Knowledge of uncertain worlds: programming with logical constraints

YANHONG A. LIU, *Computer Science Department, Stony Brook University, Stony Brook, NY 11794, U.S.A.*
E-mail: liu@cs.stonybrook.edu

SCOTT D. STOLLER, *Computer Science Department, Stony Brook University, Stony Brook, NY 11794, U.S.A.*

Abstract

Programming with logic for sophisticated applications must deal with recursion and negation, which together 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 supports 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.

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 [16]. 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, e.g., [10, 13, 31], but they have different underlying assumptions that are conflicting and subtle , and each is suitable for only certain kinds of 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 [18, 19], 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.

Vol. 31, No. 1, © The Author(s) 2020. Published by Oxford University Press. All rights reserved.
For permissions, please e-mail: journals.permission@oup.com.
Advance Access Publication on 23 December 2020 doi:10.1093/logcom/exaa077

We present examples for different games that show the power and ease of programming with DA logic. We use games because interdependent winning and losing positions taken by competing players give rise to negation in recursion. We also discuss and describe support for restricted parameters and recursive uses of knowledge units.

The rest of the paper is organized as follows. Section 2 discusses the need of easier programming with logic when faced with negation in recursion. Section 3 describes the unified language, DA logic. Section 4 presents the formal definition of the semantics of DA logic, as well as its consistency, correctness and decidability. Section 5 develops additional examples for different games. Section 6 explains restricted parameters and recursive uses of knowledge units. Section 7 discusses related work and concludes.

This paper is a revised and extended version of Liu and Stoller [20]. The revisions include many expanded explanations to make the paper more self-contained and easier to read, as well as general improvements throughout. The extension is mainly the new Section 6 on support for restricted parameters and recursive uses of knowledge units.

2 Need of easier programming with logic

We discuss the challenges of programming with negation and recursion and the need of easier programming with logic. We explain the basic ideas of well-known previous language semantics as well as founded semantics and constraint semantics, and give an overview of the proposed solutions. We use a small well-known example, the win-not-win game, for illustration.

Win-not-win game. Given a set of moves for a game, 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.

$$\texttt{win(x)} \leftarrow \texttt{move(x,y)} \land \neg\, \texttt{win(y)}$$

This seems to be a reasonable rule, because, besides giving the conditions for x to be a winning position, it also suggests that, if there is no move from x, then x is a losing position and if x is neither a winning nor a losing position, then x is a draw position. This captures the rule for winning and losing for many games, including in chess for the King to not be captured, giving winning, losing and draw positions.

However, there could be problems. For example if there is a move(1,1) for some position 1, then the win rule would give win(1) ← ¬ 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.

$$\begin{aligned} &\texttt{win(x)} \leftarrow \exists\, \texttt{y} \mid \texttt{move(x,y)} \land \texttt{lose(y)} \\ &\texttt{lose(x)} \leftarrow \forall\, \texttt{y} \mid \neg\, \texttt{move(x,y)} \lor \texttt{win(y)} \\ &\texttt{draw(x)} \leftarrow \neg\, \texttt{win(x)} \land \neg\, \texttt{lose(x)} \end{aligned}$$

The first two rules are used [8, 14] to form inductive definitions [23], 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 based on the second rule.

With winning and losing positions defined, the draw positions are the remaining positions, which are those in cycles of moves that have no moves to losing positions.

These three rules spell out the intended meaning of winning, losing and draw as implied by the single win rule. However, clearly, these rules are much more cumbersome than the single win rule.

Well-founded semantics. Indeed, with well-founded semantics (WFS) [31], which computes a 3-valued model, the single win rule above gives `win(x)` being true, false or unknown for each `x`, corresponding exactly to `x` being a winning, losing, or draw position, respectively.

WFS is highly non-trivial—informally, it is defined by the least fixed point of a transformation that combines what is usually called the one-step derivability operator, T_p and the element-wise negation of the operator U_p for computing what is called the greated unfounded set, yielding a single 3-valued model [31]; it involves computing an alternating fixed point or an iterated fixed point.

However, `win(x)` being 3-valued in WFS 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 draw positions (that is, positions for which `win` is unknown) explicitly, say to find all reachable nodes following another kind of moves from draw positions. One might try to do this by adding the following 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 for which `win(x)` is true or false, and `draw(x)` is unknown for all draw positions.

Stable model semantics. Stable model semantics (SMS) [13] computes a set of 2-valued models, instead of 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.

SMS is also highly non-trivial—informally, it is defined by guessing a truth assignment, expanding each rule into all possible instances, computing what is called the reduct by deleting rules whose negated conditions cannot be satisfied and deleting negated conditions in remaining rules, and then computing a minimum model of the resulting rules, yielding one model in a set of 2-valued models [13]; in general, the number of guesses and resulting models can be exponential.

For the single win rule, if besides some winning and losing positions, there is a separate cycle of even length, say `move(1,2)` and `move(2,1)`, then the win rule would give `win(1) ← ¬ win(2)` and `win(2) ← ¬ win(1)`. Instead of `win` being unknown for positions `1` and `2` as in WFS, SMS returns two models: one with `win` being true for `1` and false for `2`, and one with `win` being true for `2` and false for `1`. This is a very different interpretation of the win rule.

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

Founded semantics and constraint semantics. Founded semantics and constraint semantics [18, 19] unify different prior semantics. They define a 3-valued model and a set of 2-valued models, respectively. They allow different underlying assumptions to be specified for each predicate. Specifically:

1. Each predicate can be declared *certain* (that is, everything about the predicate being true (T) are given or can be inferred by following the rules, and the rest are false (F)) or *uncertain* (that is, everything about the predicate being T or F are given or can be inferred, and the rest are undefined (U)), except a predicate must be uncertain if it depends on negation in recursion or on uncertain predicates.

2. Each uncertain predicate can be further declared *complete* (that is, all rules with the predicate in the conclusion are given, and thus before inferring T and F, completion rules can be added to define the negation of the predicate using the negation of the conditions of those given rules) or not, except a predicate must be not complete if it depends on predicates that are uncertain and not complete.
3. Each uncertain and complete predicate can be declared *closed* (that is, an assertion of the predicate is made F, called self-false, if inferring it to be T requires assuming itself to be T) or not. Being closed is needed to match WFS and SMS theoretically, but is not needed to give the desired meaning for any example we found in previous literature.

Founded semantics infers T, F and U using a simple least fixed point, with additionally computing self-false assertions for closed predicates, if any, in each iteration. Constraint semantics then extends everything U to be combinations of T and F that satisfy everything given as constraints.

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.

Explicit declaration in founded semantics and constraint semantics makes programming and understanding much easier. For example, in WFS and SMS, 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 could introduce some new `q` and two new rules, `p` $\leftarrow \neg$ `q` and `q` $\leftarrow \neg$ `p`, to make it possible that, in some models, `p` is true and `q` is false. Founded semantics and constraint semantics allow `p` to be explicitly declared uncertain and not complete.

Founded semantics and constraint semantics also allow unrestricted universal and existential quantifications and unrestricted nesting of Boolean operators; 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 game:

- The positions for which `win` is true, false and undefined in founded semantics are captured using three automatically derived predicates, `win.T`, `win.F` and `win.U`, respectively, corresponding exactly to the inductively defined `win`, `lose` and `draw`, respectively. These predicates can be used explicitly and directly for further reasoning, unlike with the truth values in WFS or founded semantics.

TABLE 1 Meta-constraints and corresponding prior semantics.

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,	P.`T`, P.`F`, P.`U`	
	not complete	$m.P$ for $m \in K$.`CS`	First-Order Logic
`complete`(P)	uncertain,	as above	Fitting (Kripke-Kleene)
	complete, not closed		Supported
`closed`(P)	uncertain,	as above	WFS
	complete, closed		SMS

- The constraint semantics of the given rule for `win` and facts for `move` is captured using an automatically derived predicate `CS`. For a model `m` in the constraint semantics, `CS(m)` is true, also denoted as `m` ∈ `CS`, and we use `m.win(x)` to denote the truth value of `win(x)` in model `m`. Predicate `CS` can be used directly for further reasoning, unlike the set of models in SMS or constraint semantics.

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 introduce knowledge units. DA logic supports instantiation and re-use of existing units, and allows predicates in any existing units 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.

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 [19]. Columns 1 and 3 are introduced in DA logic:

- Each meta-constraint in column 1 specifies the corresponding declarations in column 2. For example, `complete(`P`)` specifies that P is declared uncertain, complete, and not closed. Note that the four meta-constraints capture all possible combinations of declarations.
- In column 3, P`.T`, P`.F` and P`.U` are predicates that are true for a tuple of arguments if and only if P is T, F and U, respectively, for that tuple of arguments in founded semantics. K denotes a knowledge unit, and K`.CS` denotes the constraint semantics of K.

These will be described precisely in Sections 3 and 4.

3 DA logic

This section presents the syntax and informal meaning of DA logic, for design and analysis logic. The rule form described under "Conjunctive rules with unrestricted negation" is the same as the core language in our prior work on founded semantics and constraint semantics, for which we gave a precise semantics [18, 19]. Disjunction and quantification are mentioned as extensions in our prior work [18, 19]. 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 $\neg$:

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

Symbols $\leftarrow$, $\wedge$ and $\neg$ indicate backward implication, conjunction and negation, respectively. h is a natural number. Each P_i (respectively Q) is a predicate of finite number a_i (respectively a) of arguments. 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:

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

In quantifications of this form, the domain of each quantified variable X_k 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, `∃ x ∈ win | move(x,x)` is syntactic sugar for `∃ x | win(x) ∧ move(x,x)`, and `∀ x in win | move(x,x)` is syntactic sugar for `∀ x | ¬ win(x) ∨ move(x,x)`.

Meta-constraints. Assumptions about predicates are indicated in programs using the meta-constraints in column 1 of Table 1. Each meta-constraint specifies the declarations listed in column 2 of Table 1. For example, if a kunit contains `open(`P`)`, we say that P is declared uncertain and not complete in that kunit. In each kunit, exactly one meta-constraint must be given for each predicate.

Meta-constraint `certain(`P`)` means that each assertion of P has a unique true (T) or false (F) value. Meta-constraint `uncertain(`P`)` means that each assertion of P has a unique true, false, or undefined (U) value. Meta-constraint `complete(`P`)` means that all rules with P in the conclusion are given in the containing kunit. Meta-constraint `closed(`P`)` means that an assertion of P 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. If a predicate P is not defined transitively using its own negation and is not defined transitively using a predicate that is defined transitively using its own negation, then it is given the meta-constraint `certain(`P`)` by default. Otherwise, it is given `complete(`P`)` by default.

Using kunits with instantiation. The body of a kunit K_2 can use another kunit K using an instantiation of the form:

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

By definition, this has the effect of applying the following substitution to the body of K and inlining the result in the body of K_2: 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 $arity(Q_i) - arity(P_i)$. When P_i and Q_i have the same arity, we simply write $P_i = Q_i$ in the `use` construct.

The determination of default meta-constraints, and the check for having exactly one meta-constraint per predicate, are performed after expansion of all `use` constructs.

A kunit K_2 has a *use-dependency* on kunit K if K_2 uses K. The use-dependency relation must be acyclic. We have not found intrinsically good reasons for uses to be cyclic. However, there is no real difficulty in supporting circular uses. Section 6 discusses pros and cons of circular uses and extensions to support circular uses.

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 many logic languages, including our prior work on founded semantics [18, 19], a program is an unstructured set of rules and facts. The structure and re-use provided by kunits is vital for expressing knowledge modularly, building large conceptual models and developing large practical applications.

Referencing founded semantics. The founded semantics of a predicate P in a kunit K (formally defined in Section 4.2) can be referenced in K using special predicates $P.$`T`, $P.$`F` and $P.$`U`, one for each of the three truth values T, F and U. For each truth value t, $P.t(c_1, ..., c_a)$ is true if $P(c_1, ..., c_a)$ has truth value t, and is false otherwise.

Note that the founded semantics of K can be referenced in another kunit K_2 by simply adding `use` K, i.e., instantiating K in K_2 without replacing any predicate, when predicates of K and K_2 are disjoint. Otherwise, an instantiation with predicate replacements can be used to avoid name collisions. Our could also add a language feature for referencing the founded semantics of K using predicates of the form $K.P.$`T`, $K.P.$`F` and $K.P.$`U`, instead of using instantiation.

To ensure that the semantics of P is fully determined before these predicates are used, P cannot be defined transitively using these predicates. Predicates that reference founded semantics are implicitly given the meta-constraint `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)` $\leftarrow$ `win.U(x)`. If the kunit contains constants representing players as well as positions, `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 a conjunct `move(x,y)` $\vee$ `move(y,x)`, to select `x` that are positions in moves.

Referencing constraint semantics. The constraint semantics of a kunit K (formally defined in Section 4.2) can be referenced in another kunit K_2 using the special predicate $K.$`CS`. Using this special predicate in any rule in K_2 has the effect of adding each constraint model of K as an element in the domain (that is, set of constants) of K_2. In other words, the possible values of variables in K_2 include the constraint models of K. The assertion $K.$`CS`(c) is true when c is a constraint model of K and is false for all other constants.

Note that the constraint semantics of K cannot be referenced from within K; this ensures that the set of constraint models is fully defined before it is referenced. The constraint semantics of K cannot be referenced by instantiating K; this is why we need to introduce a language feature for referencing it using predicates of the form $K.$`CS`.

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`). This restriction is needed to prevent constraint models from containing contradictions such as the following: suppose $P.$`U`(0) is true in the founded model of a kunit K, and K has at least one constraint model m; then $P.$`U`(0) must also be true in m, but $P(0)$ must be true or false, not undefined, in m, because m is 2-valued. A kunit K_2 has a *CS-dependency* on another kunit K if K_2 uses $K.$`CS`. The CS-dependency relation must be acyclic.

When the value of a variable X is a constraint model of K, a predicate P of K can be referenced 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 for all arguments.

Predicates that reference constraint semantics are implicitly given the meta-constraint `certain` and can appear only in rule bodies.

4 Formal definition of semantics of DA logic

This section extends the definitions of founded semantics and constraint semantics in [18, 19] to handle the new features of DA logic.

Handling kunits is relatively straightforward. Because 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, 19] 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, 19]; they are included for completeness.

When we say that a predicate is certain, complete, or closed, we mean that it has that declaration in column 2 of Table 1 from its meta-constraint.

4.1 Preliminary definitions

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 union of the following sets: the set of constants in K, and for each kunit K_1 such that K_1`.CS` appears in K, the set of constraint models of K_1. Constraint models are formally defined in the last paragraph of Section 4.2. 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 $P.\texttt{F}(c_1, ..., c_a)$, 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 $\mathit{Proj}(K, S)$, 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 a predicate P 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.`CS`(c) is true in I if c is a model in the constraint semantics of K_1, 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.`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 (that is, for each ground instance of each rule, if the body is true, then so is the conclusion), when the rules are interpreted as formulas in 3-valued logic [10]. 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.

4.2 Founded semantics and constraint semantics of DA logic

This subsection first defines founded semantics of DA logic without meta-constraint `closed`, then extends the founded semantics to handle meta-constraint `closed`, and then defines the constraint semantics of DA logic.

Founded semantics of DA logic without meta-constraint `closed`. Intuitively, the *founded model* of a kunit K without meta-constraint `closed`, 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 define $Founded_0(K) = LFPbySCC(NameNeg(Cmpl(K)))$, 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 and complete predicate Q with a single *combined rule* for Q, defined as follows. (1) Transform the facts and rules defining Q so they all have the same conclusion $Q(V_1, ..., V_a)$, by replacing each fact or rule $Q(X_1, ..., X_a) \leftarrow$ B with $Q(V_1, ..., V_a) \leftarrow (\exists\, Y_1, ..., Y_k \mid V_1 = X_1 \wedge \cdots \wedge V_a = X_a \wedge B)$, where $V_1, ..., V_a$ are fresh variables (i.e., not occurring in the given rules defining Q), and

$Y_1, ..., Y_k$ are all variables occurring in $X_1, ..., X_a, B$, where B denotes the entire body of the rule. (2) Combine the resulting rules for Q into a single rule defining Q whose body is the disjunction of the bodies of those rules. This combined rule for Q is logically equivalent to the original facts and rules for Q. This definition is the same as given for the core language in [18, 19], except generalized to allow rule bodies that may contain disjunction. Similar completion rules are used in Clark completion [5] and Fitting semantics [10].

The function $\mathit{AddInv}(K)$ returns the kunit obtained from K by adding, for each uncertain and 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 $\mathit{NameNeg}(K)$ 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 $\mathit{LFPbySCC}(K)$ 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 $\mathit{LFPbySCC}(K) = I_n$, where $I_0 = \emptyset$ and $I_i = \mathit{AddNeg}(\mathit{LFP}(T_{I_{i-1} \cup \mathit{Proj}(K,S_i)}), S_i)$ for $i \in 1..n$. $\mathit{LFP}(f)$ is the least fixed point of function f. The least fixed point is well-defined, because $T_{I_{i-1} \cup \mathit{Proj}(K,S_i)}$ is monotonic, because the kunit K was transformed by *NameNeg* and hence does not contain negation. The function $\mathit{AddNeg}(I, S)$ returns the union of I and the set of completion facts for predicates in S that have meta-constraint `certain`; specifically, for each such predicate P, and for each combination of values $c_1, ..., c_a$ of arguments of P, if I does not contain $P(c_1, ..., c_a)$, then $P.\mathtt{F}(c_1, ..., c_a)$ is added as a completion fact.

Founded semantics of DA logic with meta-constraint `closed`. Informally, when a predicate of kunit K has meta-constraint `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.

To formally define the set of self-false atoms, we first transform the rules of K so that they do not contain disjunction, by putting the body of each rule R containing disjunction into disjunctive normal form (DNF) and then replacing R with multiple rules, one per disjunct of the DNF; this allows direct re-use of the following definitions of unfounded set and self-false atom from [18, 19], which do not take disjunction into account.

A set U of atoms of kunit K is an *unfounded set* of K with respect to an interpretation I of K iff, for each atom A in U, for each ground instance R of a rule of K with conclusion A, either (1) some hypothesis of R is false in I or (2) some positive hypothesis of R for a closed predicate is in U; this is the usual definition of unfounded set [31], except we inserted "for a closed predicate". $\mathit{SelfFalse}_K(I)$, the set of self-false atoms of kunit K with respect to interpretation I, is the greatest unfounded set of K with respect to I.

The founded semantics is defined by repeatedly computing the semantics given by $\mathit{Founded}_0$ (the founded semantics without meta-constraint `closed`) 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 $\mathit{Founded}(K) = \mathit{LFP}(F_K)$, where $F_K(I) = \mathit{Founded}_0(K \cup I) \cup \neg \cdot \mathit{SelfFalse}_K(\mathit{Founded}_0(K \cup I))$.

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 I of K such that I is a model of $\mathit{Cmpl}(K)$ and such that $\mathit{Founded}(K) \subseteq I$ and $\neg \cdot \mathit{SelfFalse}_K(I) \subseteq I$. Let $\mathit{Constraint}(K)$ denote the set of constraint models of K. Constraint models can be computed from $\mathit{Founded}(K)$ by iterating over all assignments of true and false to atoms that are undefined in $\mathit{Founded}(K)$, and checking which of the resulting interpretations satisfy all rules in $\mathit{Cmpl}(K)$ and satisfy $\neg \cdot \mathit{SelfFalse}_K(I) \subseteq I$.

4.3 Properties of DA logic semantics

The following theorems express important properties of the semantics.

THEOREM 4.1
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. For each kunit K, the proof of consistency is a straightforward extension of the proof of consistency of founded semantics [19,Theorem 1]. The extension is to show that consistency holds for the new predicates that reference founded semantics and constraint semantics.

For predicates in K 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. These truth values are consistent because, by the induction hypothesis, the interpretation computed for predicates in earlier SCCs is consistent.

For predicates in K that reference constraint semantics, they have the same truth values as the referenced predicates in the constraint models of other kunits, and constraint models are consistent by definition.

Next we consider constraint semantics. Again note that constraint models are consistent by definition. □

THEOREM 4.2
The founded model of a kunit K is a model of K and $\mathit{Cmpl}(K)$. The constraint models of K are 2-valued models of K and $\mathit{Cmpl}(K)$.

PROOF. The proof that $\mathit{Founded}(K)$ is a model of $\mathit{Cmpl}(K)$ is essentially the same as the proof that $\mathit{Founded}(\pi)$ is a model of $\mathit{Cmpl}(\pi)$ [19,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 $\mathit{Cmpl}(K)$ by definition. Any model of $\mathit{Cmpl}(K)$ is also a model of K, because K is logically equivalent to the subset of $\mathit{Cmpl}(K)$ obtained by removing the completion rules added by *AddInv*. □

THEOREM 4.3
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. □

Proving decidability of DA Logic is straightforward, but stating it explicitly is important, because DA logic supports recursion and allows nested constraint models to be used as constants.

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. For each example, we spell out the default meta-constraint for each predicate, in a topological-sort dependency order. We use `--` to prefix comments.

5.1 Same different games

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

EXAMPLE
Consider the following kunits. First, `path_unit` defines `path` recursively using `edge`: there is a path from `x` to `y` if there is a sequence of connected edges leading from `x` to `y`. Then, `win_path_unit` defines `link`, uses `path_unit` to infer `path` with `edge` bound to `link` and finally uses `win_unit` in Section 2 to determine winning, losing and draw positions except with `move` bound to `path`. With default meta-constraints, `edge` and `path` are certain, `link` is certain and `win` is complete.

```
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
```

Alternatively, in `win_path_unit`, one could define `edge` instead of `link`, and then use `path_unit` without replacing the name `edge` to `link`, as follows.

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

5.2 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
Consider the following `draw_unit`. It defines `move` and uses `win_unit`. Then, using the result of win-not-win game, predicate `move_to_draw` defines the set of positions that have a move to a draw position, and predicate `reach_from_draw` defines the set of positions that are reachable by following a path of special moves from a draw position. With default meta-constraints, `move` is certain, `win` is complete and `move_to_draw`, `special_move`, `path` and `reach_from_draw` are certain.

```
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)` to be true, and we have `reach_from_draw(4)` and `reach_from_draw(2)` to be true.

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.

5.3 Unique undefined positions

Among the most critical information is assertions that have a unique true or false value 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
Consider the following two kunits. First, `pa_unit` defines `prolog`, `asp` and `move` and uses `win_unit`. Then, `cmp_unit` uses `pa_unit` and defines `unique(x)` to be true if (1) `win(x)` is undefined in founded semantics, (2) a constraint model of `pa_unit` exists and (3) `win(x)` is true in all models in the constraint semantics. With default meta-constraints, predicates `prolog` and `asp` are complete, `move` is specified to be closed, `win` is complete and predicate `unique` is certain. Note that `prolog`, `asp` and `move` cannot be certain because `prolog` and `asp` are defined

with negation in recursion and `move` depends on `prolog` and `asp`.

```
kunit pa_unit:
      prolog ← ¬ asp
      asp ← ¬ prolog
      move(1,0) ← prolog
      move(1,0) ← asp
      closed(move)
      use win_unit ()
kunit cmp_unit:
      use pa_unit ()
      unique(x) ← win.U(x) ∧ ∃m ∈ pa_unit.CS ∧ ∀m ∈ pa_unit.CS | m.win(x)
```

In `pa_unit`, founded semantics gives `move.U(1,0)` (because `prolog` and `asp` are undefined), `win.F(0)` (because there is no move from `0`) and `win.U(1)` (because `win(1)` cannot be true or false).

Constraint semantics `pa_unit.CS` has two models: {`prolog, move(1,0), win(1)`} and {`asp, move(1,0), win(1)`}. We see that `win(1)` is true in all two models. So `win.U(1)` from founded semantics is imprecise.

In `cmp_unit`, `unique(1)` is true. That is, `win(1)` is undefined in founded semantics, a constraint model exists, and `win(1)` is true in all models in the constraint semantics.

5.4 Multiple uncertain worlds

Given multiple worlds each corresponding to a different model, 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
Consider the following two kunits. The game in `win_unit2` uses `win_unit` on a set of moves. The game in `win_set_unit` has its own moves, but a move is valid if and only if it starts from a position that is a winning position in a model in the constraint semantics of `win_unit2`. With default meta-constraints, `move` in both kunits are certain, `win` is complete, `valid_move` is certain, `valid_win` is complete and `win_some` and `win_each` are certain.

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

In `win_unit2`, there is a 2-move cycle. The constraint semantics `win_unit2.CS` is a set of two models, say {`m1,m2`}, where `m1.win` = {`1`} and `m2.win` = {`4`}. That is, in `m1`, position `1` is winning, and position `4` is not, and in `m2`, the situation is the opposite.

In `win_set_unit`, each model `m` in `win_unit2.CS` leads to a separately defined `valid_move` under argument `m`. In `m1`, only `move(1,2)` starts from the winning position `1`, and in `m2`, only `move(4,4)` starts from the winning position `4`. So `valid_move` is true for only `valid_move(1,2,m1)` and `valid_move(4,4,m2)`.

The separate `valid_move` under argument `m` is then used to define a separate `valid_win` under argument `m`, by instantiating `win_unit` with predicates `move` and `win` bound to `valid_move` and `valid_win`, respectively, with the additional argument `m`. This yields `valid_win` being true for only `valid_win(1,m1)`.

Finally, `win_some(x)` is true for any position `x` such that `valid_win(x,m)` is true for some model `m`, and `win_each(x)` is true if `win_some(x)` is true and `valid_win(x,m)` is true for all models `m` in `win_unit2.CS`. The result is that `win_some` is true for only `win_some(1)` and is false for all other positions, and `win_each` is false for all positions.

6 Restricted parameters and circular uses of kunits

Knowledge units are similar to modules in that they provide a means to organize the knowledge expressed as logic rules and constraints. We discuss extensions that allow knowledge units to have specially specified parameters, and have circular uses. We describe the pros and cons of supporting them and show that there is no real difficulty in supporting them.

6.1 Units with restricted parameters

Knowledge units as described in Section 3 do not need specially specified parameters. Any predicate in a kunit is in fact a parameter that can be instantiated with any predicate of the same number of arguments, or even with additional arguments if desired.

Some people may be accustomed to using modules or components with a specially specified set of parameters, where all uses of the module or component must instantiate exactly this restricted set of parameters. This is straightforward to add to DA logic, by simply specifying some of the predicates in a kunit as this restricted set of parameters of the kunit.

There are both pros and cons with specially specified parameters.

- The advantage is that one can hide the remaining predicates of the kunit from uses of the kunit. Changes to the hidden predicates will not affect uses of the kunit so long as the changes do not affect the specially specified parameters.
- The disadvantage is that if a hidden predicate becomes useful outside the kunit, the predicate must be added to the specially specified parameters to be used. Furthermore, this change is not limited to new uses of this kunit, but requires changes to all previous uses of the kunit.

Knowledge units with no restriction on parameters are more general and powerful for knowledge representation, for at least two reasons.

1. They can be used in any way that is easy and clear, with any combination of instantiated predicates that is needed, without changing the kunit or any previous uses of the kunit.
2. They encourage all predicates in a kunit to be carefully defined for clarity and reusability, eliminating the need to hide predicates that are not externally used.

Procedural programming benefits greatly from hiding internal details, because additional variables and parameters are most often used for efficiency reasons. In DA Logic, rules are declarative specifications, and hiding such specifications is generally unnecessary.

Nevertheless, to support hiding certain predicates, a kunit can specially specify which predicates can be externally used, as follows:

$$\texttt{kunit}\ K\ (\mathit{preds}) : \\ \mathit{body}$$

where *preds* is a set of predicates in K that can be instantiated or can be used outside K. The `use` clause

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

does not need any change. The semantics is extended to check that each predicate P_i is in the set *preds* of specially specified predicates of kunit K, and to ensure that there is no external use of predicates not in *preds*.

Note that this extension still allows each use of a kunit to instantiate any subset of the specially specified predicates of the kunit. This design is more general than parameterized module systems in which each module has a fixed set of parameters, all of which must be instantiated at every use.

6.2 Units with circular uses

Uses of knowledge units as described in Section 3 must form acyclic dependencies.

Some people may be accustomed to module systems that allow circular uses of modules. Allowing circular uses has both pros and cons.

- The advantage is that modules could be smaller and more flexible, and could use one another recursively.
- The disadvantage is that the dependencies between predicates in modules with circular uses may be difficult to determine and understand.

Knowledge units with no circular uses are easier to understand, for at least two reasons.

1. Dependencies between predicates defined in the knowledge units are clearer at a high level, because they must follow the tree of dependencies between kunits. With circular uses of kunits, all predicates defined in those kunits potentially depend on each other, depending on the details of their definitions.
2. Within a kunit, predicates easily capture any structure including cyclic graphs and the trivial case of recursive structures like trees, and recursive rules can easily define mutually dependent predicates.

Nevertheless, to support circular uses of kunits in DA logic, we can eliminate the requirement that the use-dependency relation is acyclic, and extend the semantics of `use` to handle circularity as follows. Recall from Section 3 that using a kunit has the effect of instantiating the body using the specified substitution and then inlining the result at the use. To support circular uses, the algorithm is extended to keep track of which uses of kunits have already been instantiated and inlined. The effect of using a kunit is to check whether the same use of the kunit has already been instantiated and inlined, and if so, do nothing, otherwise instantiate and inline it.

DA logic with this extension is still decidable, because there is only a finite number of possible uses of kunits in a program.

7 Related work and conclusion

Many logic languages and semantics have been proposed. Several overview articles [1, 11, 24, 25, 28] give a good sense of the complications and challenges when there is unrestricted negation in recursion. Notable different semantics include Clark completion [5] and similar additions, e.g., [4, 12, 15, 21, 26, 27], Fitting semantics or Kripke-Kleene semantics [10], supported model semantics [2], stratified semantics [2, 29], WFS [30, 31] 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 [7], partial stable models, also called stationary models [24], Loop formulas [22], FO(ID) [6] and founded semantics and constraint semantics [18, 19]. 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 atoms that have truth value undefined 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 optimal algorithms and precise complexity guarantees, similar to [17], for inference and queries for DA logic.

Acknowledgements

We thank the anonymous referees of this journal article and our LFCS 2020 paper [20] for their thoughtful reviews and detailed and helpful comments.

Funding

This work was supported in part by National Science Foundation under grants CCF-1414078, CCF-1954837, CNS-1421893 and IIS-1447549 and Office of Naval Research under grant N00014-20-1-2751.

References

[1] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, **19**, 9–71, 1994.

[2] K. R. Apt, H. A. Blair and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufman, 1988.

[3] M. Bruynooghe, M. Denecker and M. Truszczynski. First order logic with inductive definitions for model-based problem solving. *AI Magazine*, **37**, 69–80, 2016.

[4] D. Chan. 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] K. L. Clark. Negation as failure. In *Logic and Databases*, pp. 293–322, H. Gallaire and J. Minker, eds. Plenum Press, 1978.

[6] M. Denecker and E. Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic*, **9**, 14, 2008.

[7] P. M. Dung. On the relations between stable and well-founded semantics of logic programs. *Theoretical Computer Science*, **105**, 7–25, 1992.

[8] I. Dasseville, M. Van der Hallen, G. Janssens and M. Denecker. Semantics of templates in a compositional framework for building logics. *Theory and Practice of Logic Programming*, **15**, 681–695, 2015.

[9] Y. L. Ershov, S. S. Goncharov and D. I. Sviridenko. Semantic foundations of programming. In *Proceedings of the International Conference on Fundamentals of Computation Theory*, pp. 116–122. Springer, 1987.

[10] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, **2**, 295–312, 1985.

[11] M. Fitting. Fixpoint semantics for logic programming: A survey. *Theoretical Computer Science*, **278**, 25–51, 2002.

[12] N. Y. Foo, A. S. Rao, A. Taylor and A. Walker. Deduced relevant types and constructive negation. In *Proceedings of the 5th International Conference and Symposium on Logic Programming*, pp. 126–139. MIT Press, 1988.

[13] M. Gelfond and V. Lifschitz. 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] P. Hou, B. De Cat and M. Denecker. FO(FD): Extending classical logic with rule-based fixpoint definitions. *Theory and Practice of Logic Programming*, **10**, 581–596, 2010.

[15] J. Jaffar, J.-L. Lassez and M. J. Maher. 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] Y. A. Liu. Logic programming applications: What are the abstractions and implementations? In *Declarative Logic Programming: Theory, Systems, and Applications*, Chapter 10, pp. 519–557, M. Kifer and Y. A. Liu, eds. ACM and Morgan & Claypool, 2018.

[17] Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. *ACM Transactions on Programming Languages and Systems*, **31**, 1–38, 2009.

[18] Yanhong A. Liu and Scott D. Stoller. Founded semantics and constraint semantics of logic rules. In *Proceedings of the 2018 International Symposium on Logical Foundations of Computer Science*. Volume 10703 of Lecture Notes in Computer Science, pp. 221–241. Springer, Jan. 2018.

[19] Yanhong A. Liu and Scott D. Stoller. Founded semantics and constraint semantics of logic rules. *Journal of Logic and Computation*, **30**, 1609–1638, Dec. 2020. Preprint available at http://arxiv.org/abs/1606.06269.

[20] Yanhong A. Liu and Scott D. Stoller. Knowledge of uncertain worlds: Programming with logical constraints. In *Proceedings of the 2020 International Symposium on Logical Foundations*

of Computer Science. Volume 11972 of Lecture Notes in Computer Science, pp. 111–127. Springer, Jan. 2020.

[21] J. W. Lloyd and R. W. Topor. Making prolog more expressive. *Journal of Logic Programming*, **1**, 225–240, 1984.

[22] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, **157**, 115–137, 2004.

[23] Y. N. Moschovakis. *Elementary Induction on Abstract Structures*. North Holland, 1974.

[24] T. C. Przymusinski. Well-founded and stationary models of logic programs. *Annals of Mathematics and Artificial Intelligence*, **12**, 141–187, 1994.

[25] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *Journal of Logic Programming*, **23**, 125–149, 1995.

[26] T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 195–201. OHMSHA Ltd. Tokyo and North-Holland, 1984.

[27] P. J. Stuckey. Constructive negation for constraint logic programming. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pp. 328–339, 1991.

[28] M. Truszczynski. An introduction to the stable and well-founded semantics of logic programs. In *Declarative Logic Programming: Theory, Systems, and Applications*, pp. 121–177, M. Kifer and Y. A. Liu, eds. ACM and Morgan & Claypool, 2018.

[29] A. Van Gelder. Negation as failure using tight derivations for general logic programs. In *Proceedings of the 3rd IEEE-CS Symposium on Logic Programming*, pp. 127–138, 1986.

[30] A. Van Gelder, K. Ross and J. S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 221–230, 1988.

[31] A. Van Gelder, K. Ross and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, **38**, 620–650, 1991.

Received 1 June 2020