Knowledge of Uncertain Worlds: Programming with Logical Constraints

Yanhong A. Liu and Scott D. Stoller

Computer Science Department, Stony Brook University, Stony Brook, New York, USA
{liu,stoller}@cs.stonybrook.edu

Abstract. Programming with logic for sophisticated applications must deal with recursion and negation, which have created significant challenges in logic, leading to many different, conflicting semantics of rules. This paper describes a unified language, DA logic, for design and analysis logic, based on the unifying founded semantics and constraint semantics, that support the power and ease of programming with different intended semantics. The key idea is to provide meta-constraints, support the use of uncertain information in the form of either undefined values or possible combinations of values, and promote the use of knowledge units that can be instantiated by any new predicates, including predicates with additional arguments.

Keywords: Datalog, unrestricted negation and quantification, meta-constraints, founded semantics, constraint semantics, knowledge unit

1 Introduction

Programming with logic has allowed many design and analysis problems to be expressed more easily and clearly at a high level. Examples include problems in program analysis, network management, security frameworks, and decision support. However, when sophisticated problems require reasoning with negation and recursion, possibly causing contradiction in cyclic reasoning, programming with logic has been a challenge. Many languages and semantics have been proposed, but they have different, conflicting underlying assumptions that are subtle and do not work for all problems.

This paper describes a unified language, DA logic, for design and analysis logic, for programming with logic using logical constraints. It supports logic rules with unrestricted negation in recursion, as well as unrestricted universal and existential quantification. It is based on the unifying founded semantics and constraint semantics, and it supports the power and ease of programming with different intended semantics without causing contradictions in cyclic reasoning.

- The language provides meta-constraints on predicates. These meta-constraints capture the different underlying assumptions of different logic language semantics.
- The language supports the use of uncertain information in the results of different semantics, in the form of either undefined values or possible combinations of values.

– The language further supports the use of knowledge units that can be instantiated by any new predicates, including predicates with additional arguments.

Together, the language allows complex problems to be expressed clearly and easily, where different assumptions can be easily used, combined, and compared for expressing and solving a problem modularly, unit by unit.

We present examples from different games that show the power and ease of programming with DA logic.

2 Need of easier programming with logic

We discuss the challenges of understanding and programming with negation and recursion. We use a small well-known example, the win-not-win game, for illustration.

Consider the following rule, called the win rule. It says that `x` is a winning position if there is a move from `x` to `y` and `y` is not a winning position.

```
win(x) ← move(x,y) ∧ ¬ win(y)
```

This seems to be a reasonable rule, because it captures the rule for winning for many games, including in chess for the King to not be captured, giving winning, losing, and draw positions. However, there could be potential problems. For example if there is a `move(1,1)` for some position `1`, then the win rule would imply: `win(1)` if not `win(1)`, and thus the truth value of `win(1)` becomes unclear.

Inductive definitions. Instead of the single win rule, one could use the following three rules to determine the winning, losing, and draw positions.

```
win(x) ← ∃ y | move(x,y) ∧ lose(y)
lose(x) ← ∀ y | ¬ move(x,y) ∨ win(y)
draw(x) ← ¬ win(x) ∧ ¬ lose(x)
```

The first two rules form inductive definitions [14, 6], avoiding the potential problems of the single win rule. The base case is the set of positions that have no moves to any other position and thus are losing positions. With winning and losing positions defined, the draw positions are those in cycles of moves that have no moves to losing positions.

However, clearly, these rules are much more cumbersome than the single win rule.

Well-founded semantics. Indeed, with well-founded semantics (WFS) [25], which computes a 3-valued model, the single win rule above gives `win(x)` being True, False, or Unknown for each `x`, corresponding to `x` being a winning, losing, or draw position, respectively. However, `win(x)` being 3-valued does not allow the three outcomes to be used as three predicates or sets for further computation; the three predicates defined by the three rules do allow this.

For example, there is no way to use the Unknown positions explicitly, say to find all reachable nodes following another kind of moves from draw positions. One might try to do it by adding two additional rules to the single win rule:

```
lose(x) ← ¬ win(x)
draw(x) ← ¬ win(x) ∧ ¬ lose(x)
```

However, the result is that `draw(x)` is False for all positions that `win(x)` is True or False, and is Unknown for all draw positions.

Stable model semantics. Stable model semantics (SMS) [13] computes a set of 2-valued models, instead a single 3-valued model. It has been used for solving many constraint problems in answer set programming (ASP), because its set of 2-valued models can provide the set of satisfying solutions.

For example, for the singl win rule, if besides winning and losing positions, there is a separate cycle of even length, say `move(1,2)` and `move(2,1)`, then instead of `win` being Unknown for `1` and `2` as in WFS, SMS returns two models: one with `win` being True for `1` and other winning positions but not `2`, and one with `win` being True for `2` and other winning positions but not `1`. This is a very different interpretation of the win-not-win rule.

However, for the single rule above, when there are draw positions, SMS may also return just an empty set, that is, a set with no models at all. For example, if besides winning and losing positions, there is a separate cycle of moves of odd length, say simply `move(1,1)`, then SMS returns simply the empty set. This is clearly undesired for the win-not-win game.

Founded semantics and constraint semantics. Founded semantics and constraint semantics [18] unify different prior semantics. It allows different underlying assumptions to be specified for each predicate, and computes the desired semantics as a simple least fixed point to return a 3-valued model and, if there are undefined values, as constraint solving to return a set of 2-valued models.

For the win-not-win game, one can write the single win rule, with the default assumption that `win` is *complete*, that is, the win rule is the only rule that infers `win`, which is an implicit assumption underlying WFS and SMS.

- With founded semantics, the three rules that use inductive definitions can be automatically derived, and True, False, and Undefined positions for `win` are inferred, corresponding to the three predicates from inductive definitions and the 3-valued results from WFS.
- Then constraint semantics, if desired, computes all combinations of True and False values for the Undefined values for the draw positions, that satisfy all the rules as constraints. It equals SMS for the single win rule.

Both WFS and SMS also assume that if nothing is said about some `p`, then `p` is false. When this is not desired, some programming tricks are used to get around it. For example, with SMS, to allow `p` to be possibly true in some models, one can introduce some new `q` and two new rules as below, to make it possible that, in some models, `p` is true and `q` is false.

```
p ← ¬ q
q ← ¬ p
```

Founded semantics and constraint semantics allow `p` to be simply declared as *uncertain*.

Both WFS and SMS also assume that if all ways that can infer `p` require using `p` in the condition of some rule, then `p` is false. Founded semantics and constraint semantics allow this reasoning to be used where desired, by applying it if `p` is declared as *closed*.

Founded semantics and constraint semantics also allow unrestricted universal and existential quantifications and unrestricted nesting of Boolean conditions; these are not supported in WFS and SMS.

However, founded semantics and constraint semantics alone do not address how to use different semantics seamlessly in a single logic program.

Programming with logical constraints. Because different assumptions and semantics help solve different problems or different parts of a problem, easier programming with logic requires supporting all assumptions and semantics in a simple and integrated design.

This paper treats different assumptions as different meta-constraints for expressing a problem or parts of a problem, and support results from different semantics to be used easily and directly. For the win-not-win example:

- We name the positions for which `win` is true, false, and undefined in founded semantics using three predicates, `win.T`, `win.F`, and `win.U`, corresponding exactly to the inductively defined `win`, `lose`, and `draw`. These predicates can be used explicitly and directly for further reasoning, unlike with the truth values of WFS or founded semantics.
- We let `CS` be the constraint semantics of a set of rules and facts. For `m` $\in$ `CS`, we use `m.win(x)` to denote the truth value of `win(x)` in model `m`. Predicate `CS(m)` means exactly `m` $\in$ `CS` and can be used directly for further reasoning, unlike the set of models in SMS or constraint semantics.

Table 1 summarizes the meta-constraints that can be used to express different assumptions, corresponding declarations and resulting predicates in founded semantics and constraint semantics, and corresponding other prior semantics if all predicates use the same meta-constraint. Columns 2 and 4 are presented and proved in our prior work [18]. Columns 1 and 3 are introduced in DA logic.

More fundamentally, we must enable easy specification of problems with reusable parts and where different parts may use different assumptions and semantics. To that end, we support instantiation and re-use of existing parts, and allow predicates in any existing parts to be bound to other given predicates, including predicates with additional arguments.

Even with all this power, DA logic is decidable, because it does not include function symbols and is over finite domains.

3 DA logic

This section presents the syntax and informal meaning of DA logic, for design and analysis logic. The constructs described in the paragraphs on "Conjunctive

Meta-constraint on Predicate P	Founded/Constraint Semantics		Other Prior Semantics
	Declarations on P	Resulting Predicates	
certain(P)	certain	P.T, P.F	Stratified (Perfect, Inductive Definition)
open(P)	uncertain, not complete	P.T, P.F, P.U	
		$m.P$ for $m \in K$.CS	First-Order Logic
complete(P)	uncertain, complete	as above	Fitting (Kripke-Kleene)
			Supported
closed(P)	uncertain, complete, closed	as above	WFS
			SMS

Table 1. Meta-constraints and corresponding prior semantics.

rules with unrestricted negation", "Disjunction", and "Quantification" appear in our prior work on founded semantics and constraint semantics [18]. The other features are new.

Knowledge unit. A *program* is a set of knowledge units. A *knowledge unit*, abbreviated as *kunit*, is a set of rules, facts, and meta-constraints, defined below. The definition of a kunit has the following form, where K is the name of the kunit, and *body* is a set of rules, facts, meta-constraints, and instantiations of other kunits:

```
kunit K:
  body
```

The scope of a predicate is the kunit in which it appears. Predicates with the same name, but appearing in different kunits, are distinct.

Example. A kunit for the single win rule is

```
kunit win_unit:
  win(x) ← move(x,y) ∧ ¬ win(y)
```

■

Kunits provide structure and allow knowledge to be re-used in other contexts by instantiation, as described below.

Conjunctive rules with unrestricted negation. We first present a simple core form of logic rules and then describe additional constructs that can appear in rules. The core form of a rule is the following, where any P_i may be preceded with $\neg$:

$$Q(X_1, ..., X_a) \leftarrow P_1(X_{11}, ..., X_{1a_1}) \wedge ... \wedge P_h(X_{h1}, ..., X_{ha_h}) \quad (1)$$

Q and the P_i are predicates, each argument X_k and X_{ij} is a constant or a variable, and each variable in the arguments of Q must also be in the arguments of some P_i. In arguments of predicates in example programs, we use numbers for constants and letters for variables.

If $h = 0$, there is no P_i or X_{ij}, and each X_k must be a constant, in which case $Q(X_1, ..., X_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to

the case where $h \geq 1$, in which case the left side of the backward implication is called the *conclusion*, the right side is called the *body*, and each conjunct in the body is called a *hypothesis*.

These rules have the same syntax as in Datalog with negation, but are used here in a more general setting, because variables can range over complex values, such as constraint models, as described below.

Predicates as sets. We use a syntactic sugar in which a predicate P is also regarded as the set of x such that $P(x)$ holds. For example, we may write `move = {(1,2), (1,3)}` instead of the facts `move(1,2)` and `move(1,3)`; to ensure the equality holds, this shorthand is used only when there are no other facts or rules defining the predicate.

Disjunction. The hypotheses of a rule may be combined using disjunction as well as conjunction. Conjunction and disjunction may be nested arbitrarily.

Quantification. Existential and universal quantifications in the hypotheses of rules are written using the following notations:

$$\begin{array}{ll} \exists\ X_1,\ ...,\ X_n \mid Y & \text{existential quantification} \\ \forall\ X_1,\ ...,\ X_n \mid Y & \text{universal quantification} \end{array} \tag{2}$$

In quantifications of this form, the domain of each quantified variable is the set of all constants in the containing kunit. As syntactic sugar, a domain can be specified for a quantified variable, using a unary predicate regarded as a set. For example, $\exists$ `x` $\in$ `win` $\mid$ `move(x,x)` is syntactic sugar for $\exists$ `x` $\mid$ `win(x)` $\wedge$ `move(x,x)`, and $\forall$ `x in win` $\mid$ `move(x,x)` is syntactic sugar for $\forall$ `x` $\mid \neg$ `win(x)` $\vee$ `move(x,x)`.

Meta-constraints. Assumptions about predicates are indicated in programs using the meta-constraints in the first column of Table 1. Each meta-constraint specifies the declarations listed in the second column of Table 1. For example, if a kunit contains `open(`P`)`, we say that P is declared uncertain and incomplete in that kunit. In each kunit, at most one meta-constraint may be given for each predicate.

A predicate declared *certain* means that each assertion of the predicate has a unique true (T) or false (F) value. A predicate declared *uncertain* means that each assertion of the predicate has a unique true, false, or undefined (U) value. A predicate declared *complete* means that all rules with that predicate in the conclusion are given in the containing kunit. A predicate declared *closed* means that an assertion of the predicate is made false, called *self-false*, if inferring it to be true using the given rules and facts requires assuming itself to be true.

A predicate in the conclusion of a rule is said to be *defined* using the predicates or their negation in the hypotheses of the rule, and this defined-ness relation is transitive. A predicate must be declared uncertain (using one of the corresponding meta-constraints) if it is defined transitively using its own negation, or is defined using an uncertain predicate; otherwise, it may be declared certain or uncertain and is by default certain. A predicate may be declared complete or not only if it is uncertain, and it is by default complete. If a meta-constraint is not given for a predicate, these default declarations apply.

Using kunits with instantiation. The body of a kunit K_1 can use another kunit K using an instantiation of the form:

$$\texttt{use}\ K\ (P_1 = Q_1(Y_{1,1}, ..., Y_{1,b_1}),\ ...,\ P_n = Q_n(Y_{n,1}, ..., Y_{n,b_n})) \tag{3}$$

This has the same effect as applying the following substitution to the body of K and inlining the result in the body of K_1: for each i in $1..n$, replace each occurrence $P_i(X_1, ..., X_a)$ of predicate P_i with $Q_i(X_1, ..., X_a, Y_{i,1}, ..., Y_{i,b_i})$. Note that arguments of Q_i specified in the `use` construct are appended to the argument list of each occurrence of P_i in K, hence the number of such arguments must be $\text{arity}(Q_i) - \text{arity}(P_i)$. The check for having at most one meta-constraint per predicate, and the determination of default declarations, are performed after expansion of all `use` constructs. A kunit K_1 has a *use-dependency* on kunit K if K_1 uses K. The use-dependency relation must be acyclic.

Example. For the example kunit `win_unit` given earlier in this section, the following kunit is an instantiation of the win-not-win game with different predicates for moving and winning:

```
kunit win2_unit:
  use win_unit (move = move2, win = win2)
```

■

In some logic programming languages, including our prior work on founded semantics [18], a program is an unstructured set of rules and facts. The structure and re-use provided by kunits is vital for development of larger programs for practical applications.

Referencing founded semantics. The founded semantics of a predicate P can be referenced using special predicates P.`T`, P.`F`, and P.`U`. For each of the three truth values t, $P.t(c_1, ..., c_n)$ is true if $P(c_1, ..., c_n)$ has truth value t, and is false otherwise. To ensure that the semantics of P is fully determined before these predicates are used, these predicates cannot be used in rules defining P or any predicate on which P depends. Predicates that reference founded semantics are implicitly declared certain and can appear only in rule bodies.

When referencing the undefined part of a predicate, it is sometimes desirable to prune uninteresting values. For example, consider the rule `draw(x)` ← `win.U(x)`. If the kunit contains constants representing players as well as positions, and `win(`X`)` is undefined when X is a player, and the user wants `draw` to hold only for positions, then the user could add to the rule an additional hypothesis `position(x)`, defined to hold only for positions.

Referencing constraint semantics. The constraint semantics of a kunit K can be referenced in another kunit K_1 using the special predicate K.`CS`, where K is the name of another kunit in the program. Using this special predicate in any rule in K_1 has the effect of adding all of the constraint models of K to the domain (that is, set of constants) of K_1. In other words, the possible values of variables in K_1 include the constraint models of K. The assertion K.`CS`(X) is true when X is a constraint model of K and is false for all other constants. The constraint models of a kunit K can be referenced using K.`CS` only if K does

not reference its own founded semantics (using predicates such as P.U). When the value of a variable X is a constraint model of K, a predicate P of K can be accessed using the notation $X.P(...)$. If the value of X is not a constraint model, or P is not a predicate defined in that constraint model, then $X.P(...)$ is undefined, regardless of the arguments. Predicates that reference constraint semantics are implicitly declared certain and can appear only in rule bodies. A kunit K_1 has a *CS-dependency* on another kunit K if K_1 uses K.CS. The CS-dependency relation must be acyclic.

4 Formal definition of semantics of DA logic

This section extends the definitions of founded semantics and constraint semantics in [18] to handle the new features of DA logic.

Handling kunits is relatively straightforward. Since each kunit defines a distinct set of predicates, the founded semantics of the program is simply a collection of the founded semantics of its kunits, and similarly for the constraint semantics. All `use` constructs in a kunit are expanded, as described in Section 3, before considering its semantics. Therefore, the constants, facts, rules, and meta-constraints of a kunit include the corresponding elements (appropriately instantiated) of the kunits it uses.

Handling references to founded semantics and constraint semantics requires changes in the definitions of domain, literal, interpretation, and dependency graph.

Handling disjunction, which is mentioned as an extension in [18] but not considered in the detailed definitions, requires changes in the definition of completion rules and the handling of closed predicates.

The paragraphs "Founded semantics of DA logic without closed declarations", "Least fixed point", and "Constraint semantics of DA logic" are essentially the same as in [18]; they are included for completeness.

Atoms, literals, and projection. Let π be a program. Let K be a kunit in π. A predicate is *intensional* in K if it appears in the conclusion of at least one rule in K; otherwise, it is *extensional* in K. The *domain* of K is the set of constants in K plus, for each kunit K_1 such that K_1.CS appears in K, the constraint models of K_1, computed as defined below. The requirement that the CS-dependency relation is acyclic ensures the constraint models of K_1 are determined before the semantics of K is considered.

An *atom* of K is a formula $P(c_1, ..., c_a)$ formed by applying a predicate P in K with arity a to a constants in the domain of K. A *literal* of K is a formula of the form $P(c_1, ..., c_a)$ or $P.\mathtt{F}(c_1, ..., c_a)$, for any atom $P(c_1, ..., c_a)$ of K where P is a predicate that does not reference founded semantics or constraint semantics. These are called *positive literals* and *negative literals* for $P(c_1, ..., c_a)$, respectively. A set of literals is *consistent* if it does not contain positive and negative literals for the same atom. The *projection* of a kunit K onto a set S of predicates, denoted $Proj(K, S)$, contains all facts of K for predicates in S and all rules of K whose conclusions contain predicates in S.

Interpretations, ground instances, models, and derivability. An *interpretation* I of K is a consistent set of literals of K. Interpretations are generally 3-valued. For predicates that does not reference founded or constraint semantics, $P(c_1, ..., c_a)$ is *true* (T) in I if I contains $P(c_1, ..., c_a)$, is *false* (F) in I if I contains $P.\texttt{F}(c_1, ..., c_a)$, and is *undefined* (U) in I if I contains neither $P(c_1, ..., c_a)$ nor $P.\texttt{F}(c_1, ..., c_a)$. For the predicates that reference founded semantics, for each of the three truth values t, $P.t(c_1, ..., c_a)$ is true in I if $P(c_1, ..., c_a)$ has truth value t in I, and is false otherwise. For the predicates that reference constraint semantics, $K_1.\texttt{CS}(c)$ is true in I if c is a constraint model of K_1, as defined below, and is false otherwise; the requirement that the CS-dependency relation is acyclic ensures that the constraint models of K_1 are determined before the semantics of $K_1.\texttt{CS}(c)$ is considered. If c is a constraint model that provides a truth value for $P(c_1, ..., c_a)$, then $c.P(c_1, ..., c_a)$ has the same truth value in I that $P(c_1, ..., c_a)$ has in c, otherwise it is undefined. An interpretation I of K is *2-valued* if every atom of K is true or false in I, that is, no atom is undefined. Interpretations are ordered by set inclusion $\subseteq$.

A *ground instance* of a rule R is any rule that can be obtained from R by expanding universal quantifications into conjunctions over all constants in the domain, instantiating existential quantifications with constants, and instantiating the remaining variables with constants. An interpretation is a *model* of a kunit if it contains all facts in the kunit and satisfies all rules of the kunit, interpreted as formulas in 3-valued logic [10], that is, for each ground instance of each rule, if the body is true, then so is the conclusion. A collection of interpretations, one per kunit in a program π, is a *model* of π if each interpretation is a model of the corresponding kunit.

The *one-step derivability* operator T_K performs one step of inference using rules of K, starting from a given interpretation. Formally, $C \in T_K(I)$ iff C is a fact of K or there is a ground instance R of a rule in K with conclusion C such that the body of R is true in I.

Dependency graph. The *dependency graph* $DG(K)$ of kunit K is a directed graph with a node for each predicate of K that does not reference founded semantics and constraint semantics (including these predicates is unnecessary, because they cannot appear in conclusions), and an edge from Q to P labeled $+$ (respectively, $-$) if a rule whose conclusion contains Q has a positive (respectively, negative) hypothesis that contains P. If the node for predicate P is in a cycle containing only positive edges, then P has *circular positive dependency* in K; if it is in a cycle containing a negative edge, then P has *circular negative dependency* in K.

Founded semantics of DA logic without closed declarations. We first define a version of founded semantics, denoted $Founded_0$, that does not take declarations of predicates as closed into account; below we extend the definition to handle those declarations. Intuitively, the *founded model* of a kunit K ignoring closed declarations, denoted $Founded_0(K)$, is the least set of literals that are given as facts or can be inferred by repeated use of the rules. We de-

fine $Founded_0(K) = LFPbySCC(NameNeg(Cmpl(K)))$, where functions $Cmpl$, $NameNeg$, and $LFPbySCC$, are defined as follows.

Completion. The completion function, $Cmpl(K)$, returns the *completed* version of K. Formally, $Cmpl(K) = AddInv(Combine(K))$, where $Combine$ and $AddInv$ are defined as follows.

The function $Combine(K)$ returns the kunit obtained from K by replacing the facts and rules defining each uncertain complete predicate Q with a single *combined rule* for Q that is logically equivalent to those facts and rules. The detailed definition of combined rule is the same as in [18], except generalized in a straightforward way to allow rule bodies to contain disjunction and quantifiers. Similar completion rules are used in [5, 10].

The function $AddInv(K)$ returns the kunit obtained from K by adding, for each uncertain complete predicate Q, a *completion rule* that derives negative literals for Q. The completion rule for Q is obtained from the inverse of the combined rule defining Q (recall that the inverse of $C \leftarrow B$ is $\neg C \leftarrow \neg B$), by putting the body of the rule in negation normal form, that is, using equivalences of predicate logic to move negation inwards and eliminate double negations, so that negation is applied only to atoms.

Least fixed point. Explicit use of negation is eliminated before the least fixed point is computed, by applying the function $NameNeg$. The function $NameNeg(K)$ returns the kunit obtained from K by replacing each $\neg P(X_1, ..., X_a)$ with $P.\mathtt{F}(X_1, ..., X_a)$.

The function $LFPbySCC(K)$ uses a least fixed point to infer facts for each strongly connected component (SCC) in the dependency graph of K, as follows. Let $S_1, ..., S_n$ be a list of the SCCs in dependency order, so earlier SCCs do not depend on later ones; it is easy to show that any linearization of the dependency order leads to the same result for $LFPbySCC$. For convenience, we overload S_i to also denote the set of predicates in the SCC S_i. Define $LFPbySCC(K) = I_n$, where $I_0 = \emptyset$ and $I_i = AddNeg(LFP(T_{I_{i-1} \cup Proj(K,S_i)}), S_i)$ for $i \in 1..n$. $LFP(f)$ is the least fixed point of function f. The least fixed point is well-defined, because $T_{I_{i-1} \cup Proj(K,S_i)}$ is monotonic, because the kunit K was transformed by $NameNeg$ and hence does not contain negation. The function $AddNeg(I, S)$ returns the interpretation obtained from interpretation I by adding *completion facts* for certain predicates in S to I; specifically, for each such predicate P, for each combination of values $v_1, ..., v_a$ of arguments of P, if I does not contain $P(v_1, ..., v_a)$, then add $P.\mathtt{F}(v_1, ..., v_a)$.

Founded semantics of DA logic with closed declarations. Informally, when an uncertain complete predicate of kunit K is declared *closed*, an atom A of the predicate is false in an interpretation I, called *self-false* in I, if every ground instance of rules that concludes A, or recursively concludes some hypothesis of that rule instance, has a hypothesis that is false or, recursively, is self-false in I. A formal definition of $SelfFalse_K(I)$, the set of self-false atoms of kunit K with respect to interpretation I, appears in [18]; it is the same as the definition of greatest unfounded set [25], except limited to closed predicates. The definition

does not take disjunction into account, so each rule containing disjunction is put into disjunctive normal form (DNF) and then replaced with multiple rules (one per disjunct of the DNF) not containing disjunction, before determining the self-false atoms.

The founded semantics is defined by repeatedly computing the semantics given by $Founded_0$ (the founded semantics without closed declarations) and then setting self-false atoms to false, until a least fixed point is reached. For a set S of positive literals, let $\neg \cdot S = \{P.\mathtt{F}(c_1, ..., c_a) \mid P(c_1, ..., c_a) \in S\}$. For a kunit K and an interpretation I, let $K \cup I$ denote K with the literals in I added to its body. Formally, the founded semantics is $Founded(K) = LFP(F_K)$, where $F_K(I) = Founded(K \cup I) \cup \neg \cdot SelfFalse_K(Founded(K \cup I))$.

Constraint semantics of DA logic. Constraint semantics is a set of 2-valued models based on founded semantics. A *constraint model* of K is a consistent 2-valued interpretation M of K such that M is a model of $Cmpl(K)$ and such that $Founded(K) \subseteq M$ and $\neg \cdot SelfFalse_K(M) \subseteq M$. Let $Constraint(K)$ denote the set of constraint models of K. Constraint models can be computed from $Founded(K)$ by iterating over all assignments of true and false to atoms that are undefined in $Founded(K)$, and checking which of the resulting interpretations satisfy all rules in $Cmpl(K)$ and satisfy $\neg \cdot SelfFalse_K(M) \subseteq M$.

Properties of DA logic semantics. The following theorems express the most important properties of the semantics.

Theorem 1. *The founded model and constraint models of a program π are consistent.*

Proof: First we consider founded semantics. Each kunit in the program defines a distinct set of predicates, so consistency can be established one kunit at a time, considering them in CS-dependency order. For each kunit K, the proof of consistency is a straightfoward extension of the proof of consistency of founded semantics [17, Theorem 1]. The extension is needed to show that consistency holds for the new predicates that reference founded semantics and constraint semantics.

For predicates that reference founded semantics, we prove this for each SCC S_i in the dependency graph for K; the proof is by induction on i. The predicates used in SCC S_i to reference founded semantics have the same truth values as the referenced predicates in earlier SCCs, and by the induction hypothesis, the interpretation computed for predicates in earlier SCCs is consistent.

For predicates that reference constraint semantics, the proof is by induction on the kunits in CS-dependency order. The predicates used in kunit K to reference constraint semantics have the same truth values as the referenced predicates in earlier kunits, and by the induction hypothesis, the interpretation computed for predicates in earlier kunits is consistent.

For constraint semantics, note that constraint models are consistent by definition. ■

Theorem 2. *The founded model of a kunit K is a model of K and $Cmpl(K)$. The constraint models of K are 2-valued models of K and $Cmpl(K)$.*

Proof: The proof that *Founded*(K) is a model of *Cmpl*(K) is essentially the same as the proof that *Founded*(π) is a model of *Cmpl*(π) [17, Theorem 2], because the proof primarily depends on the behavior of *Cmpl*, *AddNeg*, and the one-step derivability operator, and they handle atoms of predicates that reference founded semantics and constraint semantics in exactly the same way as other atoms. Constraint models are 2-valued models of *Cmpl*(K) by definition. Any model of *Cmpl*(K) is also a model of K, because K is logically equivalent to the subset of *Cmpl*(K) obtained by removing the completion rules added by *AddInv*. ■

Theorem 3. *DA logic is decidable.*

Proof: DA logic has a finite number of constants from given facts, and has sets of finite nesting depths bounded by the depths of CS-dependencies. In particular, it has no function symbols to build infinite domains in recursive rules. Thus, DA logic is over finite domains and is decidable. ■

5 Additional examples

We present additional examples that show the power of our language. They are challenging or impossible to express and solve using prior languages and semantics. We use `--` to prefix comments.

Same different games. The same win-not-win game can be over different kinds of moves, forming different games, as introduced with kunit instantiation. However, the fundamental winning, losing, or draw situations stay the same, parameterized by the moves. The moves could also be defined easily using another kunit instantiation.

Example. A new game can use winning, losing, draw positions defined by `win_unit` in Section 2, whose moves use paths defined by `path_unit`, whose edges use given links.

```
kunit path_unit:
  path(x,y) ← edge(x,y)
  path(x,y) ← edge(x,z) ∧ path(z,y)

kunit win_path_unit:
  link = {(1,2), (1,3), ...}   -- shorthand for link(1,2), link(1,3), ...
  use path_unit (edge = link)  -- instantiate path_unit with edge replaced
                               -- by link
  use win_unit (move = path)   -- instantiate win_unit with move replaced
                               -- by path
```

One could also define `edge` in place of `link` above, and then `path_unit` can be used without rebinding the name `edge`, as follows.

```
kunit win_path_unit:           -- as above
  edge = {(1,2), (1,3), ...}   -- as above but use edge in place of link
  use path_unit ()             -- as above but without replacing edge by
                               -- link
  use win_unit (move = path)   -- as above
```

■

Defined from undefined positions. Sets and predicates can be defined using the set of values of arguments for which a given predicate is undefined. This is not possible in previous 3-valued logic like WFS, because anything depending on undefined can only be undefined.

Example. Using the win-not-win game, the predicates `move_to_draw` and `reach_from_draw` below define the set of positions that have a move to a draw position, and the set of positions that have a special move from a draw position, respectively.

```
kunit draw_unit:
  move = {(1,1), (2,3), (3,1)}
  use win_unit ()

  move_to_draw(x) ← move(x,y) ∧ win.U(y)

  special_move = {(1,4), (4,2)}
  use path_unit (edge = special_move)

  reach_from_draw(y) ←  win.U(x) ∧ path(x,y)
```

In `draw_unit`, we have `win.U(1)`, that is, 1 is a draw position. Then we have `move_to_draw(3)`, and we have `reach_from_draw(4)` and `reach_from_draw(2)`.

Note that we could copy the single win rule here in place of `use win_unit ()` and obtain an equivalent `draw_unit`. We avoid copying when possible because this is a good principle, and in general, a kunit may contain many rules and facts. ■

Unique undefined positions. Among the most critical information is information that is true in all possible ways of satisfying given constraints but cannot be determined to be true by just following founded reasoning. Having both founded semantics and constraint semantics at the same time allows one to find such information.

Example. Predicate `unique` in `cmp_unit` below finds positions in the game in `win_unit1` that are U in the founded model but, if a constraint model exists, are winning in all possible models in constraint semantics .

```
kunit win_unit1:
  prolog ← ¬ asp
  asp ← ¬ prolog
  move(1,0) ← prolog
  move(1,0) ← asp
  move(1,1)
  use win_unit ()

kunit cmp_unit:
  use win_unit1 ()

  unique(x) ← win.U(x) ∧ ∃ m ∈ win_unit1.CS
                ∧ ∀ m ∈ win_unit1.CS | m.win(x)
```

In `win_unit1`, founded semantics gives `move.T(1,1)`, `move.U(1,0)`, `win.U(0)`, and `win.U(1)`. `win_unit1.CS = {{move(1,1), move(1,0), win(1)}}`, that is, `win(1)` is true, and `win(0)` is false. So `win.U(1)` and `win.U(0)` are imprecise, and `unique(1)` is true in `cmp_unit`. ■

Multiple uncertain worlds. Given multiple worlds with different models, different uncertainties can arise from different worlds, yielding multiple uncertain worlds. It is simple to represent this using predicates that are possibly 3-valued and that are parameterized by a 2-valued model.

Example. The game in `win_unit2` uses `win_unit` on a set of moves. The game in `win_set_unit` has its own moves, but the moves are valid if and only if they start from a position that is a winning position in a model in the constraint semantics of `win_unit2`.

```
kunit win_unit2:
  move = {(1,4),(4,1)}
  use win_unit ()

kunit win_set_unit:
  move = {(1,2),(2,3),(3,1),(4,4),(5,6)}
  valid_move(x,y,m) ← move(x,y), win_unit2.CS(m), m.win(x)

  use win_unit (move = valid_move(m), win = valid_win(m))

  win_some(x) ← valid_win(x,m)
  win_each(x) ← win_some(x) ∧ ∀ m ∈ win_unit2.CS | valid_win(x,m)
```

In `win_unit2`, there is a 2-edge cycle of moves, so `win_unit2.CS = {m1,m2}`, where `m1.win = {1}` and `m2.win = {4}`. In `win_set_unit`, each `m` in `win_unit2` leads to a separately defined predicate `valid_move` under argument `m`, which is then used to define a separate predicate `valid_win` under argument `m` by instantiating `win_unit` with `move` and `win` parameterized by additional argument `m`. ■

6 Related work and conclusion

Many logic languages and semantics have been proposed. Several overview articles [2, 20, 21, 11, 24] give a good sense of the complications and challenges when there is unrestricted negation. Notable different semantics include Clark completion [5] and similar additions, e.g., [19, 22, 15, 4, 12, 23], Fitting semantics or Kripke-Kleene semantics [10], supported model semantics [1], stratified semantics [1], WFS [25], and SMS [13]. Note that these semantics disagree, in contrast to different styles of semantics that agree [9].

There are also a variety of works on relating and unifying different semantics. These include Dung's study of relationships [8], partial stable models, also called stationary models [20], Loop fomulas [16], FO(ID) [7], and founded semantics and constraint semantics [18]. FO(ID) is more powerful than works prior to it, by supporting both first-order logic and inductive definitions while also being similar to SMS [3]. However, it does not support any 3-valued semantics.

Founded semantics and constraint semantics uniquely unify different semantics, by capturing their different assumptions using predicates declared to be certain, complete, and closed, or not.

However, founded semantics and constraint semantics by themselves do not provide a way for different semantics to be used for solving different parts of a problem or even the same part of the problem. DA logic supports these, and supports everything completely declaratively, in a unified language.

Specifically, DA logic allows different assumptions under different semantics to be specified easily as meta-constraints, and allows the results of different semantics to be built upon, including defining predicates using undefined values in a 3-valued model and using models in a set of 2-valued models, and parameterizing predicates by a set of 2-valued models. More fundamentally, DA logic allows different parts of a problem to be solved with different knowledge units, where every predicate is a parameter that can be instantiated with new predicates, including new predicates with additional arguments. These are not supported in prior languages.

Among many directions for future work, one particularly important and intriguing problem is to study precise complexity guarantees for inference and queries for DA logic.

Acknowledgments. This work was supported in part by NSF under grants CCF-1414078, CNS-1421893, and IIS-1447549.

References

1. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufman (1988)
2. Apt, K.R., Bol, R.N.: Logic programming and negation: A survey. Journal of Logic Programming 19, 9–71 (1994)
3. Bruynooghe, M., Denecker, M., Truszczynski, M.: First order logic with inductive definitions for model-based problem solving. AI Magazine 37(3), 69–80 (2016)
4. Chan, D.: Constructive negation based on the completed database. In: Proceedings of the 5th International Conference and Symposium on Logic Programming. pp. 111–125. MIT Press (1988)
5. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Databases, pp. 293–322. Plenum Press (1978)
6. Dasseville, I., Van der Hallen, M., Janssens, G., Denecker, M.: Semantics of templates in a compositional framework for building logics. Theory and Practice of Logic Programming 15(4-5), 681–695 (2015)
7. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Transactions on Computational Logic 9(2), 14 (2008)
8. Dung, P.M.: On the relations between stable and well-founded semantics of logic programs. Theoretical Computer Science 105(1), 7–25 (1992)
9. Ershov, Y.L., Goncharov, S.S., Sviridenko, D.I.: Semantic foundations of programming. In: International Conference on Fundamentals of Computation Theory. pp. 116–122. Springer (1987)

10. Fitting, M.: A Kripke-Kleene semantics for logic programs. Journal of Logic Programming 2(4), 295–312 (1985)
11. Fitting, M.: Fixpoint semantics for logic programming: A survey. Theoretical Computer Science 278(1), 25–51 (2002)
12. Foo, N.Y., Rao, A.S., Taylor, A., Walker, A.: Deduced relevant types and constructive negation. In: Proceedings of the 5th International Conference and Symposium on Logic Programming. pp. 126–139 (1988)
13. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the 5th International Conference and Symposium on Logic Programming. pp. 1070–1080. MIT Press (1988)
14. Hou, P., De Cat, B., Denecker, M.: FO(FD): Extending classical logic with rule-based fixpoint definitions. Theory and Practice of Logic Programming 10(4-6), 581–596 (2010)
15. Jaffar, J., Lassez, J.L., Maher, M.J.: Some issues and trends in the semantics of logic programming. In: Proceedings of the 3rd International Conference on Logic Programming. pp. 223–241. Springer (1986)
16. Lin, F., Zhao, Y.: Assat: Computing answer sets of a logic program by sat solvers. Artificial Intelligence 157(1-2), 115–137 (2004)
17. Liu, Y.A., Stoller, S.D.: Founded semantics and constraint semantics of logic rules. Computing Research Repository arXiv:1606.06269 [cs.LO] (June 2016 (Revised April 2017)), `http://arxiv.org/abs/1606.06269`
18. Liu, Y.A., Stoller, S.D.: Founded semantics and constraint semantics of logic rules. In: Proceedings of the International Symposium on Logical Foundations of Computer Science. pp. 221–241. Springer (2018)
19. Lloyd, J.W., Topor, R.W.: Making Prolog more expressive. Journal of Logic Programming 1(3), 225–240 (1984)
20. Przymusinski, T.C.: Well-founded and stationary models of logic programs. Annals of Mathematics and Artificial Intelligence 12(3), 141–187 (1994)
21. Ramakrishnan, R., Ullman, J.D.: A survey of deductive database systems. Journal of Logic Programming 23(2), 125–149 (1995)
22. Sato, T., Tamaki, H.: Transformational logic program synthesis. In: Proceedings of the International Conference on Fifth Generation Computer Systems. pp. 195–201 (1984)
23. Stuckey, P.J.: Constructive negation for constraint logic programming. In: Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science. pp. 328–339 (1991)
24. Truszczynski, M.: An introduction to the stable and well-founded semantics of logic programs. In: Kifer, M., Liu, Y.A. (eds.) Declarative Logic Programming: Theory, Systems, and Applications, pp. 121–177. ACM and Morgan & Claypool (2018)
25. Van Gelder, A., Ross, K., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650 (1991)