Answer Set Programming with Functions

Fangzhen Lin
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Yisong Wang[1,2]
[1]Guizhou University, Guiyang, P.R. China
[2]Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, Hong Kong

Abstract

To compute a function such as a mapping from vertices to colors in the graph coloring problem, current practice in Answer Set Programming is to represent the function as a relation. Among other things, this often makes the resulting program unnecessarily large when instantiated on a large domain. The extra constraints needed to enforce the relation as a function also make the logic program less transparent. In this paper, we consider adding functions directly to normal logic programs. We show that the answer set semantics can be generalized to these programs straightforwardly. We also show that the notions of loops and loop formulas can be extended, and that through program completion and loop formulas, a normal logic program with functions can be transformed to a Constraint Satisfaction problem.

Introduction

Currently in Answer Set Programming (ASP), functions are represented as special relations. For instance, to encode the graph coloring problem, instead of a unary function, say $color(x)$ that maps vertices to colors, one uses a binary relation, say $color(x, c)$ to mean that the vertex x is assigned the color c. For this to work, one needs to add some axioms saying that the predicate $color(x, c)$ is in fact functional. More importantly, it increases the size of the final instantiated program both in terms of the number of atoms and the number of rules. For instance, with $color(x, c)$ we get $M \times N$ atoms, where M is the number of vertices and N colors. For the N-queen's problem, if we use a predicate $q(x, y)$ to say that the queen at row x is placed in column y, this will generate N^2 atoms.

In this paper, we consider adding functions to logic programs. We shall argue that this allows for more direct and compact representation of problems like the graph coloring, the queen's problem, and the Hamiltonian circuit problem. We shall extend the answer set semantics to programs with functions and relate it to Constraint Satisfaction Problem (CSP) through program completion and loop formulas. Our preliminary experimental results indicate that the reduction in size as a result of using functions can pay off when the problem become large.

Syntactically, functions are allowed in logic programming from the very beginning (c.f. (Lloyd 1987)). However, they are normally interpreted under Herbrand universe. For instance, in Prolog, one cannot declare a fact like "$f(a) = b$". In fact, the query "$f(a) = b$" will always receive a "no" answer. In other words, functions are pre-defined and their values cannot be changed by the user. The same holds for most of the work in ASP that allows function symbols (e.g. (Bonatti 2004; Baselice, Bonatti, & Criscuolo 2007; Syrjänen 2001; Simkus & Eiter 2007; Calimeri, Cozza, & Ianni 2007)). While lparse (Syrjänen 1998) allows function symbols, terms constructed of these functions are just names standing for constants.

One noticeable exception is the work of Cabalar and Lorenzo (2004) and Cabalar (2005) where they proposed a logic programming language with functions only. The main differences between their language and ours are that we extend normal logic programs with functions rather than using a pure functional language, and that functions must be total in our language but can be partial in theirs. A more detailed comparison will be given later in the paper.

In the next section, we introduce our language of normal logic programs with functions. We then extend the answer set semantics to this language and show that under this semantics, functions can indeed be replaced by relations in a systematic way. We then extend the notions of loops and loop formulas to normal logic programs with functions and show how they can be used to translate logic programs with functions to constraint satisfaction problems (CSPs). We describe an implementation of logic programs with functions using CSP solvers based on this result, and report some preliminary experimental results.

Normal Logic Programs with Functions

In the following, let $\mathcal{L}$ be a many-sorted first-order language. Recall that in such a language, every predicate has an arity that specifies the number of arguments the predicate has and the type (sort) of each argument, and similarly for constants and functions. Variables also have types associated with them, and when they are used in a formula, their types are normally clear from the context.

The language $\mathcal{L}$ may have pre-interpreted symbols like the standard arithmetic functions such as "$+$", "$-$", and the absolute function "$|.|$".

In this paper, by an atom we mean an atomic formula that does not mention equality, and by an equality atom we mean a formula of the form $t = t'$. Unless stated otherwise, in this paper, by functions we mean proper functions, not constants.

A normal rule with functions, or simply a rule, is an expression of the form:

$$A \leftarrow B_1, \ldots, B_m, not\, C_1, \ldots, not\, C_n \quad (1)$$

where A is empty or an atom, B_i, $1 \leq i \leq m$, and C_j, $1 \leq j \leq n$) are atoms or equality atoms. If A is an atom, then the rule is a proper rule; if A is empty, we also call the rule a constraint.

A normal logic program P is a set of rules together with a set of *type definitions*, one for each type τ used in the rules of P, of the form:

$$\tau : D \quad (2)$$

where D is a finite and nonempty set of elements. Unless stated otherwise, all logic programs in this paper are assumed to be normal.

Informally, a type definition defines a domain for a type (sort). This is like in a first-order structure for a many-sorted language, there is a domain for each type. Here we require that if a constant c of type τ occurs in the rules of P, then the domain D of τ as specified in the type definitions of P must contain c.

Example 1 The graph coloring problem can be formalized with the following constraint:

$$\leftarrow arc(x, y), clr(x) = clr(y), \quad (3)$$

where arc is of arity $vertex \times vertex$, and clr a unary function of type $vertex \rightarrow color$. A particular instance of the graph coloring problem is specified by giving type definitions for $vertex$ and $color$ along with a set of facts about $arc(x, y)$.

Example 2 The Hamiltonian circuit problem can be formalized by the following rules:

$$\begin{array}{rcl} & \leftarrow & not\, reached(x), \\ & \leftarrow & not\, arc(x, hc(x)), \\ reached(hc(x)) & \leftarrow & initial(x), \\ reached(hc(x)) & \leftarrow & reached(x). \end{array}$$

Here $reached(x)$ and $initial(x)$ are of arity $vertex$, and $hc(x)$ is a unary function of the type $vertex \rightarrow vertex$. An instance of the problem is specified by a domain for $vertex$, and a set of facts for $arc(x, y)$ and a fact for $initial(x)$. This program is more or less a direct "functionalization" of Niemelä's encoding of the same problem using a normal logic program (Niemelä 1999), and is also similar to Cabalar's encoding of the problem in his functional action language[1]

Example 3 The queens problem can be formalized by the following rules:

$$\leftarrow q(x) = q(y), x \neq y,$$
$$\leftarrow |q(x) - q(y)| = |x - y|, x \neq y.$$

[1] http://www.dc.fi.udc.es/∼cabalar/fal/.

Here q is a function of the type $pos \rightarrow pos$, and $q(i)$ is the column where the ith-queen (the one in row i) is to be placed. The expression $x \neq y$ in a rule stands for $not\, x = y$. There are actually two types here: pos, whose domain is $1..N$, and int, whose domain is $-N..N$, for the N-queen problem. The pre-interpreted function "$-$" is of the type $pos \times pos \rightarrow int$ and "$|.|$" of $int \rightarrow int$. These two functions have their standard meanings. This program is essentially the same as Cabalar's encoding of 8-queens problem in his functional action language.

For logic programs without functions, an answer set is a set of atoms that defines the relations in the program: an atom is true iff it is in the answer set. For logic programs with functions, a model needs to define not only relations but also functions in the program. Given a logic program P that may contain functions, an *interpretation* I of P is a mapping that assigns each relation and function symbol in P a meaning in the domains given in the type definitions of P:

- if R is a relation of arity $\tau_1 \times \cdots \times \tau_n$ and the type definitions $\tau_i : D_i$, $1 \leq i \leq n$, are in P, then $R^I \subseteq D_1 \times \cdots \times D_n$.
- if f is a function of type $\tau_1 \times \cdots \times \tau_n \rightarrow \tau_{n+1}$, $n > 1$, and the type definitions $\tau_i : D_i$, $1 \leq i \leq n + 1$, are in P, then f^I is a function from $D_1 \times \cdots \times D_n$ to D_{n+1}. Notice here that a pre-interpreted function should follow its standard interpretation, thus cannot change it meaning from one interpretation to another.

We now define the conditions for an interpretation to be an answer set of a logic program with functions. We basically follow the stable model semantics for logic programs without functions (Gelfond & Lifschitz 1988): we first instantiate all variables in the rules, use the given interpretation to transform the program into one without negation and functions, and check if the resulting program entails the same set of atoms true in the given interpretation.

Given a logic program P, the grounding of P consists of type definitions as in P and the rules that are obtained by replacing variables in the rules of P with elements in their respective domains (recall that we are assuming a many-sorted first-order language, and each variable has a type associated with it). Thus if variable x is of type τ and the domain of τ is D according to the type definitions in P, then x is to be replaced by elements in D. Notice that even after grounding when all variables have been replaced by elements in their respective domains, we still need to keep the type definitions as they are still needed to interpret the functions in the program.

For instance, suppose P is

$$node : \{1, 2\},$$
$$color : \{r, b\},$$
$$arc : node \times node,$$
$$clr : node \rightarrow color,$$
$$brighter(clr(x), clr(y)) \leftarrow arc(x, y).$$

Notice that although the arity of the predicate arc and the type of the function clr are part of the given language, not

the program P, we write them in P for clarity. The grounding of P is

$$
\begin{aligned}
&node : \{1, 2\},\\
&color : \{r, b\},\\
&arc : node \times node,\\
&clr : node \to color,\\
&brighter(clr(1), clr(1)) \leftarrow arc(1, 1),\\
&brighter(clr(1), clr(2)) \leftarrow arc(1, 2),\\
&brighter(clr(2), clr(1)) \leftarrow arc(2, 1),\\
&brighter(clr(2), clr(2)) \leftarrow arc(2, 2).
\end{aligned}
$$

In the following, unless otherwise stated, we shall equate a logic program with its grounding. Thus rules with variables are considered shorthands that will be replaced by their instantiations.

Notice that once a variable in a rule is replaced by objects of a domain, the grounded rules may have symbols not in the original language $\mathcal{L}$. In the following, we let $\mathcal{L}_P$ be the language that extends $\mathcal{L}$ by introducing a new constant for each object that is in the domain of a type, but not a constant in $\mathcal{L}$. These new constants will have the same type as their corresponding objects. Now the fully instantiated rules will be in the language $\mathcal{L}_P$.

Notice that an interpretation I of P can be considered as a first-order structure for $\mathcal{L}_P$: the domains are those specified in the type definitions of P, the relations and functions are interpreted as given by I, and each constant is mapped to itself. In the following, we identify an interpretation of P with its associated first-order structure of $\mathcal{L}_P$ as described above, and when we say that, for example, I is a model of a formula φ, it is to be understood under in this sense.

Given an interpretation I for P, we define the reduction of P under I, written P^I, as the set of rules obtained from P by:

- replace each functional term $f(c_1, ..., c_n)$ in a rule by c if $f^I(c_1, ..., c_n) = c$;
- if a rule contains $c \neq c$ or $c = d$ in its body, where c and d are distinct constants in $\mathcal{L}_P$, then remove the rule;
- if a rule contains $not\, A$ for some A such that A is true under I, then remove this rule;
- remove all equality literals from the bodies of the remaining rules;
- remove all $not\, A$ from the bodies of the remaining rules.

Clearly, P^I is a set of rules that do not have negation, equality, or functions.

Now if P is a program that does not have any constraints, then an interpretation I is an answer set of P if for every *atom in* P, it is true under I iff it is in the least model of P^I. Here an atom $p(c_1, ..., c_n)$ is said to be in P if p occurs in P and $c_i \in D_i$, where D_i is the domain of the ith argument of p as given by the type definitions of P.

Generally, I is an answer set of P if it satisfies the constraints in P and is an answer set of the program obtained from P by removing its constraints.

For instance, let P be

$$
\begin{aligned}
&\tau : \{a, b\},\\
&p, q : \tau,\\
&f : \tau \to \tau,\\
&p(f(x)) \leftarrow not\, q(x),\\
&q(f(x)) \leftarrow not\, p(x).
\end{aligned}
$$

Consider an interpretation I such that

$$\{f^I(a) = b, f^I(b) = a\}.$$

Then P is essentially the following program (in P, replace $f(a)$ by b and $f(b)$ by a):

$$
\begin{aligned}
&p(b) \leftarrow not\, q(a),\\
&p(a) \leftarrow not\, q(b),\\
&q(b) \leftarrow not\, p(a),\\
&q(a) \leftarrow not\, p(b).
\end{aligned}
$$

Thus I is an answer set of P iff

$$
\begin{aligned}
&\{p(x) \mid x \in \{a, b\}, p(x) \text{ is true in } I\} \cup\\
&\{q(x) \mid x \in \{a, b\}, q(x) \text{ is true in } I\}
\end{aligned}
$$

is an answer set of the above program. Now if $f^I(a) = f^I(b) = a$, then P is essentially the following program:

$$
\begin{aligned}
&p(a) \leftarrow not\, q(a),\\
&p(a) \leftarrow not\, q(b),\\
&q(a) \leftarrow not\, p(a),\\
&q(a) \leftarrow not\, p(b).
\end{aligned}
$$

Thus I is an answer set of P if the set of atoms true in it is $\{p(a), q(a)\}$.

Eliminating Functions

As we mentioned in the introduction, functions are not necessary theoretically speaking. They can be eliminated by using relations. We now make this precise.

Let P be a logic program that may have functions. For each function $f : \tau_1 \times \cdots \times \tau_n \to \tau$ in P, we introduce two corresponding relations f_r and $\overline{f}_r$. They both have the arity $\tau_1 \times \cdots \times \tau_n \times \tau$, and informally $f_r(x_1, ..., x_n, y)$ stands for $f(x_1, ..., x_n) = y$ and $\overline{f}_r(x_1, ..., x_n, y)$ for $f(x_1, ..., x_n) \neq y$. Now let $\mathbb{F}(P)$ be the union of the rules obtained by grounding the following rules for each function f in P using the domains in the type definitions of P:

$$
\begin{aligned}
&\leftarrow f_r(x_1, \ldots, x_n, y_1), f_r(x_1, \ldots, x_n, y_2), y_1 \neq y_2,\\
&f_r(x_1, \ldots, x_n, y) \leftarrow not\, \overline{f}_r(x_1, \ldots, x_n, y),\\
&\overline{f}_r(x_1, \ldots, x_n, y) \leftarrow f_r(x_1, \ldots, x_n, z), y \neq z.
\end{aligned}
$$

Let $\mathbb{R}(P)$ be the set of rules obtained from the rules in P by the following transformation:

- Repeatedly replace each functional term $f(u_1, \ldots, u_n)$, where each u_i is a simple term in that it does not mention a function symbol, by a new variable x and add $f_r(u_1, \ldots, u_n, x)$ to the body of the rule where the term appears.

- Ground all the variables introduced in the previous step.

For example, if r is the following rule

$$p(f(g(a)), b) \leftarrow q(g(c)),$$

it will be first transformed into

$$p(f(x), b) \leftarrow q(y), g_r(a, x), g_r(c, y),$$

then into

$$p(z, b) \leftarrow q(y), g_r(a, x), g_r(c, y), f_r(x, z).$$

The variables x and y will be of the same type as the range of g, thus will be instantiated using elements from the range of g, and similarly, the variable z will be instantiated by elements from the range of f.

Clearly $\mathbb{F}(P) \cup \mathbb{R}(P)$ is a normal logic program without functions. This program is equivalent to P:

Theorem 1 *Let P be a normal logic program with functions. An interpretation I is an answer set of P iff $\mathbb{R}(I)$ is an answer set of $\mathbb{F}(P) \cup \mathbb{R}(P)$, where $\mathbb{R}(I)$ is the set of atoms that are true in I:*

$$\begin{aligned}\mathbb{R}(I) \quad = \quad & \{p(\vec{c}) \mid p^I(\vec{c}) \textit{ holds}\} \cup \\ & \{f_r(\vec{c}, a) \mid f^I(\vec{c}) = a\} \cup \{\overline{f}_r(\vec{c}, a) \mid f^I(\vec{c}) \neq a\}.\end{aligned}$$

Cabalar and Lorenzo's Functional Logic Programming

As we mentioned in the introduction, functions in logic programming have mostly been used with a fixed interpretation. Thus if one wants to write a logic program to compute a function, one needs to represent the function by a relation. One noticeable exception is the work of (Cabalar & Lorenzo 2004; Cabalar 2005). Cabalar and Lorenzo (2004) introduced a pure functional logic programming language. Relations are considered as functions with only two possible values, true or false. There is no negation-as-failure operator in the language. Instead, functions can take on default values. This language is extended by Cabalar (2005), and used as an action language.

A major difference between Cabalar and Lorenzo's formalism and ours is that functions can be partial in theirs but must be total in ours. For instance, consider the following program

$$\begin{aligned} & f : \{1\} \rightarrow \{a, b\}, \\ & \leftarrow f(1) = a. \end{aligned}$$

According to our semantics, the unique answer set of this program is $\{f(1) = b\}$. However, the unique model is the empty set according to theirs.

In a sense, one can see the language proposed here as a middle ground between traditional logic programming languages, which encode functions as relations, and the languages of (Cabalar & Lorenzo 2004; Cabalar 2005), which encode relations as functions.

From Programs with Functions to CSPs

For logic programs without functions, answer sets can be computed by SAT solvers using program completions and loop formulas (Lin & Zhao 2004). For programs with functions, the natural alternatives to SAT solvers are CSP solvers. The basic idea is that a functional term is like a variable in a CSP, and can have any value in the range of the function. The constraints will be program completions and loop formulas.

Before delving into the technical details, let's first see some examples. Recall that a CSP is a tuple (X, D, C), where X is a set of variables, D a set of domains, one for each variable in X, and C a set of constraints about the variables in X. A solution to a CSP is an assignment that maps each variable in X to an element in its domain such that under the assignment, all constraints in C are satisfied. Constraints can be given as formulas in a formal language. Abstractively, a constraint can be thought of as a pair $(\vec{x}, S)$, where $\vec{x}$ is a tuple of variables, and S a set of tuples of values in the domains of the variables in $\vec{x}$. Thus an assignment satisfies a constraint $(\vec{x}, S)$ if under the assignment, the tuple of values taken by the variables in $\vec{x}$ is in S.

Consider first the program for the graph coloring problem given earlier. For each vertex n, we introduce a variable corresponding to $clr(n)$ whose domain is the set of given colors, and the constraints are those corresponding to the rules obtained from (3) by instantiating variables x and y with vertices. This corresponds to the standard formulation of the graph coloring as a CSP.

As another example, consider the following program

$$\begin{aligned} & \tau : \{a, b\}, \\ & p, q : \tau, \\ & f : \tau \rightarrow \tau, \\ & p(f(x)) \leftarrow not\, q(x), \\ & q(f(x)) \leftarrow not\, p(x). \end{aligned}$$

For each x in the domain $D = \{a, b\}$, we have two propositional variables $p(x)$ and $q(x)$, and one functional variable $f(x)$ whose domain is also D. The constraints for this program are the sentences in the program completion, i.e. the instantiations of the following formulas on the domain D:

$$\begin{aligned} p(x) & \equiv \bigvee_{y \in D} x = f(y) \wedge \neg q(y), \\ q(x) & \equiv \bigvee_{y \in D} x = f(y) \wedge \neg p(y). \end{aligned}$$

For this example, the program completion is sufficient to capture the answer set semantics. In the general case, we also need loop formulas.

Program Completion

The completion semantics given by Clark (1978) allows functions in a logic program. It can be adapted here straightforwardly.

Recall that here a logic program consists of two parts: a set of rules that may have variables and functions, and a set

of type definitions that specifies a domain for each type, and we have identified such a logic program with its grounding. In the following, we denote by $\mathcal{A}toms(P)$ the set of atoms in P: recall that an atom $p(c_1, ..., c_n)$ is said to be in P if p is a predicate in P, and $c_i \in D_i$, where D_i is the domain of the type of the ith argument of p.

Notice that since pre-interpreted functions have their meanings fixed, and that constants are interpreted by themselves, if a ground term mentions only constants and pre-interpreted functions, then it can be evaluated independent of interpretations.

In the following, given an atomic formula $p(t_1, ..., t_n)$ and an atom $p(c_1, ..., c_n)$ in $\mathcal{A}toms(P)$, we say that the atomic formula $p(t_1, ..., t_n)$ can *cover* the atom $p(c_1, ..., c_n)$ if for each $1 \leq i \leq n$,

- if t_i mentions only constants and pre-interpreted functions, then t_i can be evaluated to c_i;
- if t_i is $f(\vec{s})$ and cannot be evaluated independent of interpretations, then c_i has the same type as the range of f.

Intuitively, this means that under some functional assignments, $p(t_1, ..., t_n)$ may become $p(c_1, ..., c_n)$.

Now let P be a program and $p(\vec{c}) \in \mathcal{A}toms(P)$. The *completion* of $p(\vec{c})$ (w.r.t. P), written *Comp*$(p(\vec{c}), P)$, is the following propositional formula:

$$p(\vec{c}) \leftrightarrow \widehat{Body}_1 \wedge \vec{t}_1 = \vec{c} \vee \ldots \vee \widehat{Body}_n \wedge \vec{t}_n = \vec{c} \qquad (4)$$

where

- $(p(\vec{t}_1) \leftarrow Body_1)$,..., $(p(\vec{t}_n) \leftarrow Body_n)$ are all of the (grounded) rules in P whose heads can cover $p(\vec{c})$;
- $\widehat{Body_i}$ stands for the conjunction of all element in $Body_i$ with "not" replaced by logical negation "$\neg$";
- in general, $\vec{\xi} = \vec{\psi}$ if the two vectors have the same length and their corresponding components are all equal.

Now the *completion* of P is the set of the completions of all atoms in $\mathcal{A}toms(P)$ and the formulas corresponding to the constraints in P.

Loops and loop formulas

We now extend the notions of loops and loop formulas from (Lin & Zhao 2004) to programs that may have functions.

Let P be a program. The *positive dependency graph* of P, written G_P, is the directed graph (V, E), where $V = \mathcal{A}toms(P)$, and for any $p(\vec{c}), q(\vec{d}) \in V$, $(p(\vec{c}), q(\vec{d})) \in E$ if there is a rule r of the form (1) in P such that

- $A = p(\vec{t})$ for some $\vec{t}$, and $p(\vec{t})$ can cover $p(\vec{c})$;
- $B_i = q(\vec{s})$ for some $1 \leq i \leq m$ and $\vec{s}$ such that $q(\vec{s})$ can cover $q(\vec{d})$;
- if the ith element in the above $\vec{t}$ and the kth element in the above $\vec{s}$ are syntactically identical, then the ith element in $\vec{c}$ and the kth element in $\vec{d}$ are also syntactically identical.

The last condition is to make sure that a rule such as

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

generates only dependency edges such as $(p(b), q(b))$ and $(p(c), q(c))$, but not the ones such as $(p(b), q(c))$.

A finite non-empty subset L of V is a *loop* of P if there is a non-zero length cycle that goes through only and all the nodes in L. In other words, the induced subgraph of G_P on L is strongly connected. It's clear that if the given program does not mention any functions, then the above definitions of positive dependency graph and loops are the same as those in (Lin & Zhao 2004).

For instance, if P is the following program

$$\begin{aligned} &\tau : \{a\}, \\ &\mu : \{c_1, \ldots, c_n\}, \\ &f : \tau \rightarrow \mu, \\ &p(f(a)) \leftarrow p(f(a)). \end{aligned}$$

then the loops of P are $\{p(c_1)\}, \ldots, \{p(c_n)\}$.

Given a loop L of P, and an atom $p(\vec{c})$ in L, the *external support formula* of $p(\vec{c})$ w.r.t. L (Lee 2005), written $ES(p(\vec{c}), L, P)$, is the following formula:

$$\bigvee_{1 \leq i \leq n} \left[\widehat{Body}_i \wedge \vec{c} = \vec{t}_i \wedge \bigwedge_{\substack{q(\vec{s}) \in Body_i \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \right], \qquad (5)$$

where $(p(\vec{t}_1) \leftarrow Body_1)$,..., $(p(\vec{t}_n) \leftarrow Body_n)$ are all of the rules in P whose heads can cover $p(\vec{c})$.

The *loop formula* of L in P, written $LF(L, P)$, is then the following formula:

$$\bigvee_{A \in L} A \supset \bigvee_{A \in L} ES(A, L, P). \qquad (6)$$

Notice that since an atom covers itself, our notions of completion, external support and loop formula generalize the corresponding ones for normal logic programs in (Lin & Zhao 2004).

Theorem 2 *Let P be a program. An interpretation I of P is an answer set of P iff it satisfies Comp$(P) \cup LF(P)$, where $LF(P)$ is the set of loop formulas of P.*

From Programs with functions to CSPs

We can now describe our mapping from logic programs with functions to CSPs.

First, we need to assume a certain "normal form" for functional terms in a logic program. In the following, we say that a logic program P is *free of functions in arguments* if all terms that can be evaluated independently of interpretations have been evaluated to constants in $\mathcal{L}_P$, and none of the predicates or functions that are not pre-interpreted have a functional term in their arguments.

Given any logic program P, we can transform it into one that is free of functions in arguments using the following procedure:

- evaluate all terms that mention only constants and pre-interpreted functions to constants;

- for each rule in P, repeatedly replace every occurrence of a functional term $f(u_1, \ldots, u_n)$ in any argument of a predicate or a function that is not pre-interpreted in the rule by a new variable v of the same type as the range of f, and add $f(u_1, \ldots, u_n) = v$ to the end of the body of the rule, where each u_i is a simple term in that it does not mention a function symbol;
- instantiate the rules obtained in the above step.

Example 4 Consider the HC program P in Example 2. The first two steps in the above procedure turns the rules of P into the following rules

$$\begin{array}{lcl} reached(y) & \leftarrow & initial(x), y = hc(x), \\ reached(y) & \leftarrow & reached(x), y = hc(x), \\ & \leftarrow & not\, reached(x), \\ & \leftarrow & not\, arc(x, y), y = hc(x). \end{array}$$

Grounding these rules produces $O(n^2)$ number of rules for a graph with n vertices. In comparison, grounding Niemelä's (1999) program on a graph with n vertices may produce $O(n^3)$ number of rules.

It is clear that this transformation does not introduce any new ground atoms, and the original program and the transformed one are equivalent in the sense that they have the same answer sets. Thus without loss of generality, in the following, we assume that the given logic program is free of functions in arguments.

Given such a logic program P, we translate it to a CSP, denoted by $\mathcal{R}(P) = \langle X, D, C \rangle$, as follows: the set X of variables and their domains D are as follows:

- for each atom p in $\mathcal{A}toms(P)$, there is a variable for it whose domain is $\{0, 1\}$, and
- for each functional term $f(u_1, \ldots, u_n)$ in P such that f is not pre-interpreted, there is a variable for it whose domain is the range of the function f,

the set C of the constraints is as follows: for each formula ϕ in $Comp(P) \cup LF(P)$, there is a constraint $c(\phi) = \langle S, R \rangle$ in C, where R is the constraint obtained from ϕ by replacing atoms and functional terms in it by their corresponding variables, and S is the set of variables occurring in R. Notice that $c(\phi)$ leaves pre-interpreted functions as they are in ϕ.

Under this formulation, the answer sets of a logic program P will correspond to the solutions to its corresponding CSP $\mathcal{R}(P)$ under the following mapping: let I be an interpretation of P, the variable assignment corresponding to I, written $v(I)$, is defined as follows:

- if $x \in X$ corresponds to the atom p, then $v(I)$ assigns x 1 iff p is true in I.
- if $x \in X$ corresponds to the term $f(u_1, \ldots, u_n)$, then $v(I)$ assigns x the value u iff $f^I(u_1, \ldots, u_n) = u$.

Similarly, given a variable assignment of $\mathcal{R}(P)$, a corresponding interpretation of P can be easily computed.

Theorem 3 *Let P be a logic program that is free of functions in arguments, and I an interpretation of P. Then I is an answer set of P iff $v(I)$ is a solution to $\mathcal{R}(P)$.*

Some Experimental Results

Given our translation above from logic programs to CSPs, we can compute the answer sets of logic programs with functions using an algorithm that is similar to the one used by ASSAT (Lin & Zhao 2004), except that we now use a CSP solver instead of a SAT solver.

First of all, notice that our translation from logic programs to CSPs actually has two steps: it first transforms a logic program to a set of quantifier-free sentences in the form of completions and loop formulas, and then from these sentences to CSPs. The second part is actually quite general in that it will work for any quantifier-free sentences that do not have any functional terms in the arguments of predicates and functions, provided the domain of each type is given and finite. We make use of this observation in our Algorithm 1 given below.

Algorithm 1: **FASP**(X) - X stands for a CSP solver

input : A program P
output: An answer set of P if P has one, and report no otherwise
begin

1. $\Sigma \leftarrow Comp(P)$.
2. $\mathcal{R}(\Sigma) \leftarrow$ convert Σ to the format of X (typically in the language CSP2.0 used at the 2006 CSP competition).
3. Find a solution S of $\mathcal{R}(\Sigma)$ by X.
4. If no solution, **return** no answer set.
5. Map S to an interpretation I of P.
6. Compute $M^- = \{p(\vec{c}) | p^I(\vec{c})\ holds\} \setminus \Gamma(P^I)$, where $\Gamma(P^I)$ is the least model of P^I.
7. If $M^- = \emptyset$, **return** I as an answer set.
8. Compute all the maximal loops under M^-, add their loop formulas to Σ, and **goto** step 2.

end

There are a number of available CSP solvers. We tried abscon[2], sugar 0.3[3] with minisat2.0[4], and sat4j-2.0-RC3[5]. They all performed well at the 2006 CSP Competition[6]. The benchmarks that we tried are the HC problem, the graph coloring problem, and the queen's problem. Our encodings of these problems using functions are as given above. We compared FASP(X) with the ASP solvers smodels[7], cmodels[8] with zChaff 2007.3.12, and clasp[9], using the following standard encodings of these problems that do not use functions:

1. For the HC problem, we use two versions, one originally by Niemelä (1999):

[2] http://www.cril.univ-artois.fr/∼lecoutre/research/tools/abscon.html
[3] http://bach.istc.kobe-u.ac.jp/sugar/
[4] http://minisat.se/
[5] http://download.forge.objectweb.org/sat4j/sat4j-2.0-RC3.zip
[6] http://www.cril.univ-artois.fr/CPAI06/
[7] http://www.tcs.hut.fi/Software/smodels/
[8] http://www.cs.utexas.edu/∼tag/cmodels/
[9] http://www.cs.uni-potsdam.de/clasp/

hc(X,Y) :- arc(X,Y), not otherroute(X,Y).
otherroute(X,Y) :-arc(X,Y), arc(X,Z), hc(X,Z), Y != Z.
otherroute(X,Y) :- arc(X,Y), arc(Z,Y), hc(Z,Y), X != Z.
reached(Y) :- arc(X,Y), hc(X,Y), reached(X),
not initialnode(X).
reached(Y) :- arc(X,Y), hc(X,Y), initialnode(X).
initialnode(0).
:- vertex(V), not reached(V).

and the other uses weight constraints:

{in(X,Y)} :- arc(X,Y).
:- 2 {in(X,Y) : arc(X,Y)}, vertex(X).
:- 2 {in(X,Y) : arc(X,Y)},vertex(Y).
r(X) :- in(0,X), vertex(X).
r(Y) :- r(X), in(X,Y), arc(X,Y).
:- not r(X), vertex(X).

2. For the queen's problem:

1{queen(R,C):n(R)}1:-n(C).
:-queen(R,C),queen(R,C1),n(R;C;C1),C<C1.
:-queen(R,C),queen(R1,C1),n(R;R1;C;C1),C<C1,abs(R-R1)==abs(C-C1).

3. For the graph coloring problem:

1 { clrd(V,CL):clr(CL) } 1.
:- edge(V,U), clrd(V,C), clrd(U,C).
edge(X,Y) :- arc(X,Y).
vtx(X) :- vertex(X).

The experimental results are summarized in Tables 1-4, and were done on an AMD server with 4xAMD Opteron 844 (1.8GHz) CPU, 8GB RAM running Fedora Linux Core 3.0 (x86_64 Edition).

In addition to the run times, we also give the sizes of input files. For our FASP, these are the sizes of input files given to our solver, and for the ASP solvers, these are the sizes of the files output by lparse with the options "-d none -t".

Table 1 is for the HC problem on complete graphs, and the running times for the ASP solvers are the ones using the weight constraint encoding. As can be seen, our FASP system was not competitive here. However, notice that the sizes of programs with Niemelä's original encoding are much larger than the ones with the weight constraint or functions. For the HC problem, the weight constraints are able to make the programs much smaller like our encoding with functions.

However, weight constraints are of little help for the queen's problem, at least in the way they are used in the encoding given above. As can be seen in Table 2, while the 300-queen's problem is simply too big in terms of program size for the ASP solvers to handle, our FASP with abscon was still able to handle it using our encoding with functions. Tables 3 and 4 are for the graph coloring problem, one with 3 colors and the other 4 colors. The graphs here are from ASSAT test suites[10]. As can be seen, FASP was competitive here, especially for large graphs. FASP is available at the following URL:

http://www.cse.ust.hk/fasp/

[10]http://assat.cs.ust.hk/Assat-2.0/coloring-2.0.html

In summary, our experiments seem to confirm our expectation that as the problems become large, the standard logic program encodings will produce programs that are much larger than the encodings that use functions, and as the programs become large, the performance of ASP solvers will suffer, giving advantages to solvers like our FASP.

Concluding Remarks

Currently in ASP, to compute a function, one needs to encode it as a relation. This makes the resulting program less direct and leads to large programs when grounded. In this paper we propose to add functions to logic programs, and to extend the answer set semantics and loop formulas to these logic programs. Just as the SAT solvers can be used to compute answer sets of logic programs without functions, we show that CSP solvers can be used to compute answer sets of logic programs with functions. Our experiments seem to show that for problems that require functions, our logic program encodings with functions indeed lead to much smaller ground programs compared to the logic program encodings without functions.

For future work, it is perhaps worthwhile to consider a solver that can work on logic programs with functions directly.

Acknowledgments

We thank the anonymous reviews' comments, especially those on the related work by Cabalar and Lorenzo. This work has been supported in part by the Natural Science Foundations of China under grants 90718009, 60573009, 60703095, and 60496322, and by the Hong Kong RGC CERG 616806.

References

Baselice, S.; Bonatti, P. A.; and Criscuolo, G. 2007. On finitely recursive programs. In *Proceedings of the Twenty Third International Conference on Logic Programming*, 89–103. Porto, Portugal: Springer.

Bonatti, P. A. 2004. Reasoning with infinite stable models. *Artificial Intelligence* 156(1):75–111.

Cabalar, P., and Lorenzo, D. 2004. Logic programs with functions and default values. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence*, 294–306. Lisbon, Portugal: Springer.

Cabalar, P. 2005. A functional action language front-end. In *The Third International Workshop on Answer Set Programming: Advances in Theory and Implementation*. http://www.dc.fi.udc.es/∼cabalar/asp05_C.pdf.

Calimeri, F.; Cozza, S.; and Ianni, G. 2007. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence* 50(3-4):333–361.

Chen, Y.; Lin, F.; Wang, Y.; and Zhang, M. 2006. First-order loop formulas for normal logic programs. In *Proceedings of Tenth International Conference on Principles of Knowledge Representation and Reasoning*, 298–307. Lake District of the United Kingdom: AAAI Press.

Vertices =?	smodels	cmodels	clasp	FASP(X) (LFs/Runs)			size		
				abscon	sat4j	sugar	lparse(n)	lparse(w)	FASP
10	0.020	1.860	0.010	–	–	4.1364(1/2)	46K	7.2K	4.6K
20	0.064	0.800	0.040	–	–	10.3292(1/2	428K	30K	20K
30	0.292	0.380	0.170	–	–	13.4976(1/2)	1.5M	68K	45K
40	0.876	1.310	0.490	–	–	17.2581(1/2)	3.7M	120K	80K
100	69.748	49.310	72.870	–	–	–	60.0M	757K	503K

Legends: LFs - number of loop formulas added; Runs - number of calls for CSP solver; – - No result in 2 hours. lparse(n) - the encoding in normal logic program. lparse(w) - the encoding with weight rules.

Table 1: Hamiltonian Circuit with compete graphs

N=?	smodels	cmodels	clasp	FASP(X)			size	
				abscon	sat4j	sugar	lparse	FASP
20	14.792	0.180	0.060	4.2245	10.4332	14.1777	329K	20K
25	2227.367	0.380	0.160	5.3366	16.1859	18.5702	649K	32K
50	–	3.860	2.140	9.0090	144.6419	133.4165	5.2M	130K
100	–	32.690	23.290	16.0819	#	#	42M	528K
200	–	%	276.020	262.3127	#	#	355M	2.2M
300	–	–	–	465.8901	#	#	>1.0G	5.0M

Legends: # - java.lang.OutOfMemoryError; – - no result in two hours; % - return with Unknown.

Table 2: N-queens

Graph	colorable ?	smodels	cmodels	clasp	FASP(X)			size	
					abscon	sat4j	sugar	lparse	FASP
p3000e13525	y	152.741	191.700	215.460	130.2962	#	86.7627	3.4M	789K
p6000e35946	y	693.495	11651.180	–	1531.4232	#	546.9962	6.9M	1.6M
p10000e10000	y	876.474	3.270	81.590	85.8427	56.8471	19.3543	4.3M	551K
p10000e11000	y	914.969	3.660	78.330	100.8925	64.1679	30.9318	4.4M	592K
p10000e21000	n	6.952	1.950	1.820	18.4261	#	27.3713	5.8M	1007K
p10000e22000	y	1170.997	4.440	48.380	131.6482	#	30.2036	6.0M	1.1M

Legend: pnem - a graph with n nodes and m edges; – - no result in two hours; # - java.lang.OutOfMemoryError.

Table 3: Graph coloring with 4 colors

Graph	colorable ?	smodels	cmodels	clasp	FASP(X)			size	
					abscon	sat4j	sugar	lparse	FASP
p3000e13525	n	3.584	1.670	0.700	11.4414	#	16.2339	2.7M	789K
p6000e35946	n	6.972	3.550	1.600	23.1548	#	33.1880	5.3M	1.6M
p10000e10000	y	380.999	2.650	33.030	74.5373	57.9511	16.9781	3.5M	551K
p10000e11000	y	345.577	3.210	31.260	88.7149	60.7035	25.9871	3.6M	592K
p10000e21000	n	5.572	1.900	1.570	19.2262	#	30.9637	4.6M	1007K
p10000e22000	unknown	–	–	–	–	#	–	4.7M	1.1M

Legends: Same as in Table 2.

Table 4: Graph coloring with 3 colors

Clark, K. L. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logics and Databases*. New York: Plenum Press. 293–322.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, 1070–1080. Seattle, Washington: MIT Press.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 503–508. Edinburgh, Scotland, UK: Professional Book Center.

Lifschitz, V. 1996. Foundations of logic programming. In *Principles of Knowledge Representation*, 69–127. CSLI Publications.

Lin, F., and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by sat solvers. *Artificial Intelligence* 157(1-2):115–137.

Lloyd, J. W. 1987. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Simkus, M., and Eiter, T. 2007. $\mathbb{FDNC}$: Decidable nonmonotonic disjunctive logic programs with function symbols. In *Proceedings of Fourteenth International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 514–530. Yerevan, Armenia: Springer.

Syrjänen, T. 1998. Implementation of local grounding for logic programs with stable model semantics. *Technical Report B18, Digital Systems Laboratory, Helsinki University of Technology*.

Syrjänen, T. 2001. Omega-restricted logic programs. In *Proceedings of Sixth International Conference on Logic Programming and Nonmonotonic Reasoning*, 267–279. Vienna, Austria: Springer.

Appendix: Proofs

Let I be an interpretation. By I^a we denote the set of atoms are true in I, i.e., $\{p(\vec{c})|p^I(\vec{c})\ holds\}$. Let P be a normal logic program with functions and I an interpretation of P. The *functional reduction* of P under I, denoted by $\Phi(P, I)$, is a normal logic program without functions obtained from P by

(1) replacing every term t in P with t^I until there is no functions;

(2) removing the rules whose bodies contain $c \neq c$ or $c = d$ where c and d are two distinct constants;

(3) removing all equality literals from the bodies of the remaining rules.

The following lemma is clear.

Lemma 1 *Let P be a normal logic program with functions and I an interpretation of P. I is an answer set of P iff I^a is an answer set of $\Phi(P, I)$.*

We generalize the splitting notion (Lifschitz 1996) to normal logic programs without functions but possibly containing constraints. In the following, we alternatively write a rule of the form (1) as

$$A \leftarrow \mathit{Pos}, not\, \mathit{Neg}$$

where $\mathit{Pos} = \{B_1, \ldots, B_m\}$, $\mathit{Neg} = \{C_1, \ldots, C_n\}$ and $not\, S = \{not\, a | a \in S\}$ for a given set of atoms S. Given a program without functions P and a set of atoms U where P possibly contains constraints. U *splits* P if, for every rule "$A \leftarrow \mathit{Pos}, not\, \mathit{Neg}$" in P that is not a constraint, $\mathit{Pos} \cup \mathit{Neg} \subseteq U$ whenever $A \in U$. If U splits P then the *base* of P (relative to U), denoted by $b_U(P)$, is the set of rules whose heads belong to U or $\mathit{Pos}(r) \cup \mathit{Neg}(r) \subseteq U$ if r is a constraint. Let P be a program, U a set of atoms and $C \subseteq U$, $e_U(P, C)$ stands for the program obtained from P by

- deleting each rule $A \leftarrow \mathit{Pos}, not\, \mathit{Neg}$ such that $\mathit{Pos} \cap (U \setminus C) \neq \emptyset$ or $\mathit{Neg} \cap C \neq \emptyset$,
- replacing each remaining rule $A \leftarrow Pos, not\, Neg$ by

$$A \leftarrow \mathit{Pos} \setminus U, not\, (\mathit{Neg} \setminus U).$$

Proposition 1 *Let P be a program without functions (possibly with constraints) and U a set of atoms that splits P. A set of atoms M is an answer set of P iff $M = C_1 \cup C_2$ where C_1 is an answer set of $b_U(P)$ and C_2 is an answer set of $e_U(P \setminus b_U(P), C_1)$.*

Proof: Let P' be the nonconstraint rules in P. Note that the difference between $b_U(P)$ and $b_U(P')$ is the constraints of P in which the atoms occur belong to U. Since U splits P thus U splits P' as well.
M is an answer set of P
iff M is an answer set of P' and M satisfies the constraints in P
iff M satisfies the constraints in P and $M = C_1 \cup C_2$ where C_1 is an answer set of $b_U(P')$, and C_2 is an answer set of $e_U(P' \setminus b_U(P'), C_1)$ (**Proposition** 3.10 of (Lifschitz 1996))
iff $M = C_1 \cup C_2$, C_1 is an answer set of $b_U(P)$, and C_2 is an answer set of $e_U(P \setminus b_U(P), C_2)$. ■

Theorem 1 *Let P be a normal logic program with functions. An interpretation I is an answer set of P iff $\mathbb{R}(I)$ is an answer set of $\mathbb{F}(P) \cup \mathbb{R}(P)$, where*

$$\begin{aligned}\mathbb{R}(I) \quad = \quad & I^a \cup \\ & \{f_r(\vec{c}, a) \mid f^I(\vec{c}) = a\} \cup \{\overline{f}_r(\vec{c}, a) \mid f^I(\vec{c}) \neq a\}.\end{aligned}$$

Proof:(sketch) Let $\Xi(P) = \mathbb{R}(P) \cup \mathbb{F}(P)$, $I^f = \mathbb{R}(I) \setminus I^a$ and $U = \mathcal{A}toms(\mathbb{F}(P))$. Note that I gives each function in P a total mapping, thus I^f is evidently an answer set of $\mathbb{F}(P)$ and U splits $\mathbb{F}(P) \cup \mathbb{R}(P)$. In the following, we firstly show,

$$\Phi(P, I) = e_U(\mathbb{R}(P), I^f). \tag{7}$$

Let's consider the following three simple cases for a rule r in P:

- r is of the form "$p(f(a)) \leftarrow Body$" and $Body$ mentions neither functions nor equality. Suppose $f^I(a) = c$, the range of f is $\{c_1, \ldots, c_k\}(k > 0)$. It follows that the rules in $\mathbb{R}(P)$ that are obtained from r include

$$\begin{aligned} p(c_1) &\leftarrow \{f_r(a, c_1)\} \cup Body, \\ &\vdots \\ p(c_k) &\leftarrow \{f_r(a, c_k)\} \cup Body. \end{aligned}$$

  Knowing that $f_r(a, c) \in I^f$ and there is no other c' such that $f^I(a) = c'$ where $c' \neq c$, thus there is no $f_r(a, c') \in I^f$. Therefore,

$$p(c) \leftarrow Body$$

  is the only rule kept in $e_U(\mathbb{R}(P), I^f)$ from the above rules. Clearly, this rule belongs to $\Phi(P, I)$.

- r is of the form "$A \leftarrow \{p(f(a))\} \cup Body$" where A and $Body$ mention neither functions nor equality. Suppose $f^I(a) = c$ again. As the above discussion,

$$A \leftarrow \{p(c)\} \cup Body$$

  is the only rule in $e_U(\mathbb{R}(P), I^f)$ that is obtained from r. This rule is in $\Phi(P, I)$ as well.

- r is of the form "$A \leftarrow \{f(a) = c\} \cup Body$" where A and $Body$ mention neither functions nor equality. It is transformed into:

$$A \leftarrow \{f_r(a, x), x = c\} \cup Body.$$

  Thus we have

$$r' : A \leftarrow Body$$

  belong to $e_U(\mathbb{R}(P), I^f)$ whenever $f^I(a) = c$ iff $r' \in \Phi(P, I)$ by r.

The other case that function occurring in $not\, A$ is similar as above discussion. Note that if "$\leftarrow \emptyset$" is in $\Phi(P, I)$ then "$\leftarrow \emptyset$" must be in $b_U(\Xi(P))$ and vice versa.

I is an answer set of P
iff I^a is an answer set of $\Phi(P, I)$ (Lemma 1)
iff I^a is an answer set of $e_U(\Xi(P) \setminus b_U(\Xi(P)), I^f)$ ((7))
iff $I^a \cup I^f$ is an answer set of $\Xi(P)$ (Proposition 1)
iff $\mathbb{R}(I)$ is an answer set of $\Xi(P)$. ■

Since $\Xi(P)$ mentions no functions and equality, its loops and loop formulas are defined as usual. Given a loop L of $\Xi(P)$ and $A \in L$, we denote its completion, external support and loop formula by *GCOMP*$(\Xi(P))$, *GES*$(A, L, \Xi(P))$ and *GLF*$(L, \Xi(P))$ respectively as in (Chen *et al.* 2006).

Lemma 2 *Let P be a program and I an interpretation of P. $\mathbb{R}(I) \models$ GCOMP$(\Xi(P))$ iff $I \models$ Comp(P).*

Proof:(sketch) Please note that, I is an interpretation of P. It is obviously that $\mathbb{R}(I) \models$ *GCOMP*$(\mathbb{F}(P))$. And due to *GCOMP*$(\Xi(P)) =$ *GCOMP*$(\mathbb{R}(P)) \cup$ *GCOMP*$(\mathbb{F}(P))$, thus it is sufficient to show that

$$\mathbb{R}(I) \models \textit{GCOMP}(\mathbb{R}(P)) \text{ iff } I \models \textit{Comp}(P).$$

Note that, *Atoms*$(\Xi(P)) \setminus$ *Atoms*$(P) =$ *Atoms*$(\mathbb{F}(P))$ and *Atoms*$(\Xi(P)) \setminus$ *Atoms*$(\mathbb{F}(P)) \subseteq$ *Atoms*(P). For any $A \in$ *Atoms*$(P) \setminus$ *Atoms*$(\Xi(P))$, it is not difficult to see that $I \models$ *Comp*(A, P) iff A^I does not hold. Thus, by the informal discussion in the proof of Theorem 1, it is sufficient to show that, for each atom $p(\vec{c}) \in$ *Atoms*$(P) \cap$ *Atoms*$(\Xi(P))$,

$$I \models \textit{Comp}(p(\vec{c}), P) \text{ iff } \mathbb{R}(I) \models \textit{GCOMP}(p(\vec{c}), \Xi(P)).$$

For the sake of clarity, let $\vec{c}$ be c and suppose

$$(p(t_1) \leftarrow Body_1), \ldots, (p(t_k) \leftarrow Body_k)$$

are the rules in P whose heads can cover $p(c)$. Thus the completion of $p(c)$, *Comp*$(p(c), P)$, is the following formula

$$p(c) \equiv \bigvee_{1 \leq i \leq k} t_i = c \wedge \widehat{Body_i}. \tag{8}$$

For the sake of clarity and without loss of generality, let's consider the following cases for the rule $r : p(t_i) \leftarrow Body_i$,

- t_i is identical to $f(a)$ and there is no equality and functions in $Body_i$. In this case, the only one rule obtained from r, that is in $\mathbb{R}(P)$ and whose head is $p(c)$, is the following rule:

$$p(c) \leftarrow \{f_r(a, c)\} \cup Body_i.$$

  Clearly $I \models f(a) = c \wedge \widehat{Body_i}$ iff $\mathbb{R}(I) \models f_r(a, c) \wedge \widehat{Body_i}$. Please note that the equality symbol is regarded as an identity relation as usual.

- t_i is identical to c and $Body_i = \{q(f(a))\} \cup Body$ such that there is no equality and functions in $Body$. Suppose the range of function f is $\{c_1 \ldots, c_m\}$. Now the rules in $\mathbb{R}(P)$ obtained from r are:

$$\begin{aligned} p(c) &\leftarrow \{q(c_1), f_r(a, c_1)\} \cup Body, \\ &\vdots \\ p(c) &\leftarrow \{q(c_m), f_r(a, c_m)\} \cup Body. \end{aligned}$$

  Obviously, $I \models q(f(a)) \wedge \widehat{Body}$ iff $\mathbb{R}(I) \models q(c_1) \wedge f_r(a, c_1) \wedge \widehat{Body} \vee \ldots \vee q(c_m) \wedge f_r(a, c_m) \wedge \widehat{Body}$.

- t_i is identical c and $Body_i = \{f(a) = b\} \cup Body$ where $Body$ mentions no equality and functions. The rule in $\mathbb{R}(P)$ obtained from r is:

$$p(c) \leftarrow \{f_r(a, b)\} \cup Body.$$

  Evidently, $I \models f(a) = b \wedge \widehat{Body}$ iff $\mathbb{R}(I) \models f_r(a, b) \wedge \widehat{Body}$.

The other case that function occurs in $not\, A$ is similar. Therefore, $I \models$ *Comp*(P) iff $\mathbb{R}(I) \models$ *GCOMP*$(\Xi(P))$. ■

Lemma 3 *Let P be a normal logic program and $L \subseteq$ Atoms$(\Xi(P))$. L is a loop of P if L is a loop of $\Xi(P)$.*

Proof: Note that there is no loop of $\Xi(P)$ that contains an atom in *Atoms*$(\mathbb{F}(P))$ and *Atoms*$(\mathbb{R}(P)) \subseteq$ *Atoms*(P). Let the positive dependency graphs of P and $\mathbb{R}(P)$ be (V_1, E_1) and (V_2, E_2) respectively. It suffices to prove, for any two atoms $A, B \in$ *Atoms*$(\Xi(P))$, if $(A, B) \in E_2$ then $(A, B) \in E_1$. It is trivial. ■

Lemma 4 *Let P be a program, I an interpretation of P, $I \models Comp(P)$ and L a loop of $\Xi(P)$. $I \models LF(L,P)$ iff $\mathbb{R}(I) \models GLF(L, \Xi(P))$.*

Proof: For simplicity, let $p(c)$ be an arbitrary atom in L. It suffices to show $I \models ES(p(c), L, P)$ iff $\mathbb{R}(I) \models GES(p(c), L, \Xi(P))$. Suppose

$$(p(t_1) \leftarrow Body_1), \ldots, (p(t_k) \leftarrow Body_k)$$

are the rules in P whose heads can cover $p(c)$. The external support $ES(p(c), L, P)$ is the following formula

$$\bigvee_{1 \le i \le k} \left[\widehat{Body_i} \wedge c = t_i \wedge \bigwedge_{\substack{q(\vec{s}) \in Body_i \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \right]. \qquad (9)$$

Let's consider the cases for rule $r : (p(t_i) \leftarrow Body_i)$:

- r is $(p(f(a)) \leftarrow Body)$ where $Body$ mentions no equality and functions. The only one rule obtained from r in $\mathbb{R}(P)$, whose head is $p(c)$, is the following rule

  $$p(c) \leftarrow \{f_r(a,c)\} \cup Body.$$

  Clearly $Body \cap L = \emptyset$ iff $\bigwedge_{\substack{q(\vec{s}) \in Body \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \equiv \top$. Thus

  $$I \models \left[\widehat{Body} \wedge f(a) = c \wedge \bigwedge_{\substack{q(\vec{s}) \in Body \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \right]$$

  iff $Body \cap L = \emptyset$ and $\mathbb{R}(I) \models f_r(a,c) \wedge \widehat{Body}$.

- r is $p(c) \leftarrow \{q(f(a))\} \cup Body$ where $Body$ mentions no equality and functions. Suppose the domain of the range of f is $\{c_1, \ldots, c_m\}$. Similar to the above discussion, we have

  $$I \models \left[\widehat{Body} \wedge q(f(a)) \wedge \bigwedge_{\substack{q(\vec{s}) \in Body \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \right] \text{ iff}$$

  $\mathbb{R}(I) \models [q(c_1) \wedge f_r(a,c_1) \wedge \widehat{Body} \vee \ldots \vee q(c_m) \wedge f_r(a,c_m) \wedge \widehat{Body}]$ and $Body \cap L = \emptyset$.

- r is $p(c) \leftarrow \{f(a) = b\} \cup Body$ where $Body$ mentions no equality and functions. The only one rule in $\mathbb{R}(P)$ obtained from r is the following rule

  $$p(c) \leftarrow \{f_r(a,c)\} \cup Body.$$

  Thus we have

  $$I \models \left[\widehat{Body} \wedge f(a) = c \wedge \bigwedge_{\substack{q(\vec{s}) \in Body \\ q(\vec{d}) \in L}} \vec{s} \neq \vec{d} \right]$$

  iff $Body \cap L = \emptyset$ and $\mathbb{R}(I) \models f_r(a,c) \wedge \widehat{Body}$.

The other case that function occurs in $not\, A$ is similar. Consequently $I \models LF(L,P)$ iff $\mathbb{R}(I) \models GLF(L, \Xi(P))$. ■

Lemma 5 *Let P be a program, I an interpretation of P such that $I \models Comp(P)$. $\mathbb{R}(I) \models GLF(\Xi(P))$ iff $I \models LF(P)$.*

Proof: By the above two lemmas, it suffices to show that, for any loop L of P that is not a loop of $\Xi(P)$. $I \models LF(L,P)$ if $I \models GLF(\Xi(P))$.

Suppose $L = \{A_1, \ldots, A_n\}$. Since L is not a loop of $\Xi(P)$, there must be an edge $(A_i, A_j)(1 \le i, j \le n)$ of G_P that is not an edge of $G_{\Xi(P)}$. For clarity and without loss of generality, let $A_i = p(c)$ and $A_j = q(d)$. It follows that, for any rule r of P:

$$p(t) \leftarrow Body$$

with $q(s) \in Body$ from which the edge $(p(c), q(d))$ can be derived in G_P, there is no such rule:

$$p(c) \leftarrow Body'$$

with $q(d) \in Body'$ belongs to $\Xi(P)$ that is obtained from r. Thus $\widehat{Body} \wedge t = c$ must be false under any interpretation. Thus L is not actually a loop of P. Now it is easy to see that $I \models LF(L,P)$ by $I \models Comp(P)$ and Lemma 4. ■

Theorem 2 *Let P be a program and I an interpretation of P. I is an answer set of P iff I satisfies $Comp(P) \cup LF(P)$ where $LF(P)$ is the set of loop formulas of P.*

Proof:(sketch)
I is an answer set of P
iff $\mathbb{R}(I)$ is an answer set of $\Xi(P)$ (Theorem 1)
iff $\mathbb{R}(I) \models GCOMP(\Xi(P)) \cup GLF(\Xi(P))$ (Theorem 1 in (Lin & Zhao 2004))
iff $I \models Comp(P) \cup LF(P)$ (Lemmas 2,3 and 5). ■

Lemma 6 *Let ψ be a ground formula without function occurring in ψ as an argument and I an interpretation. Then $I \models \psi$ iff $v(I)$ is a solution of $c(\psi)$.*

Proof: It is clear by induction on structures of formulas. ■

Theorem 3 *Let P be a logic program that is free of functions in arguments, and I an interpretation of P. Then I is an answer set of P iff $v(I)$ is a solution to $\mathcal{R}(P)$.*

Proof: I is an answer set of P
iff $I \models Comp(P) \cup LF(P)$ (Theorem 2)
iff $v(I)$ is a solution of $c(Comp(P) \cup LF(P))$ (Lemma 6)
iff $v(I)$ is a solution of $\mathcal{R}(P)$. ■