TPLP **26** (1-2): 163–179, 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/S1471068426100519 First published online 10 July 2026

`flingo` – *Instilling ASP Expressiveness into Linear Integer Constraints*[*]

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

JORGE FANDINNO
Computer Science, University of Nebraska Omaha, USA
(*e-mail:* jfandinno@unomaha.edu)

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

PHILIPP WANKO
Informatics, Universitat Potsdam, Germany
(*e-mail:* wanko@cs.uni-potsdam.de)

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

Abstract

Constraint answer set programming (CASP) is a hybrid paradigm that enriches answer set programming (ASP) with numerical constraint processing, a crucial requirement for many real-world applications. However, the specification of constraints in most CASP solvers aligns more closely with the expressiveness and semantics of the numerical back-end than the ASP paradigm. In the latter, numerical attributes are represented with predicates, and this allows for declaring default values, leaving the attribute undefined, making non-deterministic assignments with choice rules or using aggregated values. In CASP, most (if not all) of these features are lost once we switch to a constraint-based representation of those same attributes. In this paper, we present the `flingo` language (and tool) that incorporates the aforementioned expressiveness inside the numerical constraints, and we illustrate its use with several examples. Based on previous work that established its semantic foundations, we also present a translation from the newly introduced `flingo` syntax to regular CASP programs following the `clingcon` input format.

[*] We would like to thank anonymous reviewers for their valuable feedback that allowed us to improve presentation of several points. This work was supported by grant PID2023-148531NB-I00 funded by Spanish Ministry MCIU/AEI/10.13039/501100011033, funds FEDER, EU, by the National Science Foundation CAREER award 2338635, USA, and by 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/S1471068426100519 Published online by Cambridge University Press

KEYWORDS: answer set programming, constraint answer set programming, constraint processing

1 Introduction

One of the extensions of *answer set programming* (ASP; Lifschitz 1999) that is frequently demanded for practical applications is c*onstraint ASP* (CASP; Mellarkod *et al.* 2008; Lierler 2023). Many domains require efficient numerical computations that exceed the capabilities of state-of-the-art ASP tools for combinatorial problem solving. These tools typically divide their computation into a *grounding* phase (removing variables by instantiation) and a *solving* phase (getting the answer sets of a propositional program). In this schema, numerical constraints are solved by considering all the possible instantiations of the involved numerical variables during the grounding phase. This strategy has a serious problem of scalability: a simple change of measurement in some quantity (e.g., moving from meters to centimeters) causes a blow-up in the grounding of the program that immediately leads to intractability. The basic idea of CASP is delegating numerical computation to a back-end constraint solver, following the spirit of *Satisfiability Modulo Theory* (SMT; Nieuwenhuis *et al.* 2005). Thus, the ASP program is extended with *constraint atoms* that express the numerical restrictions associated with the problem at hand.

While this method mitigates grounding inefficiencies, it introduces a significant modeling challenge: the specification of constraint atoms often aligns more closely with the operational requirements of the numerical backend than with the standard ASP paradigm. Consequently, several high-level declarative features, such as default value declarations, undefined attributes, non-deterministic assignments via choice rules, and complex aggregates, are no longer directly accessible when these attributes are represented as constraint atoms. Just to put a simple example, consider the recent situation in which the default tariff for products from EU in the USA is fixed to 15%. This is a default that may have exceptions, such as the case of steel (with a tariff of 25%) or aircraft products, that are exempt. A possible representation[1] of this problem in `clingo` could look like:

```
tariff(P,eu,15) :- sales(P,eu,_), #false: tariff(P,eu,T), T!=15.
tariff(steel,eu,0).
tariff(aircraft,eu,25).
```

where predicate `sales(P,C,X)` reflects the total amount `X` of sales of products from country (or market) `C` of type `P`. Moreover, suppose we want to collect the total income from taxes as follows:

```
taxincome(Z) :- #sum{X*T/100,P,C:sales(P,C,X),tariff(P,C,T)}=Z.
```

If we decide to represent the tariff as a numerical constraint variable, default values are usually lost, since reasoning with constraint atoms is typically monotonic. Additionally, we cannot compute the tax income with an ASP aggregate, since the latter is defined in

[1] The `clingo` conditional literal "`#false:tariff(P,eu,T),T!=15`" can be replaced by a negated auxiliary predicate "`notother_eu_tariff(P)`" defined with rule "`other_eu_tariff(P):-tariff(P,eu,T),T!=15`".

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

terms of logical variables which, in general, cannot retrieve computations from numerical constraints (otherwise, the grounding gain is lost). Finally, the ASP code above can easily accommodate lack of information: for instance, our database may contain a fact like `sales(cars,us,30000)` without having to define any tariff for the USA itself, or may include information about the tariff for Canadian cars `tariff(cars,ca,25)` without knowing the tariff for the rest of products from that country. Declaring tariffs as constraint variables `tariff(steel,ca)`, `tariff(food,ca)`, etc forces us to explicitly assign them numerical values.

One approach in the literature that allowed undefinedness and default values for constraint variables was the logic of *Here-and-There with constraints* (HT_c) proposed by Cabalar *et al.* (2016). In that work, a first prototype `lc2casp` was presented as a proof of concept, but was still far from extrapolating standard ASP representational features to constraints. Later on, another of those common ASP features, aggregates, was also incorporated into constraint atoms (Cabalar *et al.* 2020a, b) under a semantics based on HT_c, although a proper language definition and the corresponding tool implementation was still missing.

In this paper, we introduce the `flingo` system, a comprehensive extension of `clingo` that enables the specification of undefined variables, default values, non-deterministic choices, and aggregates within constraint atoms. These features are handled uniformly with respect to the standard ASP paradigm, maintaining a consistent modeling experience. We formalize the semantics of the `flingo` language in terms of HT_c, illustrate its application through various examples, and provide a translation into regular CASP programs following the `clingcon` input format (Banbara *et al.* 2017).

The rest of the paper is organized as follows. The next section contains the background, including first an overview of HT_c, followed by a brief description of systems `clingo` and `clingcon`, on which `flingo` is based. Section 3 describes the `flingo` language, defining its input syntax, the associated semantics and some useful abbreviations. In Section 4, we provide details about the implementation of the `flingo` solver, which mainly relies on a translation into `clingcon`, used as a back-end. Finally, Section 5 summarizes our paper.

2 Background

2.1 The logic of here-and-there with constraints

The logic of *Here-and-there with constraints* (HT_c; Cabalar *et al.* 2016) is an extension of the logic of Here-and-There (HT; Heyting 1930). Analogously to Equilibrium Logic (Pearce 1997), a model selection is defined on HT_c models, which provides the logical foundations for constraint satisfaction problems (CSPs) in the setting of ASP. In HT_c, a CSP is expressed as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, also called *signature*, where $\mathcal{X}$ is a set of *variables* over some non-empty *domain* $\mathcal{D}$, and $\mathcal{C}$ is a set of *constraint atoms*. We assume a partition of the set of variables $\mathcal{X}$ into two disjoint sets: *propositional*, $\mathcal{X}^p$, and *non-propositional variables*, $\mathcal{X}^n$. Similarly, the set of constraint atoms $\mathcal{C}$ is also partitioned into two disjoint sets: *propositional*, $\mathcal{C}^p$, and *non-propositional atoms*, $\mathcal{C}^n$. The set of propositional atoms $\mathcal{C}^p$ contains a constraint atom $\mathtt{p}$ for each propositional variable $p \in \mathcal{X}^p$, that is, $\mathcal{C}^p = \{\mathtt{p} \mid p \in \mathcal{X}^p\}$. Note that we use different fonts to distinguish the propositional variable p from

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

the propositional atom $\mathtt{p}$. We also assume that the domain $\mathcal{D}$ contains the element $\mathbf{t}$, which is used to denote the truth value *true* of propositional variables. A *propositional formula* over $\mathcal{C}^p$ is defined by the grammar:

$$\varphi ::= \bot \mid \mathtt{p} \in \mathcal{C}^p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \tag{1}$$

Non-propositional atoms can be arbitrary strings, in which we distinguish substrings called *terms*. Terms are divided into *basic* and *conditional terms*. We assume a set of basic terms $\mathcal{T}^b$ that contains all the variables, domain elements, and the special symbol $\boldsymbol{u}$, representing an undefined value. A *conditional term* is an expression of the form

$$(s_1 | s_2 : \varphi) \tag{2}$$

where s_1 and s_2 are basic terms, and φ is a propositional formula. As an example,

$$\mathtt{sum}\{\ (3 * x | 1 : p)\ ;\ (2 | x : q)\ \} < 3 \tag{3}$$

is a non-propositional atom, where x is a variable, 1 and 2 are domain elements, and p and q are propositional variables. The variables, the domain elements and the string $3 * x$ are basic terms. $(3 * x | 1 : p)$ and $(2 | x : q)$ are conditional terms. Intuitively, the conditional term $(3 * x | 1 : p)$ means that if p is true, then the value of the term is $3 * x$, otherwise it is 1. Similarly, the conditional term $(2 | x : q)$ means that if q is true, then the value of the term is 2, otherwise its value is x. The value of the two conditional terms is then added up ($\mathtt{sum}$) and compared to 3.

A *formula* over $\mathcal{C}$ is defined by the grammar:

$$\varphi ::= \bot \mid c \in \mathcal{C} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi\,. \tag{4}$$

Hence, all propositional formulas are also formulas. As usual, we define $\top$ as $\bot \rightarrow \bot$, $\neg\varphi$ as $\varphi \rightarrow \bot$, and let $\varphi_1 \leftrightarrow \varphi_2$ stand for $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$. For every expression e, we let $vars(e)$ denote the set of non-propositional variables occurring in e.

For the semantics, we start by defining an extended domain as $\mathcal{D}_{\boldsymbol{u}} \stackrel{\text{def}}{=} \mathcal{D} \cup \{\boldsymbol{u}\}$. A *valuation* v over $\mathcal{X}, \mathcal{D}$ is a function $v : \mathcal{X} \rightarrow \mathcal{D}_{\boldsymbol{u}}$ where $v(x) = \boldsymbol{u}$ represents that variable x is left undefined. Moreover, if $X \subseteq \mathcal{X}$ is a set of variables, valuation $v|_X : X \rightarrow \mathcal{D}_{\boldsymbol{u}}$ stands for the projection of v on X. A valuation v can alternatively be seen as the set of pairs $\{(x \mapsto v(x)) \mid x \in \mathcal{X}, v(x) \in \mathcal{D}\}$ so that no pair in the set maps a variable to $\boldsymbol{u}$. This representation allows us to use standard set inclusion for comparison. We thus write $v \subseteq v'$ to mean that v and v' agree on all variables defined by v, but v' may define additional variables. By $\mathcal{V}$ we denote the set of valuations over $\mathcal{X}, \mathcal{D}$.

A constraint atom or formula is *basic* if it does not contain conditional terms. By $\mathcal{C}^b$ we denote the set of basic constraint atoms. The semantics of basic constraint atoms is defined by a denotation $[\![\cdot]\!] : \mathcal{C}^b \rightarrow 2^{\mathcal{V}}$ mapping each basic constraint atom c to the set of valuations satisfying c. For each propositional atom $\mathtt{p} \in \mathcal{C}^p$, we assume that its denotation satisfies $[\![\mathtt{p}]\!] \stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid v(p) = \mathbf{t}\}$. In HT_c, an *ht-interpretation* over $\mathcal{X}, \mathcal{D}$ is a pair $\langle h, t \rangle$ of valuations over $\mathcal{X}, \mathcal{D}$ such that $h \subseteq t$.

Definition 1.
Given a denotation $[\![\cdot]\!]$, an interpretation $\langle h, t \rangle$ *satisfies* a formula φ, written $\langle h, t \rangle \models \varphi$, if

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

1. $\langle h,t\rangle \models c$ if $h \in [\![\, eval_{\langle h,t\rangle}(c) \,]\!]$
2. $\langle h,t\rangle \models \varphi \wedge \psi$ if $\langle h,t\rangle \models \varphi$ and $\langle h,t\rangle \models \psi$
3. $\langle h,t\rangle \models \varphi \vee \psi$ if $\langle h,t\rangle \models \varphi$ or $\langle h,t\rangle \models \psi$
4. $\langle h,t\rangle \models \varphi \rightarrow \psi$ if $\langle w,t\rangle \not\models \varphi$ or $\langle w,t\rangle \models \psi$ for all $w \in \{h,t\}$

where $eval_{\langle h,t\rangle}(c)$ is the result of replacing each conditional term of the form $(s_1|s_2 : \varphi)$ in c with $eval_{\langle h,t\rangle}(s_1|s_2 : \varphi)$, which is defined as follows:

$$eval_{\langle h,t\rangle}(s_1|s_2 : \varphi) \stackrel{\text{def}}{=} \begin{cases} s_1 \text{ if } \langle h,t\rangle \models \varphi \\ s_2 \text{ if } \langle t,t\rangle \not\models \varphi \\ \boldsymbol{u} \text{ otherwise} \end{cases}$$

When $\langle h,t\rangle \models \varphi$ holds, we say that $\langle h,t\rangle$ is an *ht-model* of φ. We write $t \models \varphi$ if $\langle t,t\rangle \models \varphi$, and we say that t is a *model* of φ. A (ht-)model of a set Γ of formulas is a (ht-)model of all formulas in Γ. A model t of a set Γ is a *stable model* of Γ if there is no $h \subset t$ such that $\langle h,t\rangle \models \Gamma$.

2.2 ASP system `clingo` *and CASP system* `clingcon`

`Clingo` (Gebser *et al.* 2016) is an ASP system that provides a powerful interface for integrating external theories. It allows us to extend the basic ASP solver with new theory atoms by providing a theory grammar defining their syntax and an implementation of their semantics via so-called *theory propagators*. The CASP system `clingcon` (Banbara *et al.* 2017) uses this `clingo` interface to enrich the ASP language with linear constraint atoms over integers of the form

$$\&\texttt{sum}\{k_1 * x_1; \ldots; k_n * x_n\} \prec k_0 \tag{5}$$

where each k_i is an integer, each x_i is a (non-propositional) integer variable, and $\prec$ is one of the relations $<=, =, !=, <, >, >=$ Recently, it has been shown that the semantics of many of the systems built using `clingo`'s theory interface, including `clingcon`, can be nicely described in terms of HT_c by associating a denotation with each theory atom (Cabalar *et al.* 2025). `Clingcon` assumes a finite[2] domain of integers $\mathbb{Z}_{cl} \subseteq \mathbb{Z}$ with a minimum, min_{cl}, and a maximum value, max_{cl}. Consequently, the HT_c domain to capture `clingcon`'s semantics is $\mathcal{D}_{cl} = \mathbb{Z}_{cl} \cup \{\mathtt{t}\}$. The set of all valuations with domain $\mathcal{D}_{cl}$ is given by $\mathcal{V}_{cl}$. With it, the denotation of atom (5) is defined as follows:

$$\{v \in \mathcal{V}_{cl} \mid v(x_i) \in \mathbb{Z} \text{ for } 1 \leq i \leq n \text{ and } \textstyle\sum_{1 \leq i \leq n} k_i * v(x_i) \prec k_0\} \tag{6}$$

Let us now recall the syntax of `clingcon` programs (Banbara *et al.* 2017). A `clingcon` *atom* is either a propositional atom or a constraint atom of form (5). A `clingcon` *literal* is any `clingcon` atom a or its single or double negation, `not` a or `not not` a, respectively. A `clingcon` *rule* is an expression of the form

$$a :- l_1, \ldots, l_n \tag{7}$$

[2] The current implementation assumes the domain of integers between $\min_{cl}$ and $\max_{cl}$ that can be set using command line arguments `--min--int=<i>` and `--max-int=<i>`. The default values are $-2^{30}+1$ and $2^{30}-1$, which are also the minimum and maximum values that can be selected.

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

where a is a `clingcon` atom or the symbol $\bot$ denoting *falsity*, and each l_i is a `clingcon` literal. An expression of the form

$$\{\mathtt{p}\} :- l_1, \ldots, l_n \tag{8}$$

where `p` is a propositional atom, is called a *choice rule*, and it is an abbreviation for the rule '$\mathtt{p} :- l_1, \ldots, l_n$, `not not p`'. Moreover, `clingcon` accepts all the constructs of `clingo`, including disjunctive rules, aggregates, and weak constraints.

We define the semantics of `clingcon` programs via a translation τ into HT_c. For each `clingcon` literal l, we denote by τl the corresponding HT_c literal, that is, $\tau(\mathtt{not\ not}\ a) \stackrel{\text{def}}{=} \neg\neg a$, $\tau(\mathtt{not}\ a) = \neg a$, and $\tau a \stackrel{\text{def}}{=} a$ for any atom a. For each rule r of form (7), we denote by τr the formula

$$\tau l_1 \wedge \ldots \wedge \tau l_n \rightarrow \tau a \tag{9}$$

Furthermore, we need to introduce new constraints in HT_c, which are not available in `clingcon`. These constraint atoms are of the form $\&\mathtt{df}(x)$ and $\&\mathtt{int}(x)$, where x is a variable. These atoms are associated with the following denotations:

$$[\![\, \&\mathtt{df}(x) \,]\!] \stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid v(x) \neq \boldsymbol{u}\}$$
$$[\![\, \&\mathtt{int}(x) \,]\!] \stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid v(x) \in \mathbb{Z}_{cl}\}$$

For a set P of `clingcon` rules, we define $cl(P)$ as the set of HT_c formulas containing formula τr for each rule $r \in P$ as well as formulas of the form

$$\&\mathtt{df}(p) \rightarrow \mathtt{p} \qquad \text{for each propositional variable } p \in \mathcal{X}^p \tag{10}$$
$$\&\mathtt{int}(x) \qquad \text{for each integer variable } x \in \mathcal{X}^n \tag{11}$$

Formulas (10) and (11) ensure that variables only take their intended values: propositional variables are either undefined or take the value $\mathbf{t}$, while integer variables are always defined and can take any integer value. There is a one-to-one correspondence between the stable models of a `clingcon` program P and the stable models of the HT_c theory $cl(P)$ as shown in (Cabalar *et al.* 2025, Theorem 4).

3 The `flingo` language

`Flingo` is a CASP system that, much like `clingcon`, enriches the ASP language with integer constraint atoms. The main feature of `flingo` is that the values assigned to integer variables need to be *founded*. Recall that in ASP, propositional atoms are *false by default* and true propositional atoms need to be founded (Van Gelder *et al.* 1991), that is, derived from the facts via the rules of the program. Analogously, in `flingo`, integer variables are *undefined by default* and any defined integer variable needs to occur in a founded constraint atom. This behavior is inspired by HT_c and is significantly different from other existing CASP solvers. As an example, consider the program[3]

$$\{\mathtt{a}\}. \qquad\qquad \&\mathtt{sum}\{\mathtt{x}\} = 1 :- \mathtt{a}. \tag{12}$$

[3] Both `clingcon` and `flingo` (currently) rely on `clingo`'s theory grammar, which makes us express $x = 1$ as $\&\mathtt{sum}\{\mathtt{x}\} = 1$.

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

which has exactly two stable models in flingo: one where a is false and x is undefined, and another where a is true and x is assigned the value 1. Both of these are also clingcon stable models, but clingcon has infinitely many stable models where a is false and x is assigned any integer. The second distinct feature of flingo is its support of *aggregates* over integer variables, something that is not supported by other existing CASP solvers.

3.1 Syntax

As mentioned above, flingo supports an enriched syntax compared to clingcon. We start by describing flingo's *conditional terms*, which are inspired by aggregate elements in ASP and conditional terms in HT_c. A *product term* is either an integer, an integer variable, or an expression of the form $n * x$ where n is an integer and x is an integer variable. In flingo, a *conditional term* is an expression of the form

$$s : l_1, \ldots, l_n \tag{13}$$

where s is a product term and $l_1, \ldots, l_n$ is a comma separated list of propositional literals. There are two main differences with respect to conditional terms in HT_c. First, the condition is restricted to be a conjunction of propositional literals instead of an arbitrary propositional formula, as common in ASP. Second, conditional terms have no alternative term in flingo. The term s is the one considered when the condition $l_1, \ldots, l_n$ is satisfied, otherwise a default term is used. This default term is context-dependent and discussed below with the semantics of flingo. With these changes, we bring HT_c conditional terms closer to the usual logic programming syntax for aggregates.

A flingo *term* is either a product term or a flingo conditional term. The language of flingo supports the following constraint atoms:

$$\&\texttt{sum}\{t_1; \ldots; t_n\} \prec s \tag{14}$$

$$\&\texttt{sus}\{t_1; \ldots; t_n\} \prec s \tag{15}$$

$$\&\texttt{min}\{t_1; \ldots; t_n\} \prec s \tag{16}$$

$$\&\texttt{df}\{x\} \tag{17}$$

where each t_i is a flingo term, s is a product term, x is an integer variable, and $\prec$ is one of the following relations $<=, =, !=, <, >, >=$. The operation sus stands for *strict sum* (more precisely, sus stands for su*m* s*trict*), while sum is its non-strict version, similar to the one used in ASP. Intuitively, the non-strict version of sum discards undefined variables, while the strict does not. If any variable is undefined, the constraint atom with strict sum is false. As an example, consider the valuation $v = \{x_1 \mapsto 1\}$ and the constraint atoms

$$\&\texttt{sum}\{x_1;x_2\} <= 3 \tag{18}$$

$$\&\texttt{sus}\{x_1;x_2\} <= 3 \tag{19}$$

In this valuation, variable x_2 is undefined. Atom (18) is satisfied by valuation v because $v(x_1) = 1 \leq 3$, while x_2 is undefined and thus discarded from the sum. In contrast, atom (19) is not satisfied simply because x_2 is undefined. Operation min always work in a

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

non-strict way: for instance, `min` always returns the minimum value among the defined terms in the set.

A `flingo` *atom* is either a propositional atom or a constraint atom of the forms (14–17). A `flingo` *literal* is any `flingo` atom a or its single or double default negation, `not` a or `not not` a, respectively. A `flingo` *rule* is an expression of the form

$$a :- l_1, \ldots, l_n \tag{20}$$

where a is a `flingo` atom or the symbol $\bot$ denoting *falsity*, and each l_i is a `flingo` literal. Note that, syntactically, every `clingcon` rule is also a `flingo` rule.

3.2 Semantics

The semantics of `flingo` is defined using HT_c similar to `clingcon` but it uses a different denotation for constraint atoms, and a slightly different translation.

We start by defining a *neutral element* for each constraint atom operation

$$0^{\texttt{sum}} \stackrel{\text{def}}{=} 0 \qquad 0^{\texttt{sus}} \stackrel{\text{def}}{=} 0 \qquad 0^{\texttt{max}} \stackrel{\text{def}}{=} min_{cl} \qquad 0^{\texttt{min}} \stackrel{\text{def}}{=} max_{cl}$$

Both sum operations use 0 as their neutral element, while `max` and `min` use the minimum and maximum elements of the integer domain, respectively. We define the denotation of the basic constraint atoms as follows:

$$\begin{aligned}
[\![\, \&\texttt{sum}\{t_1; \ldots; t_n\} \prec s \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid v^{\texttt{sum}}(t_i) \in \mathbb{Z} \text{ for } 1 \leq i \leq n, \textstyle\sum_{1 \leq i \leq n} v^{\texttt{sum}}(t_i) \prec v(s)\} \\
[\![\, \&\texttt{sus}\{t_1; \ldots; t_n\} \prec s \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid v(t_i) \in \mathbb{Z} \text{ for } 1 \leq i \leq n, \textstyle\sum_{1 \leq i \leq n} v(t_i) \prec v(s)\} \\
[\![\, \&\texttt{min}\{t_1; \ldots; t_n\} \prec s \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid \min_{1 \leq i \leq n} v^{\texttt{min}}(t_i) \prec v(s)\} \\
[\![\, \&\texttt{df}(x) \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V}_{cl} \mid v(x) \neq \boldsymbol{u}\}
\end{aligned}$$

where

$$v(n) \stackrel{\text{def}}{=} n \qquad v(n * x) \stackrel{\text{def}}{=} \begin{cases} n * v(x) & \text{if } v(x) \in \mathbb{Z} \\ \boldsymbol{u} & \text{otherwise} \end{cases} \qquad v^{fun}(t) \stackrel{\text{def}}{=} \begin{cases} v(t) & \text{if } v(t) \in \mathbb{Z} \\ 0^{fun} & \text{otherwise} \end{cases}$$

The denotation of `&sus` requires all its terms to be defined, following the intuition of strict sum described above. In contrast, the denotation of `&sum` uses the neutral element $0^{\texttt{sum}} = 0$ to evaluate undefined terms. This has the same effect as discarding undefined terms from the sum, as done in ASP. We can now explain the difference between `&sus` and `&sum` in formal terms. Let us reconsider constraint atoms (18) and (19). On the one hand, we have $v^{\texttt{sum}}(x_1) = v(x_1) = 1$ but $v^{\texttt{sum}}(x_2) = 0$ because $v(x_2)$ is undefined. Hence, for atom (18), we have $1 + 0 \leq 3$, and thus $v \in [\![\, \texttt{sum}\{x_1; x_2\} <= 3 \,]\!]$. In contrast, for atom (19), we get $v(x_2) \notin \mathbb{Z}$ and, thus, we immediately have $v \notin [\![\, \texttt{sus}\{x_1; x_2\} <= 3 \,]\!]$. As mentioned above, `min` always behaves in a non-strict way, and accordingly, their denotations use their neutral elements to evaluate undefined terms. `&df`(x) expresses that the variable x is defined.

We now define the translation of `flingo` programs into sets of formulas in HT_c. Recall that `flingo` constraint atoms have conditional terms of the form $s : l_1, \ldots, l_n$, which do not have the syntactic form of HT_c conditional terms. Hence, the translation of `flingo` programs into HT_c needs to replace `flingo` conditional terms with HT_c conditional terms. This translation is context-sensitive, as it is determined by the specific constraint atom

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

containing the conditional term. Hence, for each flingo term t, we define a corresponding term $[t]^{fun}$ in HT_c where *fun* is one of the functions sum, sus, max, or min. This translation is defined as follows:

$$[t]^{fun} \stackrel{\text{def}}{=} t \qquad \text{if } t \text{ is a product term}$$
$$[t]^{fun} \stackrel{\text{def}}{=} (s \mid 0^{fun} \,:\, \tau l_1 \wedge \ldots \wedge \tau l_n) \qquad \text{if } t \text{ is of the form } s : l_1, \ldots, l_n$$

Whenever the condition is false, each term is evaluated to the corresponding neutral element of the function. Next, we define the translation μ of flingo atoms into HT_c atoms. For each atom c that is propositional or of the form $\&\mathtt{df}\{x\}$, its translation μc is just c itself. For any flingo constraint atom c of the form $\&fun\{t_1; \ldots; t_n\} \prec s$ with *fun* among {sum, sus, max, min}, we define μc as

$$fun\{\, [t_1]^{fun}; \ldots; [t_n]^{fun} \,\} \prec s\,. \tag{21}$$

The following proposition shows that sum and sus only differ when they contain undefined integer variables, that is, they are equivalent when all their integer variables are defined.

Proposition 1.
Let c_1 and c_2 stand for the flingo constraint atoms

$$\&\mathtt{sum}\{t_1; \ldots; t_n\} \prec s \qquad\qquad \&\mathtt{sus}\{t_1; \ldots; t_n\} \prec s$$

and let $\langle h, t\rangle$ be an ht-interpretation satisfying $\&\mathtt{int}(x)$ for each integer variable x in $vars(c_1)$ and $vars(c_2)$. Then, $\langle h, t\rangle \models \mu c_1$ if and only if $\langle h, t\rangle \models \mu c_2$.

We extend the translation μ to flingo literals in the usual way, that is, $\mu\,\mathtt{not}\; c$ is $\neg\mu c$, and $\mu\,\mathtt{not\ not}\; c$ is $\neg\neg\mu c$. Finally, we extend the translation μ to flingo rules (20) as

$$\mu l_1 \wedge \ldots \wedge \mu l_n \rightarrow \mu a \tag{22}$$

For a set P of flingo rules, we define $fl(P)$ as the set of HT_c formulas containing formula μr for each rule $r \in P$ as well as formulas of the form

$$\&\mathtt{df}(p) \rightarrow \mathtt{p} \qquad \text{for each propositional variable } p \in \mathcal{X}^p \tag{10}$$
$$\&\mathtt{df}(x) \rightarrow \&\mathtt{int}(x) \qquad \text{for each integer variable } x \in \mathcal{X}^n \tag{23}$$

Formulas (10) and (23) ensure that variables only take their intended values: propositional variables are either undefined or take the value t, while integer variables must take an integer value when defined. The formulas for propositional variables in (10) are the same as in clingcon, while the ones for integer variables in (23) are weaker than in (11), which ensures that integer variables are assigned integer values when defined, but crucially they remain undefined otherwise. Recall that in clingcon, the formulas in (11) ensure that integer variables are always defined.

Syntactically, every clingcon program P is a flingo program and, thus, we can also interpret it as a flingo program by using the corresponding translation $fl(P)$. Furthermore, every clingcon atom c has no conditional terms, and, thus, μc is just c itself. Hence, for a clingcon program P, there are two differences between $fl(P)$ and $cl(P)$: the denotation of &sum atoms and that the latter uses the formulas in (11) for integer variables, while the former uses the ones in (23).

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

While the denotation of &sum atoms in clingcon coincides with that of &sus atoms in flingo, we maintain a deliberate syntactic distinction. This choice is driven by our goal to align flingo as closely as possible with standard ASP aggregate conventions. In particular, since standard ASP aggregates adhere to non-strict semantics, flingo adopts a notation that reflects this established behavior. For instance, the flingo program

```
a :- &sum{ x : p } = 0.          p.
```

has a single stable model in which a and p are true and x is undefined. However, if we replace &sum with &sus, the resulting program leaves a undefined. This program can be rewritten in ASP using the aggregate #sum as follows:

```
a :- #sum{ X : val(X), p } = 0.     p.
```

The unique stable model of this program is $\{\texttt{a}, \texttt{p}\}$ (as all instances of val(X) are false).

To illustrate the use of strict versus non-strict sums for knowledge representation, we revisit next our running example about tariffs from the introduction. A possible representation of the example in flingo could be:

```
1  &sum{tariff(cars,ca)}=25.
2  &sum{tariff(P,eu)}=15 :- sales(P,eu,_),
3                           not &sus{tariff(P,eu)}!=15.
4  &sum{tariff(steel,eu)}=0.
5  &sum{tariff(aircraft,eu)}=25.
6  &sum{Y*tariff(P,C),P,C:sales(P,C,X),Y=X/100}=:taxincome.
```

Notice how the rule for expressing the default value (lines 2–3) uses &sus in the body to check that tariff(P, eu) does not have *some specific value* different from 15. If we used &sum instead, an undefined tariff would count as $0^{\texttt{sum}} = 0$ in the sum, and this is always different from 15, so the rule would not be applied, leaving the tariff undefined. Notice, however, that in the computation of the taxincome (line 6), it is more convenient to use &sum rather than &sus. This is because we might have facts like sales(food, ca, 10000) without having defined a tariff for Canadian food in the database. By using &sum, an undefined tariff in the aggregate just counts as 0 in the total sum (as would also happen in our standard ASP representation of the problem without constraints).

The following proposition shows the relation between the ht-models and stable models of $cl(P)$ and $fl(P^*)$ where P^* is the result of replacing each occurrence of &sum with &sus in a clingcon program P.

Proposition 2.
Let P be a clingcon program and P^* be the result of replacing each occurrence of &sum with &sus in P. Then,

(a) every ht-model of $cl(P)$ is also an ht-model of $fl(P^*)$; and
(b) every stable model of $fl(P^*)$ without undefined integer variables is also a stable model of $cl(P)$.

The relationship between flingo and clingcon described in Proposition 2(b) parallels the one between stable and classical models, with the caveat that we must restrict our

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

focus to stable models devoid of undefined integer variables. Just as every stable model is a classical model, though not vice versa, the presence of undefined variables in `flingo` allows for stable models that lack a counterpart in `clingcon`. To illustrate, consider the `clingcon` program $\{\texttt{a} :- \&\texttt{sum}\{x\} = x\}$ versus the corresponding `flingo` program $\{\texttt{a} :- \&\texttt{sus}\{x\} = x\}$. The former yields infinitely many stable models where `a` is true and x is assigned an arbitrary integer. In contrast, the `flingo` program results in a unique stable model where `a` is false and x remains undefined.

Continuing with the analogy between stable and classical models, in the latter, we can capture all models of a propositional theory by adding choice rules for each propositional variable. Similarly, we can capture all stable models of a `clingcon` program by adding facts of the form $\&\texttt{sum}\{x\} = x$ (aka $x = x$) for each integer variable x. These facts act as a kind of choice rule for integer variables, allowing them to pick any value.

Proposition 3.
Let P be a `clingcon` program and let F be the set of facts consisting of $\&\texttt{sum}\{x\} = x$ for each integer variable $x \in \mathcal{X}^n$.

Then, the set of ht- and stable models of $cl(P)$, $fl(P^* \cup F)$, and $fl(P \cup F)$ coincide.

We can further push this analogy by showing that `flingo` allows a form of non-monotonic reasoning for integer variables that is impossible in `clingcon`. Note that `clingcon` is not monotonic in the sense of propositional logic because it is an extension of the stable model semantics. Thus, we can just write a program containing propositional rules that is non-monotonic. However, we can show its non-monotonicity comes only from the propositional part of the program, while the integer part is monotonic.

Proposition 4.
Let P and F be two `clingcon` programs such that F has no propositional variables. Then, every stable model of $cl(P \cup F)$ is also a stable model of $cl(P)$.

The language of `flingo` is not monotonic in the sense of Proposition 4. For instance, consider the program consisting of the single rule $\texttt{x} = 1 :- \&\texttt{sus}\{\texttt{y}\} = 1$, whose unique stable model leaves both x and y undefined. If we add the fact $\&\texttt{sus}\{\texttt{y}\} = 1$, which does not contain propositional variables, then the resulting program's unique stable model assigns 1 to both x and y.

In addition to the basic constraint atoms defined above, `flingo` also supports `&max` constraint atoms of the form $\&\texttt{max}\{t_1; \ldots; t_n\} \prec s$, which are syntactic sugar for $\&\texttt{min}\{-t_1; \ldots; -t_n\} \succ -s$ where $\succ$ is the dual operator of $\prec$ defined as follows:

$$\succ \text{ is } \begin{cases} >= & \text{if } \prec \text{ is } <= \\ <= & \text{if } \prec \text{ is } >= \\ > & \text{if } \prec \text{ is } < \\ < & \text{if } \prec \text{ is } > \\ \prec & \text{otherwise} \end{cases}$$

A common pattern in `flingo` programs is to allow integer variables to take values within a given range. This can be expressed using *choice rules* for integer variables of the form

$$\&\texttt{in}\{s_1..s_2\} =: x :- \ l_1, \ldots, l_m \tag{24}$$

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

where s_1 and s_2 are product terms, x is an integer variable and $l_1, \ldots, l_m$ are `flingo` literals. Formally, a choice rule as in (24) is an abbreviation for the rules

$$\&\texttt{sus}\{s_1\} <= x :- \ l_1, \ldots, l_m, \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k) \tag{25}$$

$$\&\texttt{sus}\{s_2\} >= x :- \ l_1, \ldots, l_m, \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k) \tag{26}$$

where $x_1, \ldots, x_k$ are all the integer variables occurring in s_1 and s_2.

Besides this syntactic sugar, `flingo` also supports another common pattern for defining and constraining integer variables that work on the rule level. This pattern consists on defining a variable in terms of other variables. Constraint atoms (14–17) do not make any difference between the variables occurring on them. Assignment rules allow us to naturally express this pattern. An *assignment rule* is an expression of the form[4]

$$\&fun\{t_1; \ldots; t_n\} =: s \ :- \ l_1, \ldots, l_m \tag{27}$$

where each t_i is a `flingo` term, s is a product term, *fun* is one of the operations `sum`, `sus`, `max`, or `min`, and each l_i is a `flingo` literal. Such rules only define the variables in s, so that the constraint atom obtained by replacing the assignment operator =: with equality = is satisfied. However, it does not define any variables in $t_1, \ldots, t_n$. For instance, the assignment rule &`sus`{`x`; `y`}=: `z` ensures that z is defined as the sum of x and y, whenever both x and y are defined. However, if x or y are undefined, then the rule has no effect on the value assigned to z. So, if we add facts `x = 1` and `y = 2` to the above assignment, then the resulting program has a single stable model where x, y, and z are assigned 1, 2, and 3, respectively. If instead, we add only the fact `x = 1`, then the resulting program has a single stable model where x is assigned 1, while y and z are undefined. And if we add instead the facts `x = 1` and `z = 5`, then the resulting program has a single stable model where x and z are assigned 1 and 5, respectively, while y is undefined.

Formally, an assignment rule as in (27) where each t_i is a product term is an abbreviation for the rule:

$$\&fun\{t_1; \ldots; t_n\} = s \ :- \ l_1, \ \ldots, l_m, \ \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k) \tag{28}$$

where $x_1, \ldots, x_k$ are all the integer variables occurring in $t_1, \ldots, t_n$. When some t_i is a conditional term, conditions are first removed as explained in the next section.

4 The `flingo` system

We have implemented a new system called `flingo` as an extension of `clingo` (Gebser *et al.* 2016) with the integer constraint atoms as described in Section 3. The syntax uses `clingo` theory atoms starting with '`&`' to denote integer constraint atoms, and grounding of logic program variables is handled using `clingo`'s native capabilities. Logic program variables are different from integer variables. Such variables start with a lowercase letter and can only occur inside integer constraint atoms. As usual in ASP, logic program variables start with an uppercase letter and can be used anywhere in the program.

[4] Note that the assigned variables are the ones occurring on the right-hand side of the assignment operator. This is due to the current syntax of `clingo` theory atoms, where operators can only appear on the right-hand side of a constraint atom.

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

Logic program variables are grounded before the solving process starts, while integer constraint variables are handled during solving. `Flingo` uses `clingcon` as backend for handling integer constraints during solving. A grounded `flingo` program is translated into a `clingcon` program by a series of rewriting transformations detailed below. These transformations are based on ideas to compile away aggregates with constraints (Cabalar *et al.* 2020a) and to compile integer constraints allowing undefined integer variables into standard integer constraints (Cabalar *et al.* 2016), but adapted to the richer setting of `flingo`.

Our transformation consists of eight steps. The *first step* is to replace in the scope of a constraint atom whose operation is not &`sus`, each conditional term of the form $s : l_1, \ldots, l_m$ by

$$s : l_1, \ldots, l_m,\ \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k)$$

where $x_1, \ldots, x_k$ are all the integer variables occurring in s. Similarly, a product term s is replaced by the conditional term $s : \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k)$. After that, all occurrences of &`sum` atoms are replaced by &`sus` atoms.

The *second step* consists of replacing all conditional terms of the form (13) by conditional terms of the form $s : a$ where a is a propositional atom. This is done by introducing fresh propositional atoms a to represent the condition and adding rule a `:-` $l_1, \ldots, l_m$ to define this fresh atom.

The *third step* consists of removing conditions in constraint atoms. We replace each conditional constraint term of the form $s : a$ by a fresh integer variable y, and we add rules

$$\begin{aligned}
\&\texttt{sus}\{s\} &= y :- a,\ \&\texttt{df}(x_1), \ldots, \&\texttt{df}(x_k)\\
\&\texttt{sus}\{s\} &= y :- a,\ \&\texttt{df}(y)\\
\&\texttt{sus}\{0^{fun}\} &= y :- \texttt{not}\ a\\
\{a\} &:- \&\texttt{df}(y)
\end{aligned}$$

where $x_1, \ldots, x_k$ are all integer variables in s, and 0^{fun} depends on the operation *fun* where the conditional constraint term occurs as described in Section 3.

The *fourth step* is to rewrite all abbreviations described in the previous section: assignment and choice rules, and all occurrences of `max` constraint atoms.

The *fifth step* is to replace each constraint atom of the form $\&fun\{s_1; \ldots; s_n\} \prec s$ by a fresh propositional atom $\&fun(hb)\{s_1; \ldots; s_n\} \prec s$ where hb is either `head` or `body` depending on whether the atom occurs in the head or the body, respectively. `Clingcon` only recognizes constraint atoms whose operation is &`sum`, thus these atoms with operation $\&fun(hb)$ are treated as regular propositional atoms by `clingcon`. We add later rules that relate the value of the fresh propositional atoms and the integer variables.

The *sixth step* is to define &`min` atoms in terms of &`sus` atoms as follows. For every constraint atom of the form $\&\texttt{min}(hb)\{s_1; \ldots; s_n\} \prec s$, we add the rule

$$\begin{aligned}
&:- def,\ \texttt{not}\ member\\
&\&\texttt{sus}(\texttt{head})\{max_{cl}\} = m :- \texttt{not}\ def
\end{aligned}$$

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

where m is a fresh integer variable representing the minimum of $s_1, \ldots, s_n$, and *def* and *member* are fresh propositional atoms defined by the following rules for each $i = 1, \ldots, n$:

$$\begin{aligned} &def :- \&\mathtt{df}(x_{i_1}), \ldots, \&\mathtt{df}(x_{i_k}) \\ &\&\mathtt{df}\{m\} :- def \\ &:- \&\mathtt{sus}(\mathtt{body})\{s_i\} < m \\ &member :- \&\mathtt{sus}(\mathtt{body})\{s_i\} = m \end{aligned}$$

with $x_{i_1}, \ldots, x_{i_k}$ are all integer variables occurring in s_i. In addition, if *hb* is `head`, we add the rule

$$\&\mathtt{sus}(\mathtt{head})\{m\} \prec s :- \&\mathtt{min}(\mathtt{head})\{s_1; \ldots; s_n\} \prec s$$

while if *hb* is `body`, we add the rule

$$\&\mathtt{min}(\mathtt{body})\{s_1; \ldots; s_n\} \prec s :- \&\mathtt{sus}(\mathtt{body})\{m\} \prec s$$

In the *seventh step*, we relate the fresh propositional atoms introduced in step five with the integer variables. For every integer variable x occurring in the program, we add

$$\&\mathtt{sum}\{0\} = x :- \mathtt{not}\ \&\mathtt{df}(x)$$

This rule ensures that undefined integer variables are assigned a unique value and, thus, avoids multiple stable models that only differ in the value assigned to undefined integer variables. For every constraint atom $\&\mathtt{sus}(\mathtt{head})\{s_1; \ldots; s_n\} \prec s$, we add the rules

$$\begin{aligned} &:- \&\mathtt{sus}(\mathtt{head})\{s_1; \ldots; s_n\} \prec s,\ \mathtt{not}\ \&\mathtt{sum}\{s_1; \ldots; s_n\} \prec s \\ &aux :- \&\mathtt{sus}(\mathtt{head})\{s_1; \ldots; s_n\} \prec s \\ &\&\mathtt{df}(x_1) :- aux \qquad \ldots \qquad \&\mathtt{df}(x_k) :- aux \end{aligned}$$

where $x_1, \ldots, x_k$ are all integer variables occurring in $s_1, \ldots, s_n, s$, and *aux* is a fresh propositional atom. For every constraint atom of the form $\&\mathtt{sus}(\mathtt{body})\{s_1; \ldots; s_n\} \prec s$, we add the rule

$$\&\mathtt{sus}(\mathtt{body})\{s_1; \ldots; s_n\} \prec s :- \&\mathtt{sum}\{s_1; \ldots; s_n\} \prec s, \&\mathtt{df}(x_1), \ldots, \&\mathtt{df}(x_k)$$

where $x_1, \ldots, x_k$ are all integer variables occurring in $s_1, \ldots, s_n, s$.

The *final step* replaces each atom of the form $\&\mathtt{df}(x)$ by a new propositional atom `def(x)`, so `clingcon` does not assume these atoms to be in the scope of a choice rule.

Every step of this transformation is feasible in linear time with respect to the size of the program. Thus, the overall transformation is linear as well.

`Flingo` supports the same useful modeling features for the ASP methodology as `clingo`, such as choice rules, defaults, and aggregates, now lifted to linear integer arithmetic. In Listing 1, we show a `flingo` program that models the configuration of a bike that consists of a frame and an optional bag, and the calculation of the total price.

```
1  price(frame,15).                    default_range(1,2).
2  select(frame).                    { select(bag) }.
3  &sus{V} = price(P)     :- select(P), price(P,V).
4  &in{L..U} =: price(P) :- select(P), default_range(L,U)
5                           not &sus{price(P)}<L, not &sus{price(P)}>U.
6  &sus{price(P) : select(P)} =: price(total).
```

Listing 1. Bike configuration with `flingo`

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

Lines 1 and 2 define the problem instance. Lines 3-6 provide the general problem encoding. Line 3 defines the price of a selected component as the one in the instance, when provided. Lines 4-5 define the price of a selected component using the default range defined in the instance: unless the price is provably outside the default range, the price of a selected component is defined by the default range. Finally, Line 6 defines the total price as the sum of the prices of the selected components. This example illustrates how `flingo` allows for a natural encoding of configuration problems with numeric attributes, where defaults, choices, and aggregates are common modeling features. `Flingo` has been used to model and solve configuration problems formulated in the COOM configuration language (Baumeister *et al.* 2026), where the empirical evaluation demonstrates its effectiveness in solving configuration problems with numeric attributes featuring large domains, where it significantly outperforms `clingo`.

5 Discussion

We have introduced `flingo`, a novel CASP system designed to bridge the gap between high-level ASP modeling and the operational requirements of numeric constraint solving.

Our contributions can be summarized as follows: (i) We presented a logic programming language that integrates default values, choice rules, and aggregates directly into constraint atoms. This preserves the standard ASP "look and feel" while leveraging the power of numeric backends. (ii) We defined the semantics of `flingo`, distinguishing between strict (`&sus`) and non-strict (`&sum`) operations to accommodate both traditional CASP behavior and ASP-style aggregates. (iii) We developed a translation from `flingo` to `clingcon`, combining the effectiveness of state-of-the-art CASP solvers with ASP declarativeness.

Lierler (2023) analyzes an extensive list of CASP systems from the literature, including the already mentioned `clingcon`. None of these systems described there allows for leaving constraint variables undefined, representing default values, or the use of aggregates involving both constraints variables and arbitrary conditions. Perhaps the closest system to `flingo` regarding these features is `clingo{f}`[5] (Balduccini 2013), which deals with non-Herbrand numerical functions, and allows for undefinedness and default values. However, `clingo{f}` does not perform numerical constraint solving. System s(CASP) (Arias *et al.* 2018) also mixes ASP with numerical constraints, but unlike standard CASP approaches based on grounding and solving, it follows a goal-directed evaluation as in Prolog. Under this paradigm, problems are modeled and solved in a substantially different way, so transferring examples from the ASP/CASP literature to s(CASP) is not always straightforward. If we wanted to extrapolate s(CASP) constraints to grounding-based CASP, their use would actually occur during the grounding phase, rather than in the solving phase as happens in the rest of CASP systems. Finally, aggregates are not directly representable inside s(CASP) constraints, but must be encoded through other Prolog predicates.

While the current implementation of `flingo` relies on `clingo`'s theory grammar, future work will focus on making the input language more natural and exploring further optimizations in the translation process to enhance scalability. The `flingo` system is publicly available at https://github.com/potassco/flingo.

[5] https://mbal.asklab.net/clingof

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

Competing interests

The authors declare none.

References

Arias, J., Carro, M., Salatar, E., Marple, K. and Gupta, G. 2018. Constraint answer set programming without grounding. *Theory and Practice of Logic Programming* 18, 3-4, 337–354.

Balduccini, M. 2013. ASP with non-herbrand partial functions: A language and system for practical use. *Theory and Practice of Logic Programming* 13, 4-5, 547–561.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461. https://doi.org/10.1017/S1471068417000138.

Baumeister, J., Hahn, S., Herud, K., Ostrowski, M., Reutelshoefer, J., Rühling, N., Schaub, T. and Wanko, P. 2026. Towards industrial-scale product configuration. In *Theory and Practice of Logic Programming*. Cambride University Press, 1–31. https://doi.org/10.1017/S1471068426100404.

Cabalar, P., Kaminski, R., Ostrowski, M. and Schaub, T. 2016. An ASP semantics for default reasoning with constraints. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, S. Kambhampati, Ed. IJCAI/AAAI Press, 1015–1021. https://doi.org/10.5555/3060621.3060762

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2020a. An ASP semantics for constraints involving conditional aggregates. In *Proceedings of the Twenty-Fourth European Conference on Artificial Intelligence (ECAI'20)*, G. De Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín and J. Lang, Eds. IOS Press, 664–671. https://doi.org/10.3233/FAIA200152.

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2020b. A uniform treatment of aggregates and constraints in hybrid ASP. In *Proceedings of the Seventeenth International Conference on Principles of Knowledge Representation and Reasoning (KR'20)*, D. Calvanese, E. Erdem and M. Thielscher, Eds. AAAI Press, 193–202. https://doi.org/10.24963/KR.2020/20.

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2025. Strong equivalence in answer set programming with constraints. CoRR, abs/2502.0430. https://doi.org/10.48550/arXiv.2502.04302.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T. and Wanko, P. 2016. Theory solving made easy with clingo 5. In *Technical Communications of the Thirty-Second International Conference on Logic Programming (ICLP'16)*, M. Carro, A. King, N. Saeedloei and M. De Vos, Eds. Vol. 52 of Open Access Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2:1–2:15.

Heyting, A. 1930. Die formalen Regeln der intuitionistischen Logik. In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, Deutsche Akademie der Wissenschaften zu Berlin, 42–56.

Lierler, Y. 2023. Constraint answer set programming: Integrational and translational (or SMT-based) approaches. *Theory and Practice of Logic Programming* 23, 1, 195–225. https://doi.org/10.1017/S1471068421000478.

Lifschitz, V. (1999) Action languages, answer sets, and planning. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. Apt, V. Marek, M. Truszczyński and D. Warren, Eds. Springer-Verlag, 357–373.

Mellarkod, V., Gelfond, M. and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence* 53, 1-4, 251–287.

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press

Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2005. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, G. Sutcliffe and A. Voronkov, Eds. Vol. 3835 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 36–50.

Pearce, D. 1997. A new logical characterisation of stable models and answer sets. In *Proceedings of the Sixth International Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*, J. Dix, L. Pereira and T. Przymusinski, Eds. Vol. 1216 of Lecture Notes in Computer Science, Springer-Verlag, 57–70. https://doi.org/10.1007/BFb0023801.

Van Gelder, A., Ross, K. and Schlipf, J. 1991. The well-founded semantics for general logic programs. *Journal of the ACM* 38, 3, 620–650.

https://doi.org/10.1017/S1471068426100519 Published online by Cambridge University Press