tuple of constants in $\mathscr{C}$ such that $\mathbf{d}$ belongs to the domain of f. □

After examining the satisfaction of formulas in $\mathrm{SQHT}^{=}_{\mathscr{F}}$, from this we easily conclude the next result.

Lemma 2
Any $\mathrm{SQHT}^{=}_{\mathscr{F}}$ interpretation $I = (S^h, S^t)$, with $S^h = (\sigma^h, A^h)$ and $S^t = (\sigma^t, A^t)$, is a model of (A 1) and (A 2) iff $\sigma^h(f(\mathbf{d})) = \sigma^t(f(\mathbf{d})) = c$ being $\mathbf{d}$ a tuple of constants in the domain of f and c some constant in the range of f. □

Lemma 3
Let Π be an FASP-program for signature $\langle \mathscr{C}, \mathscr{F}, \mathscr{P}, \mathscr{T}, \rho \rangle$ and I be any $\mathrm{SQHT}^{=}_{\mathscr{F}}$ interpretation (S^h, S) with $S^h = (\sigma^h, A^h)$ and $S = (\sigma, A)$. Then $I \models grnd(\hat{\Pi})$ iff $I \models grnd(\Pi)^S$. □

Theorem 3
Given a ground FASP-program Π for signature $\langle \mathscr{C}, \mathscr{F}, \mathscr{P}, \mathscr{T}, \rho \rangle$, $S = (\sigma, A)$ is an answer set for $grnd(\Pi)$ iff (S, S) is an equilibrium model for $grnd(\hat{\Pi})$.

6.2 Some remarks on expressiveness

At the sight of Lin and Wang (2008), the reader may wonder about the real need for partial functions for knowledge representation. For instance, any partial function can be easily encoded as a total one by just adding a new special value (typically called *none*) to denote undefinedness[12]. However, the real difference between FASP and $\mathrm{QEL}^{=}_{\mathscr{F}}$ is not so related to totality versus partiality but instead has more to do with a 'classical' behaviour (what we called decidable functions) versus a true non-monotonic one. To illustrate this concept, consider the following example.

Example 5 (*Empty chessboard cells*)
When describing a chess ending situation, we want to specify the content of each chessboard cell. Typically, most cells will be *empty*, and in a few cases, they will contain a chessman. To this aim, we want to use a function $board(X, Y)$ that specifies the content of a given cell position $X : \{a, \ldots, h\}$ and $Y : \{1, \ldots, 8\}$ and a set of facts to describe the occupied cells, like: $board(a, 1) = blkKing, board(b, 1) = blkPawn, board(d, 3) = whtHorse$, etc. □

[12] In fact, Lin and Wang (2008) do not specify the way in which, for instance, a division by zero should be treated.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Typically, when encoding this problem in a relational ASP setting, we would include a rule asserting that all cells are empty *by default*. In a functional setting, this means that we need to declare a *default value* for a given function, something that, as seen in Section 4, can be compactly represented with the rule

$$board(X,Y) := empty \ \ :- \ \ row(X) \wedge column(Y) \wedge \neg(board(X,Y) \# empty),$$

whose informal reading is 'assign an empty content when there is no evidence that the cell is non-empty'. An important remark is that although there may exist $\mathrm{SQHT}^{=}_{\mathscr{F}}$ models in which function *board* is partial, *this function will be total in any equilibrium model* (for any correct cell position X,Y), since the default above cannot leave $board(X,Y)$ undefined.

On the other hand, a default like this does not seem easily representable in FASP, unless we make use of additional auxiliary predicates, i.e. we end up resorting to the relational fragment of FASP. The reason for this difficulty is that functions are decidable, and so their value can be defined 'from the start'. In this way, in FASP, we would have a free choice for selecting *any* value for any function, and then only choices satisfying the rules and constraints eventually lead to an answer set. In our example, this means that if we just enumerate the occupied cells, we would have an answer set for *any* possible combination of contents of the rest of cells, but no way to assume that they are empty by default.

A similar difficulty would arise when representing inertia for functions when dealing with an actions and change scenario, something that in $\mathrm{QEL}^{=}_{\mathscr{F}}$ would have a quite natural representation. For instance, if *board* became a fluent, with a third parameter I for representing a situation number, its inertia could be written as

$$board(X,Y,I+1) := board(X,Y,I) \ \ :- \ \ \neg(board(X,Y,I+1) \# board(X,Y,I)).$$

Finally, $\mathrm{QEL}^{=}_{\mathscr{F}}$ allows a functional interpretation of predicates, as done for instance in Cabalar and Lorenzo (2004) and Cabalar (2005) so that we can define them as functions with a boolean range $\{true, false\}$. As shown in Cabalar and Lorenzo (2004), if we further assert that $false$ is a default value for all boolean functions, we obtain the same expressiveness as standard ASP. To put an example, the program

$$\begin{aligned} p &\leftarrow \neg q, \\ q &\leftarrow r \wedge \neg p, \\ r &\leftarrow \neg s, \end{aligned}$$

would be re-encoded using this technique as

$$\begin{aligned} p = true \ &\leftarrow \ q = false, \\ q = true \ &\leftarrow \ r = true \wedge q = false, \\ r = true \ &\leftarrow \ s = false, \\ A := false \ &\ :- \ \neg(A \# false), \end{aligned}$$

for A varying in p,q,r,s, so that the functional equilibrium models of this FLP-program correspond to the (standard) answer sets of the original program. In other words, we can encode full ASP by exclusively using (boolean) functions with default

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

values and without resorting to any predicate (excepting equality). In the case of FASP, the impossibility of representing defaults when only dealing with functions (that is when we suppress the use of predicates) would make this encoding to collapse into classical propositional logic.

7 Related work

The present approach has incorporated many of the ideas previously presented in Cabalar and Lorenzo (2004) and Cabalar (2005). For instance, Cabalar and Lorenzo (2004) can be seen as the fragment of our FLP-programs where we disable the use of predicates and restrict default negation exclusively for specifying default values of functions.

With respect to other logical characterisations of functional programming languages, the closest one is perhaps González-Moreno *et al.* (1999), from where we extracted the separation of constructors and evaluable functions. The main difference is that $\mathrm{QEL}^{=}_{\mathscr{F}}$ provides a completely logical description of all operators, allowing an arbitrary syntax (including rules with negation, disjunction in the head, negation and disjunction of rules, etc.). Another important difference is that $\mathrm{QEL}^{=}_{\mathscr{F}}$ is constrained to strict functions, while González-Moreno *et al.* (1999) is based on non-strict functions.

Scott's *E*-Logic is not the only choice for logical treatment of partial functions. A related approach is the so-called *logic of partial functions* (Barringer *et al.* 1984). The main difference is that the logic of partial functions is a three-valued logic – formulas containing undefined terms have a third, undefined truth value. The relation to (relational) ASP in this way is much more distant than the current approach, since stable models and their logical counterpart, equilibrium models, are two valued[13].

As for the relation to other approaches exclusively dealing with Herbrand functions (Syrjänen 2001; Bonatti 2004; Šimkus and Eiter 2007), an interesting topic for future study is analysing to which extent they could be captured by $\mathrm{QEL}^{=}$ semantics, i.e. the fragment of $\mathrm{QEL}^{=}_{\mathscr{F}}$ without evaluable functions.

8 Conclusions

We can summarise the main contributions of this paper to the introduction of a new language for dealing with functions in ASP and the discussion about several modelling issues not easily solvable within other ASP modelling paradigms. In this way, the paper has tried to clarify some relevant aspects related to the use of functions in ASP for knowledge representation. These aspects include the distinction between Herbrand and evaluable (and possibly partial) functions, the concept of definedness, the treatment of equality, the directionality in function assignments or a new non-deterministic choice operation for selecting a function value.

[13] Note that in this work, we are not considering explicit negation.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

The functional nature of some predicates is hidden in many ASP domains. When functions are represented in a relational way, we require the continuous addition of constraints for uniqueness of value and a considerable amount of extra variables to replace the ability of nesting functional terms. All this additional effort may easily become a source for programming errors.

Although, as shown, the proposed approach can be translated into relational ASP and merely considered as *syntactic sugar*, we claim that the use of functions may provide a more natural, compact and readable way of representing many scenarios. The previous experience with a very close language to that of Section 5, implemented in an online interpreter[14] and used for didactic purposes in the past, shows that the functional notation helps the student concentrate on the mathematical definition of the domain to be represented and forget some of the low-level representation tasks, as those commented above, or as the definedness conditions, that must also be considered in the relational representation, but the functional interpreter checks in an automatic way.

We hope that the current approach will help to integrate, in the future, the explicit treatment of arithmetic functions made by some ASP tools that are currently handled *outside* the formal setting. For instance, the ASP grounder `lparse`[15] syntactically accepts a program like $p(div(10, X)) \leftarrow q(X)$ but raises a 'divide by zero' run-time error if fact $q(0)$ is added to the program. On the other hand, when *div* is replaced by a non-built-in function symbol, say f, the meaning is quite different, and we get $\{p(f(10, 0)), q(0)\}$ as a stable model. In this paper, we have also identified and separated evaluable and (possibly) partial functions (like *div* above) from constructors (like f in the previous example).

We have provided a translation of our functional language into normal logic programs to show that (1) it can be implemented with current ASP solvers; but more importantly that (2) the proposed semantics is *sensible* with respect to the way in which we usually program in the existing ASP paradigm. This translation has been implemented in a tool called `lppf` (*logic programs with partial functions*)[16].

For future work, we plan to follow Lin and Wang (2008) on loop formulas for converting their programs with total functions into constraint satisfaction problems and extend their work for our functional logic programs. As in Lin and Wang (2008), we expect to obtain a reduction in the size of ground functional logic programs, with respect to the size of their relational counterparts.

A topic for future study is the implementation of a solver that directly handles the functional semantics. Other open topics are the axiomatisation of the current logical framework, the addition of a second, explicit (or strong) negation or the extension of `lppf` to combine evaluable functions with constructors of arity greater than zero, using as a back end the recently available tool `DLV-complex`[17].

[14] Available at `http://www.dc.fi.udc.es/~cabalar/fal/`
[15] Available at `http://www.tcs.hut.fi/Software/smodels/`
[16] Available at `http://www.equilibriumlogic.net/el/lppf/lppf.pl`
[17] Available at `http://www.mat.unical.it/dlv-complex`

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Acknowledgements

I am especially thankful to Joohyung Lee and Yunsong Meng for pointing out some technical errors in an early version of this work and to Francisco López Fraguas for his bibliography guidance on semantics of partial functions in the field of functional logic programming. This research was partially supported by Spanish MEC project TIN-2006-15455-C03-02 and Xunta de Galicia project INCITE08-PXIB105159PR.

Appendix. Proofs

Proof of Proposition 3
Then, it suffices with defining $I^w = \{p(\mathbf{c}) \in J^w \mid p/n \in \mathscr{P}\}$ and σ^w such that for any evaluable function f and tuple $\mathbf{c}$ in elements of $Terms(\mathscr{C})$: $\sigma^w(f(\mathbf{c})) = d$ if $holds_f(\mathbf{c}, d) \in J^w$; or $\sigma^w(f(\mathbf{c})) = \mathtt{u}$ otherwise. Note that the latter is well defined since (22) guarantees that no pair of atoms $holds_f(\mathbf{c}, d)$ and $holds_f(\mathbf{c}, e)$ with $e \neq d$ is included in any J^w. The rest of mapping σ^w is built up from its structural definition implied by Condition (ii) in Definition 1. □

Lemma 3
For any term t, interpretation I and corresponding interpretation I^*, and for any replacement ρ of variables in $subterms(t)^*$, $I^*, w, \rho \models \Phi(t)$ is equivalent to $I, w, \rho \models E\ t$ and $I, w, \rho \models (t')^* = t'$ for any $t' \in subterms(t)$.

Proof
We proceed by induction. For the base case, when t is an LP-term, $E\ t$ is valid and so equivalent to $\top = \Phi(t)$; besides, $t^* = t$ by definition and t has no subterms. Assume that it is proved for a tuple of terms $\mathbf{t}$ and consider $t = f(\mathbf{t})$. Then, note that $I^*, w, \rho \models \Phi(t)$ is equivalent to Condition (A): $I^*, w, \rho \models \Phi(\mathbf{t})$ and $I^*, w, \rho \models holds_f(\mathbf{t}^*, X_t)$. Now the first conjunct of (A) is equivalent, by induction, to $I, w, \rho \models E\ \mathbf{t}$ and $I, w \models (t')^* = t'$ for any subterm of $\mathbf{t}$, whereas the second conjunct of (A) is equivalent, by the correspondence between I and I^*, to $I, w, \rho \models f(\mathbf{t}) = X_t$, provided that we have already obtained $I, w, \rho \models \mathbf{t}^* = \mathbf{t}$. To sum up, (A) is therefore equivalent to $I, w, \rho \models E\ \mathbf{t} \wedge f(\mathbf{t}) = X_t$ and $I, w, \rho \models (t')^* = t'$ for any subterm of $\mathbf{t}$. Since $E\ \mathbf{t} \wedge f(\mathbf{t}) = X_t$ is equivalent to $E\ f(\mathbf{t}) \wedge f(\mathbf{t}) = X_t$ and this, by definition, is the same as $E\ t \wedge t = t^*$, we finally obtain $I, w, \rho \models E\ t$ and $I, w, \rho \models (t')^* = t'$ for any subterm of t. □

Lemma 5
For any body literal L: $I^*, w \models L^*$ iff $I, w \models L$.

Proof
Depending on the form of L, we have the following:

(1) If L is some atom $p(\mathbf{t})$, then $I^*, w \models L^*$ means that for some substitution ρ of variables in $subterms(\mathbf{t})^*$: $I^*, w, \rho \models p(\mathbf{t}^*)$ and $I^*, w, \rho \models \Phi(\mathbf{t})$. By Lemma 3, the second conjunct is equivalent to $I, w, \rho \models E\ \mathbf{t}$ and $I, w, \rho \models t^* = t$ for any subterm t of $\mathbf{t}$ (and so of L) and in particular $I, w, \rho \models \mathbf{t}^* = \mathbf{t}$. But this means that $I^*, w, \rho \models p(\mathbf{t}^*)$ is equivalent to $I, w, \rho \models p(\mathbf{t})$ by the correspondence of I and

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

I^*. Since $p(\mathbf{t})$ implies $E\ \mathbf{t}$, we can remove the latter, and, as a result, the original condition $I^*, w, \rho \models L^*$ is equivalent to $I, w, \rho \models p(\mathbf{t})$ and $I, w, \rho \models t^* = t$ for any subterm t of L. As $p(\mathbf{t})$ does not contain variables in $subterms(\mathbf{t})^*$, the previous conditions are equivalent to $I, w \models p(\mathbf{t})$, and there exists some ρ for which $I, w, \rho \models t^* = t$. But as $I, w \models p(\mathbf{t})$ means that $p(\mathbf{t})$ is defined in I, w, the existence of a substitution ρ for variables in $subterms(\mathbf{t})^*$ that satisfies $I, w, \rho \models t^* = t$ for any subterm t of L is guaranteed and so is a redundant condition that can be removed.

(2) If L has the form $t_1 = t_2$, then the proof follows steps similar to case 1.

(3) If L has the form $\neg A$, then $I^*, w \models \neg A^*$ is equivalent to $I^*, t \not\models A^*$. Applying the proof for cases 1 and 2 to atom A, this is equivalent to $I, t \not\models A$ that is further equivalent to $I, w \models \neg A$. □

Obviously, Lemma 5 directly implies that $I, w \models B$ is equivalent to $I^*, w \models B^*$.

Lemma 6
$I^*, w \models \Gamma(r)$ iff $I, w \models r$.

Proof
If $r = (H\ \text{:-}\ B)$, depending on the form of H, we have the following:

(1) If $H = \bot$ is easy to see that $(\bot\ \text{:-}\ B)$ is equivalent to $(\bot \leftarrow B)$, then $I^*, w \models \bot \leftarrow B^* \Leftrightarrow I^*, t \not\models B^* \Leftrightarrow$ (by Lemma 5) $I, t \not\models B \Leftrightarrow I, w \models \bot \leftarrow B$.

(2) If H is like $p(\mathbf{t})$, then $p(\mathbf{t})\ \text{:-}\ B$ is equivalent to $p(\mathbf{t}) \leftarrow B \wedge E\ \mathbf{t}$. Then, $I^*, w \models p(\mathbf{t}^*) \leftarrow \Phi(\mathbf{t}) \wedge B^* \Leftrightarrow$ for all $w' \geqslant w$: if $I^*, w' \models \Phi(\mathbf{t}) \wedge B^*$, then $I^*, w' \models p(\mathbf{t}^*)$. Let us call (A) to this condition. By Lemma 5, $I^*, w' \models B^*$ is equivalent to $I, w' \models B$. Now note that rules are universally quantified. Take any replacement ρ of variables in $subterms(\mathbf{t})^*$. By Lemma 3, $I^*, w', \rho \models \Phi(\mathbf{t})$ is equivalent to $I, w', \rho \models E\ \mathbf{t}$ and $I, w', \rho \models t'^* = t'$ for any $t' \in subterms(\mathbf{t})$. If this holds, $I^*, w', \rho \models p(\mathbf{t}^*)$, which coincides with $I, w', \rho \models p(\mathbf{t}^*)$, is equivalent to $I, w', \rho \models p(\mathbf{t})$. To sum up, (A) is equivalent to for all $w' \geqslant w$, if $I, w', \rho \models B \wedge E\ \mathbf{t}$, then $I, w', \rho \models p(\mathbf{t})$ for any replacement ρ. But this is the same as $I, w \models p(\mathbf{t}) \leftarrow B \wedge E\ \mathbf{t}$.

(3) If H has the form $f(\mathbf{t}) := t'$, we may first observe that $(H\ \text{:-}\ B)$ is equivalent to $f(\mathbf{t}) = t' \leftarrow E\ \mathbf{t} \wedge E\ t' \wedge B$. Then, $I^*, w \models holds_f(\mathbf{t}^*, t'^*) \leftarrow \Phi(\mathbf{t}) \wedge \Phi(t') \wedge B^*$ is equivalent to, for any world, $w' \geqslant w$ and any replacement of variables ρ: if $I^*, w', \rho \models \Phi(\mathbf{t}) \wedge \Phi(t') \wedge B^*$, then $I^*, w', \rho \models holds_f(\mathbf{t}^*, t'^*)$. By Lemmas 3 and 5, the antecedent is equivalent to $I, w', \rho \models E\ \mathbf{t} \wedge E\ t' \wedge B$ plus $I, w', \rho \models k^* = k$ for each $k \in subterms(\mathbf{t} \cdot t')$. On the other hand, $I^*, w', \rho \models holds_f(\mathbf{t}^*, t'^*)$ is equivalent, by correspondence of I and I^*, to $I, w', \rho \models f(\mathbf{t}^*) = t'^*$ and this, in the presence of the equivalent condition for the antecedent obtained before, is equivalent to $I, w', \rho \models f(\mathbf{t}) = t'$. The rest of the proof follows as in the previous case.

(4) If H has the form $f(\mathbf{t}) \in \{X \mid \varphi(X)\}$, then, after some simple transformations, it can be checked that $(H\ \text{:-}\ B)$ is equivalent to the conjunction of the formulas

$$f(\mathbf{t}) = X \vee \neg f(\mathbf{t}) = X \leftarrow \varphi(X) \wedge E\ \mathbf{t} \wedge B, \tag{A 1}$$

$$\bot \leftarrow \neg \exists X(\varphi(X) \wedge f(\mathbf{t}) = X) \wedge E\ \mathbf{t} \wedge B. \tag{A 2}$$

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

The proof for this case is tedious but follows steps similar to the previous two cases. By analogy, it is not difficult to see that $I, w \models (A\,1)$ iff $I^*, w \models (23)$ and that $I, w \models (A\,2)$ iff $I^*, w \models (24)$. □

Proof of Theorem 1

The proof directly follows from Lemma 6. □

For the proof of Theorem 2, we will show that safety is preserved for the first step of the translation, that is when the resulting program contains double negation and existential quantifiers in the rule bodies. To this aim, we recall below the definition of safety for rules of this form extracted from (Cabalar 2009).

Definition 17 (*Safe rule*)

A rule $r : H \leftarrow B$ is said to be safe when both

(a) any free variable occurring in r also occurs free and restricted in β,
(b) for any condition $\exists x\ \varphi$ in B, x occurs free and restricted in φ, □

where the definition of restricted variable in a conjunction of literals is Definition 11, but only the applicable Items 1 and 2, that does not deal with functions. Note that for free variables, the above condition means that unrestricted variables cannot occur in the head or negated in the body.

Lemma 7

If X is restricted in an FLP-rule conjunction of literals B, then X is restricted B^*.

Proof

Following Definition 11, we have four cases:

(1) If X was restricted by some $p(\mathbf{t})$, then B^* will contain a corresponding positive atom $p(\mathbf{t}^*)$ where functional terms have been replaced by auxiliary variables but X still belongs to the tuple $\mathbf{t}^*$.
(2) If X was in a term $f(\mathbf{t})$ inside a positive atom in B, then X will be included in the corresponding atom $holds_f(\mathbf{t}^*, Y)$ that will also be positive in B^*.
(3) It X was in a positive atom $f(\mathbf{t}) = X$ (analogously for $X = f(\mathbf{t})$), then the translation will contain an auxiliary variable Y and the pair of positive atoms $Y = X$ and $holds_f(\mathbf{t}^*, Y)$.
(4) If X was in a positive atom $X = Y$ (resp. $Y = X$) and Y was restricted by another different atom, note that $X = Y$ will be preserved in B^* and that we can apply the previous items for concluding that Y is restricted in B^*. □

Lemma 8

If Π is safe, then $\Gamma(\Pi)$ is safe. □

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Proof

We will have two types of variables in $\Gamma(\Pi)$: the original ones in Π plus the auxiliary ones introduced in the translation of functional terms. We will show their safety in $\Gamma(\Pi)$ for each case, further distinguishing between choice and non-choice variables, when they belonged to Π.

- If X is a variable in some rule $r : H \ :\!-\ B$ in Π and is not a choice variable, we may have that it was restricted in B or not. If it was restricted in B, from Lemma 7 and the fact that B^* belongs to the bodies of all rules in $\Gamma(\Pi)$, we conclude that X is also restricted in those rule bodies, and so X is safe in $\Gamma(r)$. If X was not restricted in the body, as it was safe, it was not in the scope of negation in Π and was not t' or one of $\mathbf{t}$ in any of the possible heads in Definition 5. Following the translation, it is easy to see that a variable can only end being in the scope of negation if it already occurred in a negative literal in the body of r or it was one of the arguments in $\mathbf{t}$ in a head of the form $f(\mathbf{t}) \in \{Y \mid \varphi(Y)\}$, but none of these cases hold. On the other hand, it can also be checked that a variable can end in a head of $\Gamma(r)$ only when it was an element in $\mathbf{t}$ in a head like $p(\mathbf{t})$, a head like $f(\mathbf{t}) \in \{Y \mid \varphi(Y)\}$ or a head like $f(\mathbf{t}) = t'$, or X was t' in the last case. But again, none of these cases hold. As a result, X does not occur (free) in the heads of rules in $\Gamma(\Pi)$ nor negated in their bodies.
- If X is a choice variable in Π for some rule with head $f(\mathbf{t}) \in \{X \mid \varphi(X)\}$, since it was safe, we know that it is restricted in $\varphi(X)$. From Lemma 7, we conclude that X is restricted in $\varphi(X)^*$. Now, $\Gamma(\Pi)$ contains the rules (23) and (24). In the case of (23), as $\varphi(X)^*$ belongs to the body without being inside an existential quantifier, we immediately conclude that X is restricted in the body and so is safe in that rule. For (24), we have that X ends being existentially quantified, inside a formula $\exists X(holds_f(\mathbf{t}^*, X) \wedge \varphi(X)^*)$, but as X is free and restricted inside the quantified formula, we conclude again that it is safe in the rule.
- If X is an auxiliary variable, it can only be one of the auxiliary variables $\mathbf{X}$ in Definition 8 for translation of literals. Note that when we translate a positive body literal A into A^*, the latter will be included in the final rule bodies, whereas it has the form of $\exists\mathbf{X}(\alpha(\mathbf{X}))$ and, this is crucial, that $\alpha(\mathbf{X})$ results from translating terms in A and is always a conjunction of positive literals. Thus, $\mathbf{X}$ will be restricted in $\alpha(\mathbf{X})$, and thus, these variables will be safe in the result. The same happens for negative literals $\neg A$ and their translation $\neg\exists\mathbf{X}(\alpha(\mathbf{X}))$, since safety for existentially quantified variables only requires that they are restricted inside the quantified formula. □

Proof of Theorem 2

It follows from Lemma 8 for $\Gamma(\Pi)$, resulting from the first step of the translation, and from Theorem 7 in Cabalar (2009) for the second step that eventually yields Π^*. □

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Proof of Lemma 3

First, we observe that the grounding of FASP-rules yields the same result in $grnd(\Pi)$ and $grnd(\hat{\Pi})$. This is because for any rule $\alpha \leftarrow \beta$ in Π, we get a rule (42) in $\hat{\Pi}$. But then, after grounding, we can remove those rules in $\hat{\Pi}$ for which X has been replaced by some c not in the range of X, since in those cases, there is no head $\tau(c)$ in $grnd(\hat{\Pi})$. Similarly, when c belongs to the range of X, $\tau(c)$ will be a fact in $\hat{\Pi}$, and so it can be removed from the rule body so that we obtain the same result as directly grounding $\alpha \leftarrow \beta$ in Π.

Now, from Lemma 2, we get that σ^h and σ coincide for the evaluation of functions. Thus, we can replace any functional term $f(\mathbf{c})$ in $grnd(\hat{\Pi})$ by its value $\sigma(f(\mathbf{c}))$. On the other hand, from Proposition 1, we can replace any $\neg\varphi$ such that $I, t \not\models \varphi$ by $\bot$ and any one such that $I, t \not\models \varphi$ by $\top$. Finally, as all function terms in $grnd(\hat{\Pi})$ refer to arguments in the corresponding function domain, equality is always applied to defined terms, and so it has the same interpretation in S^h and S. As a result, $I \models t_1 = t_2$ iff $I \models \neg\neg(t_1 = t_2)$, and we can replace equality by the corresponding truth constant, as did for negative literals. □

Proof of Theorem 3

For the left to right direction, assume that S is an answer set for $grnd(\Pi)$ but (S, S) is not an equilibrium model of $grnd(\hat{\Pi})$. This means that there exists some smaller model $I = (S^h, S)$ of $grnd(\hat{\Pi})$, $S^h = (\sigma^h, A^h)$ that, from Lemma 2, satisfies $\sigma^h = \sigma$ and for which $A^h \subset A$. From Lemma 3, $I \models grnd(\hat{\Pi})$ is equivalent to $I \models grnd(\Pi)^S$. Now, as $grnd(\Pi)^S$ does not contain function symbols or negation, it is easy to see that the latter is equivalent to $A^h \models grnd(\Pi)^S$ in propositional logic. But the latter contradicts the fact that $S = (\sigma, A)$ is an answer set of $grnd(\Pi)$.

For the right to left direction, assume that (S, S) is an equilibrium model of $grnd(\hat{\Pi})$ but not an answer set of $grnd(\Pi)$. The latter means that there exists some $A' \subset A$ for which $A' \models grnd(\Pi)^S$. But then, we can build the SQHT$^=_{\mathcal{F}}$ interpretation $I = (S^h, S)$ with $S^h = (\sigma, A')$. As $grnd(\Pi)^S$ does not contain negation or function symbols, $A' \models grnd(\Pi)^S$ implies $I \models grnd(\Pi)^S$, and, in its turn, by Lemma 3, this is equivalent to $I \models grnd(\hat{\Pi})$. But since I is strictly smaller than (S, S), we get a contradiction with the equilibrium condition for the latter. □

References

Barringer, H., Cheng, H. and Jones, C. B. 1984. A logic covering undefinedness in program proofs. *Acta Informatica 21*, 251–269.

Bonatti, P. A. 2004. Reasoning with infinite stable models. *Artificial Intelligence 156*, 75–111.

Cabalar, P. 2005. A functional action language front-end [online]. In *Presentation at the 3rd Workshop on Answer Set Programming (ASP'05)*. Accessed January 2011. URL: `http://www.dc.fi.udc.es/ai/~cabalar/asp05_C.pdf`

Cabalar, P. 2008. Partial functions and equality in answer set programming. In *Proc. of the 24th International Conference on Logic Programming, ICLP 2008 (Udine, Italy, December 9–13 2008)* M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 392–456.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Cabalar, P. 2009. Existential quantifiers in the rule body. In *Proc. of the 23rd Workshop on (Constraint) Logic Programming (WLP'09)*. U. Geske and A. Wolf, Eds. Universitätsverlag Potsdam, 59–74

Cabalar, P. and Lorenzo, D. 2004. Logic programs with functions and default values. In *Proc. of the 9th European Conference on Logics in AI (JELIA'04)*. J. J. Alferes and J. Leite, Eds. Lecture Notes in Computer Science, vol. 3229. Springer-Verlag, 294–306.

Ferraris, P., Lee, J. and Lifschitz, V. 2007. A new perspective on stable models. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI'07)*. M. M. Veloso, Ed. Morgan Kunfmann, 372–379.

Gebser, M., Schaub, T. and Thiele, S. 2007. GrinGo : A new grounder for answer set programming. In *Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. C. Baral, G. Brewka and J. Schlipf, Eds. Lecture Notes in Computer Science, vol. 4483. Springer, 266–271.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming*. R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

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

González-Moreno, J. C., Hortalá-González, T., López-Fraguas, F. and Rodríguez-Artalejo, M. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming 40*, 1, 47–87.

Hanus, M. 1994. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming 19*, 20, 583–628.

Hanus, M. 2007. Multi-paradigm declarative languages. In *Proc. of the International Conference on Logic Programming (ICLP 2007)*. V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer, 45–75.

Heyting, A. 1930. Die formalen Regeln der intuitionistischen Logik. *Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-mathematische Klasse*, 42–56.

Heyting, A. 1956. *Intuitionism. An Introduction*. North-Holland.

Lee, J. and Palla, R. 2009. System F2LP – Computing answer sets of first-order formulas. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*. E. Erdem, F. Lin and T. Schaub, Eds. Lecture Notes in Artificial Intelligence 5753. Springer-Verlag, 515–521.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7*, 3, 499–562.

Lifschitz, V., Pearce, D. and Valverde, A. 2007. A characterization of strong equivalence for logic programs with variables. In *Proc. of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*. C. Baral, G. Brewka and J. S. Schlipf, Eds. Springer-Verlag, 188–200.

Lin, F. and Wang, Y. 2008. Answer set programming with functions. In *Proc. of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*. G. Brewka and J. Lang, Eds. AAAI Press, 454–465

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. K. R. Apt, V. W. Marek, M. Truszczynski and D. S. Warren, Eds. Springer-Verlag, 169–181.

McCarthy, J. 1980. Circumscription: A form of non-monotonic reasoning. *Artificial Intelligence 13*, 27–39.

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press

Naish, L. 1991. Adding equations to NU-Prolog. In *Proc. of the 3rd International Symposium on Programming Language Implementation and Logic Programming*. J. Maluszynski and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, 15–26.

Pearce, D. 1996. A new logical characterisation of stable models and answer sets. In *Non Monotonic Extensions of Logic Programming. Proc. NMELP'96 (LNAI 1216)*. J. Dix, L. M. Pereira and T. C. Przymusinski, Eds. Springer-Verlag.

Pearce, D. and Valverde, A. 2004. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proc. of the 9th European Conference on Logics in AI (JELIA'04)*. J. J. Alferes and J. Leite, Eds. Springer-Verlag, 147–160.

Pearce, D. and Valverde, A. 2008. Quantified equilibrium logic and foundations for answer set programs. In *Proc. of the 24th International Conference on Logic Programming, ICLP 2008, (Udine, Italy, December 9–13 2008)*. M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 546–560.

Rodríguez-Artalejo, M. 2001. Functional and constraint logic programming. In *Revised Lectures of the International Summer School CCL'99*. C. Hubert, M. Claude and T. Ralf, Eds. Lecture Notes in Computer Science, vol. 2002. Springer, 202–270.

Rouveirol, C. 1994. Flattening and saturation: Two representation changes for generalization. *Machine Learning 14*, 1, 219–232.

Scott, D. 1979. Identity and existence in intuitionistic logic. *Lecture Notes in Mathematics 753*, 660–696.

Šimkus, M. and Eiter, T. 2007. Decidable non-monotonic disjunctive logic programs with function symbols. In *Proc. of the 14th International Conference on Logic for Programming, Artificial Intelligence*. N. Dershowitz and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 4790. Springer-Verlag, 514–530.

Syrjänen, T. 2001. Omega-restricted logic programs. In *Proc. of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*. T. Eiter, W. Faber and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 2173. Springer-Verlag, 267–279.

Syrjänen, T. 2007. Lparse 1.0 user's manual [online]. Acessed January 2011. URL: `http://www.tcs.hut.fi/Software/smodels/lparse.ps`

https://doi.org/10.1017/S1471068410000517 Published online by Cambridge University Press