TPLP **26** (1-2): 147–162, 2026. © The Author(s), 2026. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

doi:10.1017/S1471068426100520 First published online 29 June 2026

*Parametric Modular Answer Set Programs Made Declarative**

JORGE FANDINNO and YULIYA LIERLER
Computer Science, University of Nebraska Omaha, USA
(*e-mails:* jfandinno@unomaha.edu, ylierler@unomaha.edu)

TORSTEN SCHAUB
Institut für Informatik und Computational Science, University of Potsdam, Germany
(*e-mail:* torsten@cs.uni-potsdam.de)

submitted 26 May 2026 UTC; revised 26 May 2026 UTC; accepted 28 May 2026 UTC

Abstract

In this paper, we explore the concept of modularity in first-order answer set programming (ASP). We introduce a new formalism called parametric modular logic programs, which allows defining subprograms with parameters and intensionality statements. We demonstrate how this formalism can capture the semantics of clingo-programs with *collective control*, a feature that enables structuring and instantiating subprograms. We provide theoretical foundations for modular ASP, illustrate its usefulness, and connect to traditional non-modular ASP.

KEYWORDS: answer set programming, modularity, formal methods

1 Introduction

Answer set programming (ASP) is a well-established declarative logic programming paradigm under the *answer set/stable models semantics* (Gelfond and Lifschitz 1988). The ASP methodology relies on devising a logic program so that its answer sets are in one-to-one correspondence to the solutions of the target problem. This approach is fully declarative, since the logic program only describes a problem and conditions on its solutions, not the way to obtain them. The latter is delegated to systems called answer set solvers. A software engineer is then tasked with translating problem's specifications into the formal language of logic programs rather than devising algorithms for obtaining solutions to a problem. Under these circumstances both the art of programming and the

* We would like to thank the anonymous reviewers for their valuable feedback that allowed us to improve the presentation of several points. This work was supported by the National Science Foundation CAREER award 2338635, USA, and the DFG grant SCHA 550/15, Germany. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

construction of an argument for code's correctness become more direct (Cabalar *et al.* 2020; Fandinno *et al.* 2020; Hansen and Lierler 2025; Fandinno *et al.* 2025). This is essential in building trustworthy software solutions, making ASP a viable role player in the current programming landscape. Yet, to realize the full potential of ASP more attention to its theoretical and practical aspects have to be paid. Here, we target advancing theoretical foundations of the "modularity" frontiers of this programming paradigm.

Modularity is one of the key techniques in principled software development, and it is essential for modeling large-scale practical applications. It provides a solid abstraction to study distinct approaches to solving a problem by identifying modules/parts of code that are responsible for different aspects and helps in constructing arguments of correctness by decomposing the argument about the whole program into arguments about its components (Cabalar *et al.* 2020). In general, traditional ASP programs lack modularity because subsections of the code cannot be evaluated in isolation. Most well known variants of ASP systems – *lparse-smodels* pair, earlier versions of *clingo*, *dlv* – expect a program at its full length bypassing any support for modularity. Yet, researchers realized early the importance of this aspect. Theoretical and practical advances exist. For instance, Oikarinen and Janhunen (2006, 2008) and Janhunen *et al.* (2007) devise a propositional modular framework for ASP and introduce concepts of equivalence for such programs; Harrison and Lierler (2016) introduce first-order modular logic programs. Lifschitz and Turner (1994) proposed *splitting* as a tool to "modularize" otherwise monolithic logic programs. Mentioned works were at the origin of more theoretical advances refining their proposals. On the practical side, Gebser *et al.* (2014) describe the *ASP+control* framework that, for example, allows encodings which include time horizons and assume repetitive code for each time step mimicking settings of planning domains. This framework was implemented within *clingo* 4. A PDDL planner PLASP backed up by ASP technology utilizes this feature (Dimopoulos *et al.* 2017). Multi-shot solving (Gebser *et al.* 2019) became one of the staples of *clingo* 5. It prominently positions an interactive mode of programming with ASP making control over ASP "modules" simple. Yet, control equates with a procedure that encroaches on the declarativness of ASP. In this paper, we study how "first-order modules" and "control" can be given a declarative meaning, by tackling on particular kind of control that we call *collective*. This work is a foundational step towards making the concepts pertaining (i) ASP modules as they are used in practice and (ii) control precise, well understood and *declarative*. This allow us to construct arguments of correctness for logic programs with collective control.

This paper is organized as follows. We start by introducing a motivating example. We then introduce theoretical preliminaries. Modular programs are introduced as the basis for what we call parametrized modular logic programs. We conclude by illustrating how parametrized modular programs capture the semantics of programs such as the one discussed as a motivating example. We use this example to showcase possible formal claims that can be seen as arguments of correctness for modular ASP programs.

2 Motivating examples

We start by introducing a simple example that illustrates the key concepts of parametric modular logic programs and collective control.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

```
#program base.                                  1
q(0,0).                                         2
#program property(k).                           3
q(N,k+1) :- q(N-1,k).                           4
```

Listing 1. The *property* parametric-program

```
#script(python)                                 5
from clingo.symbol import Number                6
def main(prg):                                  7
  n = prg.get_const("n").number                 8
  subs = [("base",[])]                          9
  for k in range(0,n):                          10
    subs.append(("property",[Number(k)]))       11
  prg.ground(subs)                              12
  prg.solve()                                   13
#end.                                           14
```

Listing 2. The implementation of control (A)

Listing 1 contains two subprograms `base` and `property(k)`. Keyword `base` marks a dedicated subprogram (with an empty parameter list). The `base` subprogram in Listing 1 includes the single fact `q(0,0)`. Without further control instructions, *clingo* grounds and solves the `base` subprogram only. This yields the standard behavior of answer set systems. In order to process the other subprograms such as `property(k)`, control instructions have to be explicitly stated. For instance, assume the *property* parametric-program in the context of the following *control*:

> *given a positive integer* n, *compute the answer sets of a program constructed from the* property *program by appending the content of its* `base` *program fact* $q(0,0)$ *to* n *copies of the rules in* `property(k)` *where parameter* k *ranges from* 0 *to* $n-1$. (A)

The python routine presented in Listing 2 below implements control (A).

If the contents of Listings 1 and 2 are stored in a file named `property.lp` then the command line `clingo property.lp -c n=100` (the presented code is compatible with *clingo* version 5.6) returns the unique answer set $\{q(0,0)\ \ q(1,1)\ q(2,2)\ \ \ldots\ \ q(100,100)\}$. Directive `-c n=100` provides a specific value 100 to be utilized in place of `n` by *clingo*. The **while**-loop of the routine in Listing 2 appends to the base program n pairs of the subprogram named *property* where the placeholder k is replaced by numbers 0 through $n-1$. In line 12, the instruction is given to ground all rules in `base` together with all rules in the `property` subprograms with their parameters k substituted by the corresponding integers. In line 13, the resulting ground program is solved. Thus, the *property* parametric-program together with the specified control is understood as the following "typical" logic program:

$$\begin{array}{lll} q(0,0). & q(N,0+1)\ \texttt{:-}\ q(N-1,0). & q(N,1+1)\ \texttt{:-}\ q(N-1,1). \\ \cdots & q(N,99+1)\ \texttt{:-}\ q(N-1,99). & \end{array} \tag{1}$$

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

Though extremely simple, the program in Listing 1 illustrates the key concepts of parametric modular logic programs and collective control that appear in planning problems.

A parametric-program together with its control can be seen as the means to concisely described what can be a lengthy program. Furthermore, it naturally groups rules of a program together. In the closing section, we illustrate how such grouping can be used in constructing arguments about formal properties of a program augmented with control based on the properties about its individual subcomponents. Let us summarize what we call a *collective control* exemplified by (A): this control

- collects the subprograms of interest with the values for the respective parameters;
- instantiates the respective parameters by their values in each of the subprograms;
- grounds the collection as if it were the union of rules;
- solves the ground program.

In what follows we provide the declarative semantics to *clingo*-programs with the collective control. The proposed semantics allow us to associate meaning with each individual component of a program with control without referring to concatenation and grounding.

3 Theoretical preliminaries

We open this section by describing syntax and semantics of normal logic programs with arithmetic (Fandinno *et al.* 2020; Lifschitz 2021; Fandinno *et al.* 2024). Then, we review the key concepts stemming from the multi-shot framework (Gebser *et al.* 2019).

3.1 Logic programs and their semantics

Many-sorted first-order formulas. We assume a many-sorted signature consisting of two sorts, *integer* and *general*, where the integer sort is a subsort of the general sort. This signature contains arithmetic functions such as $+$, $-$, and $\times$, whose argument and value sorts are integer, and comparison predicate constants such as $\leq$ or $>$, whose argument sorts are of sort general. We use infix notation for these arithmetic function and predicate constants. This signature also contains as object constants of sort integer all numerals $\overline{0}, \overline{1}, \overline{-1}, \overline{2}, \ldots$. We often identify numerals with integers and just write 0 instead of $\overline{0}$, but sometimes it is important to distinguish them. When it is important to distinguish a numeral that is an element of the signature from an underlying integer we will use an overline. This notation allows us to distinguish between expressions $\overline{5+3}$ and $\overline{5}+\overline{3}$. The former is identified with the numeral $\overline{8}$, whereas the later is identified with an algebraic expression that encodes function application to two numerals $\overline{5}$ and $\overline{3}$. In the remainder we use the overline only in cases exemplified by expression $\overline{5+3}$. We assume that for every sort, an infinite sequence of *object variables* of that sort is chosen. *Terms* over a signature σ are defined recursively as usual with the addition that arguments of functions have to be of appropriate sorts. *Atomic formulas* over σ are either (i) expressions of the form $p(t_1, \ldots, t_n)$, where p is a predicate constant and $t_1, \ldots, t_n$ are terms such that their sorts are subsorts of the argument sorts $s_1, \ldots, s_n$ of p, or (ii) expressions of the form $t_1 = t_2$, where t_1 and t_2 are terms. *(First-order) formulas* over signature σ are formed

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

from atomic formulas and the 0-place connective $\bot$ (falsity) using the binary connectives $\wedge$, $\vee$, $\rightarrow$ and the quantifiers $\forall$, $\exists$. The other connectives are treated as abbreviations: $\neg F$ stands for $F \rightarrow \bot$ and $F \leftrightarrow G$ stands for $(F \rightarrow G) \wedge (G \rightarrow F)$. A term or other expression is called *ground* if it contains no variables. A ground expression is called *precomputed* if it contains no arithmetic function constants. For instance, $p(f(1))$ is precomputed, whereas $p(1+2)$ is ground but not precomputed.

A *rule* is a formula of the form $Head \leftarrow Body$ (or, `Head :- Body` in source code format), where *Head* is an atomic formula or $\bot$, and *Body* is a list of *literals*, that is, atomic formulas possibly preceded by one or two occurrences of *not*. A *program* is a set of rules.

Every rule can be identified with some first-order formula over signature σ (Fandinno *et al.* 2020). Thus, a program can be seen as a first-order theory over σ. The translation from rules to formulas is not trivial due to the need to handle all possible terms occurring in the rule. Here, we use a simplified version of this translation, which is sufficient for the examples considered in this paper. Formal results apply to any translation that captures the intended meaning of rules as formulas. We identify each rule $Head \leftarrow Body$ with the universal closure of the formula $B \rightarrow Head$, where B is the conjunction of all literals in *Body* after replacing *not* by $\neg$. In the resulting formulas, every variable or constant occurring as an argument of an arithmetic function is assumed to be of the sort integer, similarly to the translation by Lifschitz (2021). We often write rules as formulas $B \rightarrow Head$ omitting the reference to the universal closure. For instance, we identify the rule listed in Line 4 of Listing 1 with the universal closure of formula $q(N-1,k) \rightarrow q(N,k+1)$, where N is a variable and k is an object constant, both of the integer sort.

Interpretations are defined as usual for many-sorted first-order languages (Lifschitz *et al.* 2008, Section 1.2.2). An *interpretation* I of a signature σ is *standard* if it satisfies the following conditions:

- its domain of the sort integer is the set of all numerals, and its domain of the sort general contains all precomputed terms,
- all numerals and object constants of the sort general are interpreted as themselves,
- arithmetic function are interpreted as customary in arithmetic, and
- predicate constants corresponding to comparison are interpreted according to some fixed total order such that numerals are contiguous and ordered as usual.

The value t^I assigned to a ground term t over σ by an interpretation I and the satisfaction relation (denoted, $\models$) between an interpretation of σ and a sentence over σ are defined recursively, in the usual way (Lifschitz *et al.* 2008, Section 1.2.2). An interpretation is called a *model* of a *theory* – a (possibly infinite) set of sentences – when it satisfies every sentence in this theory.

Pearce and Valverde (2004, 2005) introduced the quantified logic of here-and-there. One of its applications is an alternative definition of stable models for logic programs. A key benefit of this alternative is that it bypasses a reference to a grounding process when characterizing stable models. The many-sorted case of the quantified logic of here-and-there was recently studied by Fandinno *et al.* (2024). We review it here considering a signature σ as described above and standard interpretations. We start by introducing some notation. Let I to be a standard interpretation over signature σ. If $\mathbf{t}$ is a tuple $t_1, \ldots, t_n$ of ground terms, then $\mathbf{t}^I$ is the tuple $t_1^I, \ldots, t_n^I$ of values. By $\mathcal{A}_I$ we denote

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

the set of atoms of the form $p(\mathbf{t})$ such that $I \models p(\mathbf{t})$, where p is a predicate symbol and $\mathbf{t}$ is a tuple of precomputed terms. An *HT-interpretation* over σ is a pair $\langle \mathcal{H}, I \rangle$, where I is a standard interpretation and $\mathcal{H} \subseteq \mathcal{A}_I$. (In terms of Kripke models with two worlds, $\mathcal{H}$ describes the predicates in the here-world and I captures the there-world). The satisfaction relation $\models_{\text{ht}}$ between HT-interpretation $\langle \mathcal{H}, I \rangle$ of σ and a sentence F over σ^I is defined recursively:

- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} p(\mathbf{t})$, if $p(\mathbf{t}^I) \in \mathcal{H}$;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} t_1 = t_2$ if $t_1^I = t_2^I$;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F \wedge G$ if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F$ and $\langle \mathcal{H}, I \rangle \models_{\text{ht}} G$;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F \vee G$ if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F$ or $\langle \mathcal{H}, I \rangle \models_{\text{ht}} G$;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F \to G$ if (i) $\langle \mathcal{H}, I \rangle \not\models_{\text{ht}} F$ or $\langle \mathcal{H}, I \rangle \models_{\text{ht}} G$, and (ii) $I \models F \to G$;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} \forall X\, F(X)$ with X of the general sort, if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F(t)$ for every precomputed term t;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} \forall X\, F(X)$ with X of the integer sort, if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F(\overline{n})$ for every integer n;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} \exists X\, F(X)$ with X of the general sort, if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F(t)$ for some precomputed term t;
- $\langle \mathcal{H}, I \rangle \models_{\text{ht}} \exists X\, F(X)$ with X of the integer sort, if $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F(\overline{n})$ for some integer n.

If $\langle \mathcal{H}, I \rangle \models_{\text{ht}} F$ holds, we say that $\langle \mathcal{H}, I \rangle$ *satisfies* F and that $\langle \mathcal{H}, I \rangle$ is an *HT-model* of F.

It is easy to see that $(\mathcal{A}_I, I)$ is an HT-model of a sentence F whenever I is a model of F. About a model I of a theory Γ, we say it is *stable* if, for every proper subset $\mathcal{H}$ of $\mathcal{A}_I$, HT-interpretation $\langle \mathcal{H}, I \rangle$ does not satisfy Γ. In application to theories of a single sort (and thus in the absence of arithmetic function symbols), this definition is equivalent to the original definition by Pearce and Valverde (2004, 2005). In addition, if the theory is finite, then this definition of a stable model is also equivalent to the definition of such model in terms of the operator SM (Ferraris *et al.* 2007, 2011), when all predicate constants are considered to be "intensional". In the sequel, we sometimes abuse our terminology and notation and identify set $\mathcal{A}_I$ of ground atoms with interpretation I.

3.2 Parameterizable subprograms

We now review the key concepts introduced by Gebser *et al.* (2019) when discussing the multi-shot framework. We then use these concepts to formally characterize the collective control described in Section 2. A *program declaration* is an expression of the form `#program` $m(p_1, \ldots, p_j)$, where $m, p_1, \ldots, p_j$ are symbolic constants. We call m the name of the declaration and $p_1, \ldots p_j$ its parameters/placeholders. Different occurrences of program declarations with the same name are assumed to share the same parameters. In this way, each name is associated with a unique parameter specification. A *clingo-program* is a list of rules and declarations. Listings 1 exemplifies this concept. The *scope* of a program declaration in a *clingo*-program consists of the set of all rules following this declaration up to the next program declaration or the end of the list. In Listing 1, the scope of the declaration in Line 1 consists of atom `q(0,0)`, while that in Line 3 contains `q(N,k+1):-(N-1,k)`. Given *clingo*-program R along with a symbolic constant m, we define $R(m)$ as the set of all rules in the scope of all occurrences of program declarations

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

with name m. We often refer to $R(m)$ as a subprogram of R. All rules outside the scope of any (explicit) program declaration are thought of being implicitly preceded by `#program base`. Take R_1 to denote Listing 1, $R_1(base) = \{q(0,0)\}$ and $R_1(property) = \{q(N-1,k) \to q(N,k+1)\}$.

A *valuation* v of a set of placeholders PH is a function that maps elements in this set to precomputed terms. Given a program/formula F, a set PH of placeholders and valuation v on PH, by $v(F)$ we denote a program/formula constructed from F by substituting each of the occurrences of some placeholder $c \in PH$ with the value $v(c)$ assigned to it by the valuation v. Given a name m with associated parameters/placeholders $p_1, \ldots, p_j$, and valuation v on $\{p_1, \ldots, p_j\}$, we say that $v(R(m))$ is *the v-instantiation of subprogram* $R(m)$. For instance, if v is the valuation $\{k \mapsto 42\}$, then $v(R_1(property))$ consists of the universal closure of $q(N-1,42) \to q(N,42+1)$. We are now almost ready to formally characterize the collective control that provides specifications for instantiations of subprograms of interest and then considers these collectively as the union. Let us define one more concept. A *subprogram-spec* is a triple $[m, PH, v]$, where m is a name, PH is a set of associated parameters/placeholders with m, and v is a valuation on PH. Given (i) *clingo*-program R and (ii) set $[m_1, PH_1, v_1], \ldots, [m_j, PH_j, v_j]$ of *subprogram-specs*, the collective control associates this pair with the logic program composed of the rules

$$v_1(R(m_1)) \cup v_2(R(m_2)) \cup \cdots \cup v_j(R(m_j)). \tag{2}$$

For instance, command line `%clingo property -c n=100` (discussed in Section 2) assumes *clingo*-program in Listing 1 and the set composed of the following subprogram-specs: $[base, \emptyset, ()]$, $[property, \{k\}, (k \mapsto 0)]$, $\cdots$, $[property, \{k\}, (k \mapsto 99)]$. Although *clingo*-programs provide means to group rules into subprograms, the expression (2) clearly states that the object resulting from an application of the collective control to a *clingo*-program looses any aspect of the modularity.

As customary in logic programming, its programs define signatures implicitly. Namely, predicate and object symbols occurring in a program form its signature. If arithmetic operations are present the signature is infinite containing all numerals.

4 Modular logic programs

Our ultimate goal is to define parametrized modular programs that are capable to capture semantics of *clingo*-programs with collective control declaratively in a way that such modular programs treat individual subprograms with reverie so that each instantiation of a subprogram will have its individual meaning devoid of the rest of the context.

Prior to that we introduce modular logic programs that are close relatives to first-order modular logic program proposed by Harrison and Lierler (2016). In that work, first-order modular logic program was a collection of logic programs with some predicates identified as intensional and others as extensional. Thus, some members of that collection could be viewed as "definitions/modules" for concepts captured by intensional predicates. Here, we allow arithmetic and use a concept called intensionality statements introduced by Fandinno and Lierler (2023) to claim more control over granularity of modules.

This section is organized as follows. We start by introducing simple intensionality statements – a special case of such statements. We then define modules and modular

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

programs that use intensionality statements, intuitively, to pin the scope of module's applicability. The section culminates in defining parametric modular programs.

4.1 Simple intensionality statements

Fandinno and Lierler (2023) introduced intensionality statements to generalize the notion of intensional predicate constants. This concept allows some predicate constants to be intensional on some arguments while being extensional on others. This fine granularity came with the price that checking certain interesting properties is an undecidable problem. In particular, this is true for deciding whether the granular version of splitting theorem studied by Fandinno and Lierler (2023) is applicable. In the sequel, the splitting theorem by Fandinno and Lierler (2023) plays an important role. For these reasons, we find of value to focus on a special case of intensionality statements, which we call *simple*. Despite the fact that they are less general they are sufficient for our purposes and importantly more tractable. Before we proceed towards the presentation of the formal definitions let us take a moment to discuss the terms *intensional* and *extensional*. This terminology roots in deductive databases, where tuples in the database are seen as purely extensional in the sense that the whole extension of the predicate is known. Derived predicates are then purely intensional in the sense that only their definition is fixed. Here, we adjust this terminology to allow ourselves to be more detailed. We can point at a predicate and some of its arguments as the ones being "defined", while the same predicate on the remaining arguments is considered "known". This is important in our context. Indeed, as mentioned in the introduction, one of the distinguished uses of *ASP+control* is for modeling dynamic domains. These applications form a natural example where the same predicate is used both as intensional and extensional depending on its arguments. In a modular program describing a dynamic domain, a subprogram may naturally represent the effects of the actions at a given time step, where the same predicate is used to represent the state of the world at different time steps. In this case, the predicate is intensional for the arguments representing the current time step, and extensional for the arguments representing the previous time step.

A *simple intensionality statement* κ over a signature σ is a function mapping each predicate symbol p/n in σ to a set of n-tuples satisfying the following conditions: (i) each element of the tuple is either a variable or a precomputed term of σ, and (ii) no variable occurs twice in a tuple. In the following, we name the i-th element of a tuple assigned by a simple intensionality statement as X_i and variations when it is a variable. We say that a predicate symbol is (purely) *extensional* when its associated set is empty, and that it is (purely) *intensional* when it contains a tuple of variables (with no ground terms). When all predicate symbols are purely intensional or purely extensional, then the simple intensionality statement corresponds to the notion of intensional predicate constants (Ferraris *et al.* 2011). When the arity of p/n is clear from the context, we write κ^p instead of $\kappa(p/n)$. Consider, for instance, predicate symbol $q/2$ and let κ^q be set $\{\langle X, 1\rangle, \langle X, 2\rangle\}$. Intuitively, this intensionality statement states that the ground atoms formed by predicate constant q with its arguments of the form $\langle t, 1\rangle$ or $\langle t, 2\rangle$ are

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

intensional for any term t; otherwise, these atoms are extensional. For an intensionality statement κ and a predicate symbol p/n, by $\lambda^p_\kappa(\mathbf{X})$ we denote the formula

$$\bigvee_{(t_1,\ldots t_n)\in\kappa^p} \left(\bigwedge_{\substack{t_i \text{ is not a variable} \\ (1\leq i\leq n)}} X_i = t_i \right) \tag{3}$$

where $\mathbf{X} = \langle X_1, \ldots, X_n\rangle$ is a tuple of distinct variables of the appropriate length and sort (in the sequel we adopt this convention and use $\mathbf{X}$ to denote tuples of variables). To each simple intensionality statement κ, we associate a set $EM(\kappa)$ of sentences, called the *extensional axioms* of κ, containing a sentence of the form

$$\forall \mathbf{X}\, (\neg\lambda^p_\kappa(\mathbf{X}) \rightarrow p(\mathbf{X}) \vee \neg p(\mathbf{X})) \tag{4}$$

for every predicate symbol p/n in σ. For a theory Γ over σ, we say that a standard interpretation I is *κ-stable* if it is a stable model of $\Gamma \cup EM(\kappa)$. When it is clear from the context we sometimes drop κ as a subscript of λ^p_κ.

Let us illustrate these concepts on examples. Assume the signature σ_1 composed of a single predicate q. Consider $\kappa^q_1 = \{\langle X, 1\rangle, \langle X, 2\rangle\}$; then $\lambda^q_1(X_1, X_2)$ is $(X_2 = 1 \vee X_2 = 2)$ (here and in the sequel we understand λ_i as an abbreviation for λ_{κ_i}). Thus, extensional axioms formula $EM(\kappa_1)$ is equivalent to the the universal closure of

$$(X_2 \neq 1 \wedge X_2 \neq 2) \rightarrow (q(X_1, X_2) \vee \neg q(X_1, X_2)). \tag{5}$$

Consider theory Γ_1 composed of the universal closures of formulas $q(X, 0) \rightarrow q(X, 1)$, and $q(X, 1) \rightarrow q(X, 2)$. Sets $\{q(1, 3)\}$ and $\{q(0, 0), q(0, 1), q(0, 2)\}$ are among the κ_1-stable models of Γ_1. In case of the former, note that $q/2$ is not intensional for $\langle 1, 3\rangle$ and hence the extensional axiom (5) allows $q(1, 3)$ to be either true or false. In case of the latter, note that $q/2$ is not intensional for $\langle 0, 0\rangle$ either; hence, the extensional axiom allows $q(0, 0)$ to be either true or false. Atoms $q(0, 1)$ and $q(0, 2)$ must now be true. On the other hand, set $\{q(0, 1)\}$ is not a κ_1-stable model of Γ_1 because $q/2$ is intensional for $\langle 0, 1\rangle$ and there is no rule to derive it from.

There are some well known general properties about traditional logic programs and their stable models. For example, we can conclude that an atom is not a member of any answer set when it does not occur as a head of any rule. A similar result is also the case for κ-stable models. Proposition 1 below presents this result formally. We call a substitution θ *simple* with respect to signature σ whenever all values assigned by θ are precomputed terms over σ (we drop the reference to the signature when it is clear from the context).

Proposition 1.
Let I be a κ-stable model of Γ, p/n be predicate symbol, and $\mathbf{t}$ be a tuple of terms of length n such that $I \models \lambda^p_\kappa(\mathbf{t})$. It is the case that $I \not\models p(\mathbf{t})$ when there is no rule of the form $B \rightarrow p(\mathbf{r})$ in Γ and simple variable substitution θ such that $(\mathbf{r}\theta)^I = \mathbf{t}$ and $I \models B\theta$.

Intuitively, condition $I \models \lambda^p_\kappa(\mathbf{t})$ means that given a simple intensionality statement κ, theory Γ "defines" this predicate on the tuples associated with it.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

4.2 Modules and modular programs

A *module* over signature σ is a pair $\langle \kappa, \Pi \rangle$, where κ is a simple intensionality statement over σ and Π is a program over σ; we say that a κ-stable model of Π is a *model* of the module. For instance, consider module Δ_1 over σ_1 to stand for $\langle \kappa_1, \Gamma_1 \rangle$, where σ_1, κ_1 and Γ_1 are defined in Section 4.1. The models of Δ_1 are the κ_1-stable models of Γ_1.

A *modular program* $\mathcal{P}$ is a pair $\langle \kappa, \mathcal{M} \rangle$ where κ is a simple intensionality statement and $\mathcal{M} = \{\Delta_1, \ldots, \Delta_n\}$ is a finite set of modules over σ such that

$$\forall \mathbf{X} \, (\lambda_i^p(\mathbf{X}) \to \lambda^p(\mathbf{X})) \text{ is valid} \tag{6}$$

for every predicate symbol p/n in σ and every module Δ_i in $\mathcal{M}$; where λ^p and λ_i^p respectively are the formulas associated with p/n by κ and each κ_i. Intuitively, condition (6) states that every atom defined in a submodule must be intensional in the modular program. A standard interpretation I is a *stable model/answer set* of $\mathcal{P}$ if it is a model of each of its modules Δ_i and it satisfies formula

$$\forall \mathbf{X} (\lambda^p(\mathbf{X}) \wedge \neg \lambda_1^p(\mathbf{X}) \wedge \ldots \wedge \neg \lambda_n^p(\mathbf{X}) \to \neg p(\mathbf{X})) \tag{7}$$

for every predicate symbol p/n in σ. Intuitively, formula (7) states that every intensional atom that is not defined in any submodule must be false in all stable models.

Consider a sample modular program $\mathcal{P}_1$ over σ_1 whose simple intensionality statement is defined by $\kappa^q = \langle X, Y \rangle$ (in other words, predicate symbol q is purely intensional); and set of modules consists of Δ_0, Δ_1 and Δ_2, where

$$\Delta_0 = \langle \kappa_0, \{q(0,0)\} \rangle, \text{ and}$$
$$\Delta_2 = \langle \kappa_2, \{q(0,2) \to q(0,3), \; q(0,3) \to q(0,4)\} \rangle$$

with $\kappa_0^q = \{\langle 0, 0 \rangle\}$ and $\kappa_2^q = \{\langle X, 3 \rangle, \langle X, 4 \rangle\}$; and Δ_1 is understood as earlier. The only stable model of $\mathcal{P}_1$ is

$$\{q(0,0), \; q(0,1), \; q(0,2), \; q(0,3), \; q(0,4)\}. \tag{8}$$

We assume that an empty conjunction is equivalent to $\top$. The condition (7) for q is equivalent to the universal closure of $(X_1 \neq 0 \wedge X_2 = 0) \vee \neg(0 \leq X_2 \leq 4) \to \neg q(X_1, X_2)$. Hence, every atom of the form $q(t_1, t_2)$ is false when both $t_1 \neq 0$ and $t_2 = 0$ or when $t_2 \notin \{0, 1, 2, 3, 4\}$. All models of Δ_1 where $q(t_1, 0)$ is false must also satisfy that $q(t_1, 1)$ and $q(t_2, 2)$ are false, and all models of Δ_2 where $q(t_1, 2)$ is false must also satisfy that $q(t_1, 3)$ and $q(t_1, 4)$ are false. Since $q(t_1, 0)$ is false for every $t_1 \neq 0$, then atoms $q(t_1, t_2)$ with $t_1 \neq 0$ and $t_2 \in \{1, 2, 3, 4\}$ must be false as well. Hence, it remains to discuss atoms of the form $q(0, t_2)$ with $t_2 \in \{0, 1, 2, 3, 4\}$. Atom $q(0, 0)$ must be true because all the models of Δ_0 satisfy it. Atoms $q(0, 1)$ and $q(0, 2)$ must be true because they are true in the only models of Δ_1 satisfying $q(0, 0)$. Atoms $q(0, 3)$ and $q(0, 4)$ must be true because they are true in the only models of Δ_2 satisfying $q(0, 2)$. Note that, set (8) is exactly the unique κ-stable model of the program containing all rules in Δ_0, Δ_1 and Δ_2. This is not a coincidence, as we show in the next section.

Relation between modular programs and non-modular ones. For every term t, we define $[t]$ recursively as follows:

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

- if t is a variable or a precomputed term, then $[t] = t$;
- if t is $t_1 \odot t_2$ with $\odot \in \{+, -, \times\}$ and both t_1 and t_2 are ground terms, then $[t]$ is the numeral $\overline{n_1 \odot n_2}$ with $\overline{n_1} = [t_1]$ and $\overline{n_2} = [t_2]$;
- if t is $t_1 \odot t_2$ with $\odot \in \{+, -, \times\}$ and at least one of t_1 and t_2 is a variable, then $[t]$ is the expression $[t_1] \odot [t_2]$;

For example, $[0+1] = 1$, and $[X+1] = X+1$. If $\mathbf{t} = (t_1, \ldots, t_n)$ is a tuple of terms, then $[\mathbf{t}]$ denotes the tuple $([t_1], \ldots, [t_n])$.

A module $\langle \kappa, \Pi \rangle$ is called *simple* when for every atom $p(\mathbf{t})$ in the head of a rule in Π, there is a tuple $\mathbf{u}$ in κ^p and a substitution θ such that $[\mathbf{t}] = \mathbf{u}\theta$. A modular program is called *simple* if all its modules are simple. Modular program $\mathcal{P}_1$ defined above is simple. Indeed, we can check that module $\Delta_1 = \langle \kappa_1, \Gamma_1 \rangle$ is simple. We have two atoms in the heads of its rules, namely, $q(X, 1)$ and $q(X, 2)$. A substitution that maps X_1 to X will turn tuples in κ_1 into the arguments of these atoms. Similarly, we can check that modules Δ_0 and Δ_2 are also simple. As another example, let us consider module

$$\langle \{\langle X, 1\rangle\}, \{q(N-1, 0) \to q(N, 0+1)\}\rangle \tag{9}$$

which corresponds to the program *property*(k) in Listing 1 with $k = 0$. This module is also simple. Take the substitution that maps X to N and note that $[0+1]$ is numeral 1.

We next introduce the notion of *coherent modular program* and show that every coherent modular program has the same stable models as a non-modular program obtained by taking the union of all rules in its modules. To do so, we need to introduce the notion of a *dependency graph* of a modular program. Given a modular program $\mathcal{P} = \langle \kappa, \{\Delta_1, \ldots, \Delta_n\}\rangle$, the *(directed) graph of dependencies*, denoted $\mathfrak{G}(\mathcal{P})$, is defined as follows:

- Its vertices are pairs (p, i) with p a predicate symbol and $1 \leq i \leq n$.
- It has an edge from (p, i) to (q, j) when for some rule r of $\mathcal{P}$,
  - there is an atom of the form $p(\mathbf{t})$ in the head of r, and
  - there is a nonnegated atom of the form of $q(\mathbf{t}')$ in the body of r, and
  - there are tuples $\mathbf{u}_i$ in κ_i^p and $\mathbf{u}_j$ in κ_j^q, and substitutions θ and θ' such that $[\mathbf{t}] = \mathbf{u}_i\theta$ and $[\mathbf{t}'] = \mathbf{u}_j\theta'$.

We say that a strongly connected component of $\mathfrak{G}(\mathcal{P})$ is *contained in module* Δ_i if all vertices in this component are of the form (p, i).

As an example dependency graph $\mathfrak{G}(\mathcal{P}_1)$ contains three vertices $(q, 0)$, $(q, 1)$ and $(q, 2)$ and an edges from $(q, 2)$ to $(q, 1)$ and from $(q, 1)$ to $(q, 0)$. It is easy to see that the strongly connected component (q, i) is contained in module Δ_i for any $i \in \{0, 1, 2\}$.

A simple modular program $\mathcal{P}$ is called *coherent* if it satisfies the following two conditions:

- every strongly connected component of the dependency graph $\mathfrak{G}(\mathcal{P})$ is contained in some module; and
- every pair of modules $\langle \kappa_i, \Pi_i\rangle$ and $\langle \kappa_j, \Pi_j\rangle$ in $\mathcal{P}$ satisfy that, for every predicate symbol p, there is no pair of tuples $\mathbf{u}_i \in \kappa_i^p$ and $\mathbf{u}_j \in \kappa_j^p$ with $i \neq j$, such that $\mathbf{u}_i$ and $\mathbf{u}_j$ unify. Two tuples $\mathbf{t}_i = \langle t_1^i, \ldots t_m^i\rangle$ and $\mathbf{t}_j = \langle t_1^j, \ldots t_m^j\rangle$ of terms *unify* if there

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

is a substitution θ of the variables that applied to both $\mathbf{t}_i$ and $\mathbf{t}_j$ yields $t^i_k = t^j_k$, $1 \leq k \leq m$

Theorem 2.
The answer sets of a coherent modular program $\mathcal{P}$ are the same as the answer sets of the program obtained by taking the union of all rules in its modules.

Theorem 3.
Deciding if a modular program is coherent is feasible in polynomial time.

4.3 Parametric modular logic programs

As illustrated in Section 3.2, in practice, modules are often defined in a parametric way, that is, in terms of some parameters that can be instantiated in different ways to obtain different modules. To capture this idea, we introduce the notion of *parametric modules* that generalize modules introduced in Section 4.2.

Consider a set PH of object constants not occurring in σ, then by σ^{PH} we denote the signature σ extended with the object constants in PH. By PH^+ we denote the signature composed of the object constants in PH, numerals, and arithmetic function constants.

A *parametric intensionality statement* over σ^{PH} is a function mapping each predicate symbol p/n in σ to a set of n-tuples satisfying the conditions:

- each element of the tuple is either a variable, a precomputed term, or a ground term over signature PH^+;
- no variable occurs twice in a tuple.

A *parametric module* over σ is a triple $\langle PH, \chi, \Pi \rangle$, where χ is a parametric intensionality statement over σ^{PH} and Π is a formula (program understood as a set of formulas) over σ^{PH}. In the sequel, to illustrate relevant concepts we use the parametric module

$$\langle \{k\}, \chi_1, \{q(N-1,k) \to q(N,k+1)\} \rangle, \tag{10}$$

where we assume the signature composed of a single predicate q and parametric intensionality statement χ^q_1 is characterized by the singleton set consisting of $\langle X, k+1 \rangle$.

Given a parametric intensionality statement χ over σ^{PH} and substitution Θ on PH by $\chi\Theta$ we denote a function mapping each predicate symbol p/n in σ to the following set of n-tuples $\{\langle [t_1\Theta], \ldots, [t_n\Theta] \rangle \mid \langle t_1, \ldots, t_n \rangle \in \chi^p\}$. For a simple substitution Θ on PH, $\chi\Theta$ forms a simple intensionality statement. (We defined simple substitutions in Section 4.1.)

For example, let Θ_1 be a simple substitution that maps k to precomputed term 1. Then

$$\begin{array}{lll} (q(N-1,k) \to q(N,k+1))\Theta_1 & \text{is} & q(N-1,1) \to q(N,1+1); \\ \chi^q_1\Theta_1 & \text{is} & \{\langle X, 2 \rangle\}. \end{array}$$

Given a parametric module $\Phi = \langle PH, \chi, \Pi \rangle$ over signature σ and a simple substitution Θ on PH, we can construct a module $\Phi_\Theta = \langle \chi\Theta, \Pi\Theta \rangle$ (as introduced in Section 4.2). In the scope of this section it is convenient to call such modules *instances*. The *signature of any instance* of the parametric module is that of a parametric module, namely, σ.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

When the set of placeholders PH is empty in a considered parametric module, it serves a role of an instance itself.

For example, let D^k denote parametric module in (10). Take Θ_0 be a simple substitution that maps k to precomputed term 0. Then, instance $D^k_{\Theta_0}$ coincides with module (9). Similarly, if Θ_1 is a simple substitution that maps k to precomputed term 1, then instance $D^k_{\Theta_1}$ stands for $\langle\{\langle X, 2\rangle\}, \{q(N-1,1) \rightarrow q(N, 1+1)\}\rangle$.

We are ready to look into the example discussed in Section 2, namely, *property* parametric-program (Listing 1) that comes with collective control (A). We identify subprogram `property(k)` with parametric module D^k $(k \geq 0)$ introduced above. Substitutions Θ_i $(0 \leq i \leq n)$ map parameter k into precomputed term (integer) i. We identify subprogram *base* in the running example with a parametric module

$$\langle \emptyset, \{\langle 0, 0\rangle\}, \{q(0,0)\}\rangle \tag{11}$$

that we denote as D and which coincides with the module Δ_0 introduced in Section 4.2 (this module is an instance itself as its set of placeholders is empty). We view the *property* parametric-program together with the collective control as modular program

$$\langle \kappa, \{D, D^k_{\theta_0}, \ldots, D^k_{\theta_n}\}\rangle, \tag{12}$$

where the signature of this modular program contains predicate $q/2$ and the simple intentionality statement κ defines q as purely intensional. This modular program has a unique answer set, which, for example, in case when $n = 99$ coincides with the answer set of program (1). This last observation is not a coincidence and is supported by Theorem 2. In fact, that theorem gives us formal grounds for our earlier actions of identifying subprograms *base* and `property(k)` with parametric modules D and D^k $(k \geq 0)$, respectively.

5 Utility of declarative approach to modularity

We now illustrate how this declarative approach to modularity allows us to make formal claims about a program with control without a reference to underlying workings of *clingo*. Rather, these arguments naturally accompany intuitive readings of the parametric programs that we supply *clingo* with.

Consider the *property* program in Listing 1. Our first intuition is that its `base` subprogram states that

$$\text{pair } \langle 0, 0\rangle \text{ of integers has property } q. \tag{13}$$

In Section 4.3 we argued that the `base` subprogram can be identified with module D defined in (11). Any model of this instance supports claim (13) formally.

Our second intuition is that the `property(k)` subprogram states that given i is an integer substituted for k within this subprogram and property q holds for some pair $\langle j-1, i\rangle$, where j is an integer, then property q also holds for the pair $\langle j, i+1\rangle$. Take i to denote an arbitrary integer and Θ_i to denote a simple substitution that maps k into i. Recall that we view the `property(k)` subprogram as parametric module D^k; whereas $D^k_{\Theta_i}$ captures an instance of that program obtained by replacing integer i for parameter k. With that it is easy to see that any model of this instance supports the intuition we claim.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

Last but not least is grasping the overall meaning of the *property* program when control is in place and formally reasoning about its properties. First, the control specifies which exact instances of the subprograms are being part of an overall program being processed. Second, it implicitly defines the signature under which the program is interpreted. To continue with our running example, consider control (A). This control together with the considered parameterizable subprograms warrants the signature of a single predicate $q/2$ and infinite set of numerals. We now construct an inductive proof that the modular program listed in (12) (the modular program that we identify with the *property* program with the collective control) has a unique answer set I whose set $\mathcal{A}_I$ of true atoms has the form $\{q(0,0) \quad q(1,1) \quad q(2,2) \quad \ldots \quad q(n,n)\}$. The uniqueness of this answer set follows immediately from the following claims

Claim 1.
For any natural number i that is less or equal to n, $q(i,i)$ belongs to $\mathcal{A}_I$;

Claim 2.
For any natural number i that is greater than n, $q(i,i)$ does not belong to $\mathcal{A}_I$;

Claim 3.
For any distinct natural numbers i and j, $q(i,j)$ does not belong to $\mathcal{A}_I$.

We prove *Claim 1* by induction. Base case. Take i to be 0, $q(0,0)$ is part of $\mathcal{A}_I$ due to the observation we made as we discussed the first intuition. Our inductive hypothesis is that the claim in question is the case for some integer j that is less than n. We now show that it is the case for $j+1$. By inductive hypothesis $q(j,j)$ is part of $\mathcal{A}_I$. Then, $q(j+1,j+1)$ is part of $\mathcal{A}_I$ due to the presence of the instance $D^k_{\theta_j}$ in (12) and observation we made as we discussed the second intuition.

For *Claim 2*, condition (7) of the definition of stable models of modular programs for the case program (12) and predicate symbol $q/2$ has the form

$$\forall X_1 X_2(\top \wedge (X_1 \neq 0 \vee X_2 \neq 0) \wedge X_2 \neq 1 \wedge \cdots \wedge X_2 \neq n \rightarrow \neg q(X_1, X_2)). \tag{14}$$

This implies that whenever the second argument, let us call it m, of predicate q is greater than n, atom of the form $q(\cdot, m)$ is not part of $\mathcal{A}_A$. Claim 2 follows from this observation.

To show that *Claim 3* holds, it is sufficient to consider the case when j is less or equal to n. Indeed, recall the argument supporting *Claim 2*. Now we prove this claim by induction on j. *Base case*. Take $j=0$. And consider arbitrary integer i different from j. Trivially, $i \neq 0$. The consequence of (14) is satisfied by I. Then, $q(i,j)$ is not part of $\mathcal{A}_I$. *Inductive step*. Consider $0 \leq j < n$. The *inductive hypothesis* states: *for any natural number i that is different from j, $q(i,j)$ does not belong to $\mathcal{A}_I$*. We now show that *for any natural number i that is different from $j+1$, $q(i,\overline{j+1})$ does not belong to $\mathcal{A}_I$*.

Consider any natural number i different from $j+1$. Module $D^k_{\theta_j}$ that has the form $\langle\{\langle X, \overline{j+1}\rangle\}, \{q(N-1,j) \rightarrow q(N, j+1)\}\rangle$. Formula $\lambda^q(i, \overline{j+1})$ is $X_2 = \overline{j+1}$. Thus, $I \models \lambda^q(i, \overline{j+1})$. Take Θ to denote the substitution $N \mapsto i$. This is the only substitution which turns atom $q(N, j+1)$ that occurs in the head of the only rule in $D^k_{\theta_j}$ into ground atom $q(i, j+1)$. Ground atom $q(N-1,j)\Theta$ forms the body of that rule and has the form $q(i-1,j)$. Given that $i \neq j+1$, it follows $i-1 \neq j+1$. By inductive hypothesis $q(i,\overline{j+1})$ does not belong to $\mathcal{A}_I$. By Proposition 1, $q(i,\overline{j+1})$ is not part of $\mathcal{A}_I$.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

Note how all our formal claims about the shape and the uniqueness of the answer set of the *property* program with collective control are void from the reference to the inner workings of *clingo*. There is no reference to concatenation, grounding, and solving.

6 Conclusions

This paper champions a need for investigating a declarative approach to modularity in ASP. We take a step in this direction by introducing parametric modular programs and their declarative semantics. In the concluding section we illustrate the utility of this approach by reasoning about a sample ASP program with collective control in a formal way that is void from the reference to the inner workings of ASP systems. This is an important step towards refining our understanding about modular formalisms and providing ASP practitioners with formal tools to reason about the properties of modular programs frequently required in the design of complex systems. Yet, we only scratched the surface of this important topic. There are many directions for future research. For example, it is important to investigate how the proposed approach can be extended to capture other forms of controls. In practice, it is common that controls use so called external atoms to "deactivate" certain parts of the subprograms "on demand". We believe that our framework can be extended to capture this kind of control as well, and we plan to explore this direction. Another viable direction for future work is a development of a declarative meta-language for the instantiation of parametric modules. The specifications in that language would define which instantiations of parametric modules form the actual intended program.

Competing interests

The authors declare none.

References

Cabalar, P., Fandinno, J. and Lierler, Y. 2020. Modular answer set programming as a formal specification language. *Theory and Practice of Logic Programming* 20, 5, 767–782.

Dimopoulos, Y., Gebser, M., Lühne, P., Romero, J. and Schaub, T. 2017. plasp 3: Towards effective ASP planning. In *Proceedings of the Fourteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, M. Balduccini and T. Janhunen, Eds. Vol. 10377 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 286–300.

Fandinno, J. and Lierler, Y. 2023. Splitting answer set programs with respect to intensionality statements. In *Proceedings of the Thirty-seventh National Conference on Artificial Intelligence (AAAI'23)*, B. Williams, Y. Chen and J. Neville, Eds. AAAI Press, 6338–6345.

Fandinno, J., Lifschitz, V., Lühne, P. and Schaub, T. 2020. Verifying tight logic programs with anthem and vampire. *Theory and Practice of Logic Programming* 20, 5, 735–750.

Fandinno, J., Lifschitz, V. and Temple, N. 2024. Locally tight programs. *Theory and Practice of Logic Programming* 24, 5, 942–972.

Fandinno, J., Hansen, Z., Lierler, Y., Glinzer, C., Heuer, J.Schaub, T., Stolzmann, T. and Lifschitz, V. 2025. ANTHEM 2.0: Automated reasoning for answer set programming. *Theory and Practice of Logic Programming* 25, 4, 668–684.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press

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

Ferraris, P., Lee, J. and Lifschitz, V. 2011. Stable models and circumscription. *Artificial Intelligence* 175, 1, 236–263.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2014. Clingo = ASP + control: Extended report. Technical report. Universität Potsdam. URL: http://www.cs.uni-potsdam.de/wv/pdfformat/gekakasc14a.pdf.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82. https://doi.org/10.1017/S1471068418000054.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, MIT Press, 1070–1080. https://doi.org/10.1201/b10397-6.

Hansen, Z. and Lierler, Yu 2025. SM-based semantics for answer set programs containing conditional literals and arithmetic. In PADL, Vol. 15537 of Lecture Notes in Computer Science, Springer, 71–87

Harrison, A. and Lierler, Y. 2016. First-order modular logic programs and their conservative extensions. In *Theory and Practice of Logic programming, 32nd Int'l. Conference on Logic Programming (ICLP)* Special Issue, 2016, Cambridge University Press.

Janhunen, T., Oikarinen, E., Tompits, H. and Woltran, S. 2007. Modularity aspects of disjunctive stable models. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, C. Baral, G. Brewka and J. Schlipf, Eds. Vol. 4483 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 175–187.

Lifschitz, V. 2021. Transforming gringo rules into formulas in a natural way. In *Proceedings of the Seventeenth European Conference on Logics in Artificial Intelligence (JELIA'21)*, W. Faber, G. Friedrich, M. Gebser and M. Morak, Eds. Vol. 12678 of Lecture Notes in Computer Science, Springer-Verlag, 421–434.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of International Conference on Logic Programming (ICLP)*, P. Van Hentenryck, Ed. MIT Press, 23–37.

Lifschitz, V., Morgenstern, L. and Plaisted, D. 2008. Knowledge representation and classical logic. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz and B. Porter, Eds. Elsevier, 3–88. http://www.cs.utexas.edu/users/ai-lab?

Oikarinen, E. and Janhunen, T. 2006. Modular equivalence for normal logic programs. In *Proceedings of the 17th European Conference on Artificial Intelligence, ECAI 2006*, G. Brewka, S. Coradeschi, A. Perini and P. Traverso, Eds. IOS Press, Amsterdam, The Netherlands, 412–416.

Oikarinen, E. and Janhunen, T. 2008. Achieving compositionality of the stable model semantics for smodels programs. *Theory and Practice of Logic Programming* 8, 717–761.

Pearce, D. and Valverde, A. 2004. Towards a first order equilibrium logic for nonmonotonic reasoning. In *Proceedings of the Ninth European Conference on Logics in Artificial Intelligence (JELIA'04)*, J. Alferes and J. Leite, Eds. Vol. 3229 of Lecture Notes in Computer Science, Springer-Verlag, 147–160. https://doi.org/10.1007/978-3-540-30227-8_15.

Pearce, D. and Valverde, A. 2005. A first order nonmonotonic extension of constructive logic. *Studia Logica* 30, 2-3, 321–346.

https://doi.org/10.1017/S1471068426100520 Published online by Cambridge University Press