24th European Conference on Artificial Intelligence - ECAI 2020
Santiago de Compostela, Spain

An ASP semantics for Constraints involving Conditional Aggregates

Pedro Cabalar[1] and **Jorge Fandinno**[2] and **Torsten Schaub**[2] and **Philipp Wanko**[2]

Abstract. We elaborate upon the formal foundations of hybrid Answer Set Programming (ASP) and extend its underlying logical framework with aggregate functions over constraint values and variables. This is achieved by introducing the construct of conditional expressions, which allow for considering two alternatives while evaluating constraints. Which alternative is considered is interpretation-dependent and chosen according to an associated condition. We put some emphasis on logic programs with linear constraints and show how common ASP aggregates can be regarded as particular cases of so-called conditional linear constraints. Finally, we introduce a polynomial-size, modular and faithful translation from our framework into regular (condition-free) Constraint ASP, outlining an implementation of conditional aggregates on top of existing hybrid ASP solvers.

1 Introduction

Many real-world applications have a heterogeneous nature. Let it be in bio-informatics [13], hardware synthesis [18], or train scheduling [1], all cited ones consist of genuine qualitative and quantitative constraints. While the former often account for topological requirements, like reachability, the latter usually address (fine-grained) time or resource requirements.

The hybrid nature of such applications has led to mixed solving technology, foremost in the area of Satisfiability modulo Theories (SMT; [19]). Meanwhile, the need for hybridization has also prompted similar approaches in neighboring areas such as Answer Set Programming (ASP; [17]). However, while ASP solving technology is at eye-height with that of SAT and SMT, its true appeal lies in its high-level modeling language building on a non-monotonic semantics. Among others, this allows for expressing defaults and an easy formulation of reachability. When it comes to extending ASP with foreign reasoning methods, the design often follows the algorithmic framework of SMT and leaves semantic aspects behind. For instance, a popular approach is to combine ASP with Constraint Processing (CP; [9]), also referred to as Constraint ASP (CASP; [16]). This blends non-monotonic aspects of ASP with monotonic ones of CP but fails to provide a homogeneous representational framework. In particular, the knowledge representation capabilities of ASP, like defaults and aggregates, remain inapplicable to constraint variables.

We addressed this in [7] by integrating ASP and CP in the uniform semantic framework called *Here-and-There with constraints* (HT_C). The idea is to rebuild the logic of ASP from constraint atoms encapsulating arbitrary foreign constraints. This relies upon the logic of Here-and-There (HT; [15]) along with its non-monotonic extension, called Equilibrium Logic [20]. Although HT_C offers a uniform representation, for instance, featuring defaults for constraint variables, it still lacks an essential element of ASP's modeling language, namely, aggregates with conditional elements. This issue is addressed in the paper at hand. As an example, consider the hybrid ASP rule [3]

$$total(R) := sum\dot{\{}\ tax(P) : lives(P,R)\ \dot{\}}\ \leftarrow\ region(R) \qquad (1)$$

gathering the total tax revenue of each region R by summing up the tax liabilities of the region's residents, P. As a matter of fact, the calculation of tax liability is highly complex, and relies on defaults and discounts to address incomplete information, which nicely underlines the need for non-monotonic constraint variables. Once instantiated, $lives(P,R)$ and $region(R)$ are propositional atoms, while the entire rule head is regarded as a constraint atom, whose actual meaning eludes the ASP system (just as in lazy SMT [19]). The aggregate function sum is applied to a set of conditional expressions of the form $tax(P) : lives(P,R)$. This makes sure that each instantiated rule gathers only taxes accrued by the inhabitants of the respective region R. Although such an aggregation is very appealing from a modeling perspective, it leaves us with several technical challenges. First, the set of arguments is context-dependent and thus a priori unknown. Second, the identification of valid arguments necessitates the evaluation of Boolean conditions within foreign and thus opaque constraint atoms.

Accordingly, we start by developing a formal account of *conditional expressions*; they allow us to consider two alternatives while evaluating constraints. Which alternative is considered is interpretation-dependent and chosen according to the evaluation of a given condition, expressed as a logical formula. As a case-study, we focus on a syntactic fragment extending logic programs with linear constraints. We show that sum, $count$, min and max aggregate atoms in ASP constitute a special case of conditional linear constraints. Interestingly, our framework refrains from imposing the treatment of aggregate atoms as a whole; rather considering them as sub-expressions that can be combined with other arithmetic operations, leaving aggregate atoms a simple particular case. Finally, we develop a translation of programs with conditional expressions into CASP, which is itself condition-free. This enables the use of off-the-shelf CASP solvers as back-ends for implementing our approach. In this way, our translation allows us to delegate the responsibilities for evaluating constraints with conditional expressions: an ASP solver is in charge of evaluating conditions, while an associated CP solver only deals with condition-free constraints.

[1] University of Corunna, Spain
[2] University of Potsdam, Germany

[3] We put dots on top of braces, viz. "$\dot{\{}\ \ldots\ \dot{\}}$", to indicate *multisets*.

2 The Logic of Here-and-There with Constraints

The syntax of HT_C is based on a set of (constraint) variables $\mathcal{X}$ and constants or domain values [4] from some non-empty set $\mathcal{D}$. A *constraint atom* is some expression that is used to relate values of variables and constants according to the atom's semantics. We use $\mathcal{C}$ to denote the set of all constraint atoms.

Most useful constraint atoms have a syntax defined by some grammar or regular pattern: for instance, difference constraints are expressions of the form "$x - y \leq d$", where x and y are variables from $\mathcal{X}$ and d is a constant from $\mathcal{D}$. At the most general level, however, we only require that the constraint atom is represented by some string of symbols (we generally call *expression*), possibly, of infinite length. Apart from operators or punctuation symbols, this string may name some variables from $\mathcal{X}$, constants from $\mathcal{D}$, and, for convenience, also a special symbol $\mathbf{u} \notin \mathcal{D}$ that stands for *undefined*. We define the extended domain $\mathcal{D}_\mathbf{u} \stackrel{def}{=} \mathcal{D} \cup \{\mathbf{u}\}$. The set $vars(c) \subseteq \mathcal{X}$ collects all variables occurring in constraint c. We sometimes refer to a constraint atom using the notation $c[s]$ meaning that the expression for c contains some distinguished occurrence of subexpression s. We further write $c[s/s']$ to represent the syntactic replacement in c of subexpression s by s' as usual. We assume that $c[x/d] \in \mathcal{C}$ for every constraint atom $c[x] \in \mathcal{C}$, variable $x \in \mathcal{X}$ and $d \in \mathcal{D}_\mathbf{u}$. That is, replacing a variable by any element of the extended domain results in a syntactic valid constraint atom.

A *valuation* v over $\mathcal{X}, \mathcal{D}$ is some total function $v : \mathcal{X} \to \mathcal{D}_\mathbf{u}$ where $v(x) = \mathbf{u}$ represents that variable x is left undefined. Moreover, if $X \subseteq \mathcal{X}$ is a subset of variables, valuation $v|_X : X \to \mathcal{D}_\mathbf{u}$ stands for the projection of v on X. A valuation v can be alternatively represented as the set $\{(x, v(x)) \mid x \in \mathcal{X}, v(x) \in \mathcal{D}\}$, excluding pairs of form $(x, \mathbf{u})$ from the set. This representation allows us to use standard set inclusion for comparison. We thus write $v \subseteq v'$ to mean that $\{(x, v(x)) \mid x \in \mathcal{X}, v(x) \in \mathcal{D}\} \subseteq \{(x, v'(x)) \mid x \in \mathcal{X}, v'(x) \in \mathcal{D}\}$. This is equivalent to: $v(x) \in \mathcal{D}$ implies $v'(x) = v(x)$ for all $x \in \mathcal{X}$. We also allow for applying valuations v to fixed values, and so extend their type to $v : \mathcal{X} \cup \mathcal{D}_\mathbf{u} \to \mathcal{D}_\mathbf{u}$ by fixing $v(d) = d$ for any $d \in \mathcal{D}_\mathbf{u}$. The set of all valuations over $\mathcal{X}, \mathcal{D}$ is denoted by $\mathcal{V}_{\mathcal{X},\mathcal{D}}$ and $\mathcal{X}, \mathcal{D}$ dropped whenever clear from context.

We define the semantics of constraint atoms via *denotations*, which are functions $[\![\cdot]\!] : \mathcal{C} \to 2^\mathcal{V}$, mapping each constraint atom to a set of valuations. We require denotations $[\![\cdot]\!]$ to satisfy the following properties for all $c \in \mathcal{C}$, $x \in \mathcal{X}$, and any $v, v' \in \mathcal{V}$:

1. $v \in [\![c]\!]$ and $v \subseteq v'$ imply $v' \in [\![c]\!]$,
2. $v \in [\![c]\!]$ implies $v \in [\![c[x/v(x)]]\!]$,
3. if $v(x) = v'(x)$ for all $x \in vars(c)$ then $v \in [\![c]\!]$ iff $v' \in [\![c]\!]$.

Intuitively, Condition 1 makes constraint atoms behave monotonically. Condition 2 stipulates that denotations respect the role of variables as placeholders for values, that is, replacing variables by their assigned value does not change how an expression is evaluated. Condition 3 asserts that the denotation of c is fixed by combinations of values for $vars(c)$, while all other variables are irrelevant and may freely vary.

The flexibility of syntax and semantics of constraint atoms allows us to capture entities across different theories. For instance, assuming a value $\mathbf{t} \in \mathcal{D}$ for representing the truth value *true*, Boolean propositions can be modelled via constraint atoms having a denotation $[\![a]\!] = \{v \in \mathcal{V} \mid v(a) = \mathbf{t}\}$. As we mention above, we can cover different variants of linear equations. For example, difference constraint of the form "$x - y \leq d$" can be captured via constraint atoms of the same syntax "$x - y \leq d$" whose denotation is the expected

$$[\![\text{"}x - y \leq d\text{"}]\!] = \{v \in \mathcal{V} \mid v(x), v(y), d \in \mathbb{Z}, v(x) - v(y) \leq d\} ,$$

where $vars(\text{"}x - y \leq d\text{"}) = \{x, y\} \subseteq \mathcal{X}$ and $d \in \mathcal{D}$. Note that this difference constraint can only be satisfied when x and y hold an integer value and $d \in \mathbb{Z}$. For clarity, we simply remove quotes, when clear from the context. In what follows, we assume that integers and the truth value $\mathbf{t}$ are part of the domain, that is, $\{\mathbf{t}\} \cup \mathbb{Z} \subseteq \mathcal{D}$.

A formula φ over $\mathcal{C}$ is defined as

$$\varphi ::= \bot \mid c \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \quad \text{where } c \in \mathcal{C}.$$

We define $\top$ as $\bot \to \bot$ and $\neg\varphi$ as $\varphi \to \bot$ for every formula φ. By $var(\varphi)$ we denote the set of all variables occurring in all constraint atoms in formula φ. A *theory* is a set of formulas.

An *interpretation* over $\mathcal{X}, \mathcal{D}$ is a pair $\langle h, t\rangle$ of valuations over $\mathcal{X}, \mathcal{D}$ such that $h \subseteq t$. The interpretation is *total* if $h = t$.

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

1. $\langle h, t\rangle \not\models \bot$
2. $\langle h, t\rangle \models c$ *if* $h \in [\![c]\!]$
3. $\langle h, t\rangle \models \varphi \wedge \psi$ *if* $\langle h, t\rangle \models \varphi$ *and* $\langle h, t\rangle \models \psi$
4. $\langle h, t\rangle \models \varphi \vee \psi$ *if* $\langle h, t\rangle \models \varphi$ *or* $\langle h, t\rangle \models \psi$
5. $\langle h, t\rangle \models \varphi \to \psi$ *if* $\langle w, t\rangle \not\models \varphi$ *or* $\langle w, t\rangle \models \psi$ *for* $w \in \{h, t\}$

For compactness, we sometimes write $\langle t, t\rangle \models \varphi$ simply as $t \models \varphi$. In the rest of the paper, we assume a fixed underlying denotation.

A formula φ is a *tautology* when $\langle h, t\rangle \models \varphi$ for every interpretation $\langle h, t\rangle$ (wrt to some underlying denotation). Hence, a constraint atom c is tautologous whenever $[\![c]\!] = \mathcal{V}$. We say that an interpretation $\langle h, t\rangle$ is a model of a theory Γ, written $\langle h, t\rangle \models \Gamma$, when $\langle h, t\rangle \models \varphi$ for every $\varphi \in \Gamma$. We write $\Gamma \equiv \Gamma'$ if Γ and Γ' have the same models. We omit braces whenever Γ (resp. Γ') is a singleton.

A (total) interpretation $\langle t, t\rangle$ is an *equilibrium model* of a theory Γ, if $\langle t, t\rangle \models \Gamma$ and there is no $h \subset t$ such that $\langle h, t\rangle \models \Gamma$. Valuation t is also called a *stable model* of Γ and $SM(\Gamma)$ collects the set of stable models of Γ.

Finally, constraint atoms also allow us to capture constructs similar to aggregates. For instance, a constraint atom

$$sum\dot{\{}\ s_1, s_2, \dots\ \dot{\}} = s_0 \tag{2}$$

with each $s_i \in \mathcal{X} \cup \mathcal{D}$ for $0 \leq i$ can express that the (possibly infinite) sum of the values associated with the expressions $s_1, s_2, \dots$ is equal to s_0. The semantics of this constraint atom can be given by the following denotation:

$$[\![\ sum\dot{\{}s_1, s_2, \dots\dot{\}} = s_0\ ]\!] = \{v \in \mathcal{V} \mid v(s_i) \in \mathbb{Z}, i \geq 1, v(s_0) = \textstyle\sum_{i\geq 1} v(s_i)\}$$

This kind of construct allows us to gather the total tax revenue of the country with an expression of the form $sum\dot{\{}tax(p_1), tax(p_2), \dots\dot{\}}$ where p_1, p_2 are all the people in the country. We abbreviate this expression as $sum\dot{\{}\ tax(P)\ \dot{\}}$. Note however that such simple aggregate-like constructs do not allow for obtaining the total tax revenue of each region R, as in (1). What is missing is the possibility of applying the sum operation to a set of conditional expressions. We address this issue in the next section.

[4] Formally, we assume unique names for constants and use the same symbol for the constant name and the domain element. Note that, from a purely logical point of view, constraint variables are first order variables or 0-ary functions for which the standard name assumption is not assumed. As a result, its associated value is interpretation dependent.

3 Extending HT_C with Conditional Constraints

We now extend the logic HT_C with conditional expressions, inspired by the concept of aggregate elements in ASP [8]. While atoms are naturally conditioned by using an implication, a formal account is needed for conditioning subatomic expressions.

As before, we leave the syntax of expressions open as arbitrary strings, possibly combining elements from $\mathcal{X}$ and $\mathcal{D}_\mathbf{u}$, but we additionally assume now that some subexpressions, called *conditional expressions*, may have the form $(s|s'\colon\varphi)$ where φ is a formula called the *condition*. An expression or constraint atom which, in its turn, does not contain any conditional expression is called *condition-free*. We do not allow nested conditional expressions, that is, we assume that s and s' are condition-free expressions and that constraint atoms in formula φ are also condition-free. The intuitive reading of $(s|s'\colon\varphi)$ is "get the value of s if φ, or the value of s' otherwise." According to this reading, we must establish some connection among subexpressions $(s|s'\colon\varphi)$, s and s' both in the possible constraints $\mathcal{C}$ we can form and in the interrelation among their valuations. Therefore, if we have a constraint atom in $\mathcal{C}$ that has form $c[\tau]$ with $\tau = (s|s'\colon\varphi)$, we require that the constraint atoms $c[\tau/s], c[\tau/s'], c[\tau/\mathbf{u}]$ also belong to $\mathcal{C}$. Moreover, the valuations for these constraint atoms must satisfy

4. $v \in [\![\ c[\tau/\mathbf{u}]\ ]\!]$ implies $v \in [\![\ c[\tau/s]\ ]\!]$ and $v \in [\![\ c[\tau/s']\ ]\!]$

Condition 4 strengthens Condition 1 for the case of conditional constraint atoms. Intuitively, it says that, if a constraint does not hold for some subexpression, then it cannot hold when that subexpression is left undefined. For instance, if we include a constraint atom $x-(y|z\colon p)\leq 4$, then we must allow for forming the three constraint atoms $x-y\leq 4$ and $x-z\leq 4$ and $x-\mathbf{u}\leq 4$, too, and any valuation for the latter must also be a valuation for the former two.

Satisfaction of constraint atoms is defined by a previous syntactic unfolding of their conditional subexpressions, using some interpretation $\langle h,t\rangle$ to decide the truth values of formulas in conditions.

Definition 2 *Given an interpretation $\langle h,t\rangle$ and a conditional expression $\tau=(s|s'\colon\varphi)$ we define:*

$$eval_{\langle h,t\rangle}(\tau)=\begin{cases} s & \text{if } \langle h,t\rangle\models\varphi \\ s' & \text{if } \langle t,t\rangle\not\models\varphi \\ \mathbf{u} & \text{otherwise}\end{cases} \tag{3}$$

Note that the cases for $\langle h,t\rangle\models\varphi$ and $\langle t,t\rangle\not\models\varphi$ agree with our stated intuition that $(s|s'\colon\varphi)$ gets the value of s if φ is satisfied, or the value of s' when φ is not satisfied. The remaining case leaves the expression undefined when neither $\langle h,t\rangle\models\varphi$ nor $\langle t,t\rangle\not\models\varphi$ hold.

For a constraint atom $c\in\mathcal{C}$, we define $eval_{\langle h,t\rangle}(c)$ as the constraint atom that results from replacing each conditional expression τ in c by $eval_{\langle h,t\rangle}(\tau)$. Accordingly, $eval_{\langle h,t\rangle}(c)$ is condition-free. As an example, consider the conditional difference constraint

$$x-(y|3\colon p)\leq 4 \tag{4}$$

and the valuation $t=\{(x,7),(y,0)\}$. Then, the result of its evaluation $eval_{\langle t,t\rangle}(x-(y|3\colon p)\leq 4)$ is the condition-free difference constraint $x-3\leq 4$. Note that p is not satisfied by $\langle t,t\rangle$ and, thus, the conditional expression is replaced by its "else" part, viz. 3.

Satisfaction of formulas containing conditional terms is then naturally defined by replacing Condition 2 in Definition 1 by:

2′. $\langle h,t\rangle\models c$ if $h\in[\![\ eval_{\langle h,t\rangle}(c)\ ]\!]$

In our running example, we obtain $\langle t,t\rangle\models x-(y|3\colon p)\leq 4$ because $\langle t,t\rangle\models x-3\leq 4$.

Recall that, due to Condition 1 of denotations, condition-free constraint atoms behave monotonically, that is, $t\subseteq t'$ and $\langle t,t\rangle\models c$ imply $\langle t',t'\rangle\models c$. However, this no longer holds for conditional constraint atoms, which may behave non-monotonically. For instance, in our running example, the valuation $t'=\{(x,7),(y,0),(p,\mathbf{t})\}$ satisfies both $t\subseteq t'$ and $\langle t',t'\rangle\not\models x-(y|3\colon p)\leq 4$. This is because $eval_{\langle t',t'\rangle}(x-(y|3\colon p)\leq 4)$ yields a condition-free constraint atom different from the one above, namely $x-y\leq 4$. This constraint atom is not satisfied by $\langle t',t'\rangle$.

The following proposition tells us that usual properties of Here-and-There are still valid in this new extension. [5] Given any HT formula φ let $\varphi[\overline{a}/\overline{\alpha}]$ denote the uniform replacement of any tuple of atoms $\overline{a}=(a_1,\ldots,a_n)$ in φ by a tuple of arbitrary HT_C formulas $\overline{\alpha}=(\alpha_1,\ldots,\alpha_n)$.

Proposition 1 *Let $\langle h,t\rangle$ and $\langle t,t\rangle$ be two interpretations, and φ be a formula. Then,*

1. *$\langle h,t\rangle\models\varphi$ implies $\langle t,t\rangle\models\varphi$,*
2. *$\langle h,t\rangle\models\varphi\rightarrow\bot$ iff $\langle t,t\rangle\not\models\varphi$,*
3. *If φ is an HT tautology then $\varphi[\overline{a}/\overline{\alpha}]$ is an HT_C tautology.*

As an example of property 3 in Proposition 1, we can conclude, for instance, that $(x-(y|3\colon p)\leq 4)\rightarrow\neg\neg(x-(y|3\colon p)\leq 4)$ is an HT_C tautology because we can replace a in the HT tautology $a\rightarrow\neg\neg a$ by the HT_C formula $(x-(y|3\colon p)\leq 4)$. In particular, the third statement guarantees that all equivalent rewritings in HT are also applicable to HT_C. Note that every HT_C theory can be considered as an HT theory where each constraint atom is regarded as a proposition without further structure. As a result, the deductions made in HT about a theory are sound with respect to HT_C, even they may not be complete because HT misses the relation between atoms that derive from their internal structure.

4 Conditional linear constraints

We now focus on constraint atoms for dealing with *conditional linear constraints* on integer variables, or *linear constraints* for short. These constraints can be seen as a generalisation of regular aggregate atoms used in ASP. The syntax of linear constraints is defined as follows:

$$\lambda\ ::=\ d\ \mid\ d\cdot x \qquad\qquad \tau\ ::=\ \lambda\ \mid\ (\lambda|\lambda'\colon\varphi)$$

where $d\in\mathbb{Z}\subseteq\mathcal{D}$ is an integer constant, $x\in\mathcal{X}$ a constraint variable and φ a formula. We call τ a *term*; it is either a *linear term*, λ, or a *conditional term* of form $(\lambda|\lambda'\colon\varphi)$. A *linear expression* α is a possibly infinite sum $\tau_1+\tau_2+\ldots$ of terms τ_i. Then, a *linear constraint* is an inequality $\alpha\leq\beta$ of linear expressions α and β. We denote the set of variables occurring in α by $vars(\alpha)$. A linear constraint $\alpha\leq\beta$ is said to be in normal form if $\beta=d\in\mathbb{Z}$. We adopt some usual abbreviations. We simply write x instead of $1\cdot x$ and we directly replace the '$+$' symbol by (binary) '$-$' for negative constants. Moreover, when clear from the context, we sometimes omit the '$\cdot$' symbol and parentheses. We do not remove parentheses around conditional expressions. As an example, $-x+\ (3y|2y\colon\varphi)\ -2z$ stands for $(-1)\cdot x+(3\cdot y|2\cdot y\colon\varphi)+(-2)\cdot z$. Other abbreviations must be handled with care. In particular, we neither remove products of form $0\cdot x$ nor replace them by 0 (this is because x may be undefined, making the product undefined, too).

[5] An extended version of the paper including all proofs can be found here: `http://arxiv.org/abs/2002.06911`

We also extend the $\leq$ symbol in $\alpha \leq \beta$ to other comparison relations defined as the following abbreviations of formulas: $(\alpha < \beta) \stackrel{def}{=} \alpha \leq \beta \wedge \neg(\beta \leq \alpha)$, $(\alpha = \beta) \stackrel{def}{=} (\alpha \leq \beta) \wedge (\beta \leq \alpha)$ and $(\alpha \neq \beta) \stackrel{def}{=} (\alpha < \beta) \vee (\beta < \alpha)$. Notice that $\alpha \neq \beta$ is stronger than $\neg(\alpha = \beta)$ since the former requires α and β to have different values (and so, to be both defined), while the latter checks that $\alpha = \beta$ does not hold, and this includes the case in which any of the two is undefined. For any linear expression α, we define $def(\alpha) \stackrel{def}{=} \alpha \leq \alpha$ to stand for "α is defined," that is, α has a value.

For the semantics of linear constraints $\alpha \leq \beta$, we resort to [7] where α and β were *condition-free*, that is, they were sums of linear terms. As shown there, given any linear expression $\alpha = \lambda_1 + \lambda_2 + \ldots$ of that form, we can define their partial valuation v so that $v(\alpha)$ corresponds to:

$$v(\lambda_1 + \lambda_2 + \ldots) \stackrel{def}{=} \begin{cases} \mathbf{u} & \text{if } v(\lambda_i) \notin \mathbb{Z}, \text{for some } \lambda_i \\ \sum_{i \geq 0} v(\lambda_i) & \text{otherwise} \end{cases}$$

where $v(\lambda)$ for non-constant linear terms is defined as expected: $v(k \cdot x) \stackrel{def}{=} k \cdot v(x)$ if $v(x) \in \mathbb{Z}$, and $\mathbf{u}$ otherwise. In other words, a (condition-free) linear expression is evaluated as usual, except that it is undefined if it contains some undefined subterm (or eventually, some undefined variable). Then, the denotation of a condition-free linear constraint $\alpha \leq \beta$ is defined as:

$$[\![\, \alpha \leq \beta \,]\!] \quad \stackrel{def}{=} \quad \{v \mid v(\alpha), v(\beta) \in \mathbb{Z}, v(\alpha) \leq v(\beta)\}$$

These are valuations v in which $v(\alpha) \leq v(\beta)$ holds as expected, but both values $v(\alpha)$ and $v(\beta)$ must be defined integers. It is easy to see that, when $\alpha \leq \beta$ is condition-free, it can only be satisfied at h, if all variables occurring in the constraint are defined in h.

When we move to evaluating conditional terms, we further need some interpretation $\langle h, t\rangle$ to decide the satisfaction of formulas in conditions. The following result asserts that, if h assigns some value to a term τ (conditional or not), this value is also preserved in t.

Proposition 2 *For any term τ and interpretation $\langle h, t\rangle$, if $h(eval_{\langle h,t\rangle}(\tau)) \neq \mathbf{u}$, then $h(eval_{\langle t,t\rangle}(\tau)) = t(eval_{\langle h,t\rangle}(\tau))$.*

Now, for satisfaction of a conditional linear constraint, it suffices to apply Condition 2′ of the previous section:

$$\langle h, t\rangle \models \alpha \leq \beta \quad \text{if} \quad h \in [\![\, eval_{\langle h,t\rangle}(\alpha \leq \beta) \,]\!]$$

That is, we remove conditional terms by applying $eval_{\langle h,t\rangle}(\alpha \leq \beta)$ and then use the denotation of the resulting condition-free linear constraint.

An important consequence of the introduction of conditional terms is that a linear constraint $\alpha \leq \beta$ may now be satisfied even though some of its variables are undefined. For instance, the constraint

$$x + y > 1 \tag{5}$$

is not satisfied for interpretation $t = \{(y, 5)\}$ since $t(x) = \mathbf{u}$ and we cannot compute $\mathbf{u} + 5$. However, the conditional linear expression

$$(x|0\colon def(x)) + (y|0\colon def(y)) > 1 \tag{6}$$

checks whether $x + y$ is greater than one, but replaces any of these two variables by 0 when they are not defined. Take the example $t = \{(y, 5)\}$ where x is undefined and y is 5. Then, the result of applying function $eval_{\langle t,t\rangle}$ to (6) amounts to $0 + y > 1$, which is satisfied by t, that is, $t \in [\![\, 0 + y > 1 \,]\!]$.

Linear constraints offer a comfortable setting for a practical implementation of the sum aggregate we informally presented in the introduction, since their computation can be eventually delegated to a specialized constraint solver, as done in [7]. Besides, other common aggregates such as $count$, max and min can be defined relying on sum, as we show below. A straightforward encoding of (2) is the linear constraint

$$s_1 + s_2 + \ldots = s_0 \tag{7}$$

which abbreviates the conjunction of the following two linear constraints in normal form:

$$(-s_0 + s_1 + s_2 + \cdots \leq 0) \tag{8}$$

$$(s_0 - s_1 - s_2 - \cdots \leq 0) \tag{9}$$

Actually, this encoding allows us to extend the expressions s_i in the aggregate to be any linear or conditional term now. For the case of variables and constants $s_i \in \mathcal{X} \cup \mathcal{D}$, it is easy to check that the previous denotation we defined for sum, $[\![\, (2) \,]\!]$, is equal to $[\![\, (8) \,]\!] \cap [\![\, (9) \,]\!]$ and this, in its turn, means that $\langle h, t\rangle \models (2)$ iff $\langle h, t\rangle \models (8) \wedge (9)$, i.e. (2) and (7) are logically equivalent. The encoding of sum in (7) also clarifies its multiset behavior. For instance, atom

$$sum\dot{\{}\ x,\ y\ \dot{\}} > 1 \tag{10}$$

amounts now to (5), that is, $x + y > 1$ and there is no problem for collecting several occurrences of the same value as in interpretation $\{(x, 1), (y, 1)\}$ since $sum\dot{\{}1, 1\dot{\}} > 1$ amounts to $1 + 1 > 1$ which is obviously true.

Up to now, using conditional terms inside sum allows us writing, for instance,

$$sum\dot{\{}\ (x| - x\colon x \geq 0)\ ,\ y\ \dot{\}} > 1$$

to replace x in (10) by its absolute value, leading to the linear constraint $(x| - x\colon x \geq 0) + y > 1$. A more common situation, however, is to use a condition to decide whether a term should be included in the multiset or not. To this aim, we *redefine* the sum construct so that its syntax follows the general pattern

$$sum\dot{\{}\ \lambda_1\colon\varphi_1,\ \lambda_2\colon\varphi_2,\ \ldots\ \dot{\}} \tag{11}$$

where λ_i are linear terms and φ_i are condition-free formulas. Semantically, we assume that the new notation (11) is just an abbreviation of the linear expression $\theta_1 + \theta_2 + \ldots$ where each θ_i corresponds now to the conditional term:

$$(\lambda_i|0\colon\ \varphi_i \wedge def(\lambda_i)\ )$$

Note first that θ_i becomes 0 when its condition is not fulfilled, being a simple way to remove λ_i from the multiset of the sum. A second important observation is that we have reinforced the condition φ_i with the formula $def(\lambda_i)$ so that, if the term λ_i is undefined, the conditional also becomes 0 rather than $\mathbf{u}$. This behavior is interesting since some sums may be formed using undefined variables, either because no information has been provided for them, or because they come from unwanted, undefined terms generated from grounding, such as, say $tax(3/0)$.

When $\varphi_i = \top$, we allow to replace the multiset element "$\lambda_i : \top$" by "λ_i". As an example, notice that, under this new understanding, the aggregate atom (10) corresponds now to:

$$sum\dot{\{}\ x\colon\top,\ y\colon\top\ \dot{\}} > 1$$

and its translation into a linear constraint eventually corresponds to (6) rather than (5). Thus, the atom may still be true even though some variable is undefined, as we discussed before. In fact, it is not difficult to see that an aggregate expression α like (11) is always defined for any valuation v because $v(eval_{\langle v,v\rangle}(\alpha)) \in \mathbb{Z}$. Note that an aggregate expression may still be undefined for interpretations $\langle h, t\rangle$ due to a different evaluation of one of its conditions between h and t. We assume this understanding from now on.

5 Programs with linear constraints and aggregates

In this section, we provide a logic programming language based on a syntactic fragment of HT_C. In principle, logic programming rules can be built as usual, that is, implications (usually written backwards) $Head \leftarrow Body$ where $Head$ is a disjunction of atoms and $Body$ a conjunction of literals (that is, atoms or their default negation). However, a constraint atom in the head must be handled with care, since it does not provide any directionality for its set of variables. For instance, if we want that tot gets the value of some variable x, we cannot just use the rule $(tot = x \leftarrow \top)$ because there is no difference wrt. $x = tot$. In fact, without further information in the program, this rule would assign some arbitrary value for both x and tot to make the constraint atom true. To allow for directional assignments, [7] introduced the following construct. An *assignment* A for variable x is an expression of the form $x := \alpha \mathinner{..} \beta$ (with α, β linear expressions) standing for the formula

$$\neg\neg def(A) \wedge (def(A) \rightarrow \alpha \leq x \wedge x \leq \beta) \tag{12}$$

where $def(A) \stackrel{def}{=} def(\alpha) \wedge def(\beta)$. An assignment A is *applicable* in $\langle h, t\rangle$ when $\langle h, t\rangle \models def(A)$. The non-directional version of assignment A is defined as $\Phi(x := \alpha \mathinner{..} \beta) \stackrel{def}{=} (\alpha \leq x) \wedge (x \leq \beta)$. We see that an assignment A makes some additional checks regarding the definedness of α and β before imposing any condition on the variable x. In particular, $(def(A) \rightarrow \alpha \leq x \wedge x \leq \beta)$ guarantees that α and β can be used to fix the value of x, but not of variables in α and β themselves. On the other hand, $\neg\neg def(A)$ can be seen as a constraint checking that α and β must be eventually defined in the stable model, but through other rule(s) in the program. When the upper and lower bounds coincide, we just write $(x := \alpha) \stackrel{def}{=} (x := \alpha \mathinner{..} \alpha)$, that is, $\neg\neg def(\alpha) \wedge (def(\alpha) \rightarrow x = \alpha)$. As a result, $\Phi(x := \alpha) = (x = \alpha)$. The following proposition relates an assignment A and its non-directional version $\Phi(A)$ in some particular interesting cases.

Proposition 3 *Given an assignment $A = (x := \alpha \mathinner{..} \beta)$, we have*

1. $A \wedge def(A) \equiv \Phi(A)$
2. $\neg A \equiv \neg\Phi(A)$

In particular, if $A = (x := \alpha \mathinner{..} \beta)$ contains no variables other than the assigned x, then $def(A) = \top$ and so $A \equiv \Phi(A)$.

We are now ready to introduce the syntactic class of logic programs. A *linear constraint rule*, or LC-rule for short, is a rule of the form:

$$A_1; \ldots; A_n \leftarrow B_1, \ldots, B_m, \neg B_{m+1}, \ldots, \neg B_k \tag{13}$$

with $n \geq 0$ and $k \geq m \geq 0$, where each A_i is an *assignment* and each B_j is a *linear constraint*. For any rule r like (13), we let $H(r)$ stand for the set $\{A_1, \ldots, A_n\}$ and $B(r)$ be the set $\{B_1, \ldots, B_m, \neg B_{m+1}, \ldots, \neg B_k\}$. By abuse of notation, we sometimes use $H(r)$ to stand for the disjunction $\bigvee H(r)$ and $B(r)$ to stand for the conjunction $\bigwedge B(r)$. An HT_C theory consisting of LC-rules only is called *LC-program*.

As an example, the following LC-rule corresponds to one of the ground instances of the rule (1) in the introduction.

$$total(r) := \alpha \ \leftarrow \ region(r) \tag{14}$$

with $\alpha = sum\{\, tax(p_1) : lives(p_1, r), tax(p_2) : lives(p_2, r), \ldots \,\}$. Intuitively, rule (14) states that the value of $total(r)$ is equal to the value computed by α whenever the Boolean variable $region(r)$ holds. This rule says nothing about the value of $total(r)$ if the condition $region(r)$ does not hold. As a result, in this case, and in absence of other rules defining its value, the value of $total(r)$ should be left undefined. This has some analogy to the fact that every true atom in a stable model must be supported by some rule in the program. Similarly, we may expect that every *defined variable* in a stable model is also supported by some LC-rule. This result is not immediately obvious, since we allow assignments with conditional linear expressions in the head (which includes aggregate expressions). We lift next the notion of supported model from standard ASP to the case of LC-programs. Given a valuation v, a variable $x \in \mathcal{X}$ and an LC-program Π, we say that value $d \in \mathcal{D}$ is *supported* for x wrt. Π and v if there is a rule $r \in \Pi$ and an assignment of the form $x := \alpha..\beta$ in the head of r satisfying the following conditions:

1. $v(x) = d$ and $v(\alpha) \leq d \leq v(\beta)$, so both $v(\alpha), v(\beta) \in \mathbb{Z}$
2. $v \not\models A'$ for every assignment of the form $y := \alpha'..\beta'$ in the head of r where y is a variable different from x,
3. $v \models B(r)$.

We say that a valuation v is *supported* wrt. some LC-program Π if every $v(x) \neq \mathbf{u}$ is a supported value for x wrt. Π and v.

Proposition 4 *Every stable model of any LC-program is also supported.*

Notice that an LC-rule may contain nested implications in the head due to the presence of assignments. The following theorem shows that LC-rules can be unfolded into a set of implications where the antecedent is a conjunction of literals and the consequent is a disjunction of constraint atoms. More formally, an HT_C-rule is an expression of the form of (13) with $n \geq 0$ and $k \geq m \geq 0$, where each A_i and B_j is a *linear constraint*. Note that the difference between LC- and HT_C-rules resides in the fact that the head of the former are build of assignments while the head of the later are build of linear constraints.

Theorem 1 *A rule r as in* (13) *is equivalent to the conjunction $\bigwedge_{\Delta \subseteq H(r)} \Psi_\Delta$ where Ψ_Δ is the following implication:*

$$\bigvee_{A \in \Delta} \Phi(A) \ \leftarrow \ B(r) \wedge \bigwedge_{A \in \Delta} def(A) \wedge \bigwedge_{A' \in H(r) \setminus \Delta} \neg\Phi(A')$$

The implication above is not an HT_C-rule yet: note that each $\Phi(A)$ in the head may be a conjunction of the form $\alpha \leq x \wedge x \leq \beta$ and each $\neg\Phi(A')$ in the body can be a negated conjunction of a similar form that, by De Morgan laws, becomes a disjunction $\neg(\alpha \leq x) \vee \neg(x \leq \beta)$. Still, these constructs can be easily unfolded in HT by distributivity properties that guarantee that $\varphi \wedge \varphi' \leftarrow \psi$ is equivalent to the pair of rules $\varphi \leftarrow \psi$ and $\varphi' \leftarrow \psi$ and something analogous for disjunctions in the body. Therefore, every LC-rule can be rewritten as a set of HT_C-rules. As a small illustration, take the LC-rule (14) with a single head assignment $A = (total(r) := \alpha)$. We can only form two sets $\Delta_1 = \{A\}$ and $\Delta_2 = \emptyset$ that, according to Theorem 1, generate the respective implications:

$$total(r) = \alpha \ \leftarrow \ region(r) \wedge def(\alpha) \tag{15}$$

$$\bot \leftarrow region(r) \wedge \neg(total(r) = \alpha) \tag{16}$$

Moreover, since the aggregate satisfies $t(eval_{\langle t,t\rangle}(\alpha)) \neq \mathbf{u}$ for any valuation t, we can prove that $\langle t,t\rangle \models def(\alpha)$ and so, by Proposition 1, constraint (16) can be equivalently transformed into

$$\bot \leftarrow region(r) \wedge \neg(total(r) = \alpha) \wedge def(\alpha)$$

that is an HT consequence of (15), and so, can be eventually removed.

6 Implementation Outline

In this section, we propose a method for implementing LC-programs with conditional aggregates that relies on a syntactic transformation for removing conditional expressions. This transformation produces a condition-free set of HT_C-rules that can then be solved as in [7], using an off-the-shelf CASP solver as a back-end. In fact, the reduction to conditional-free syntax can be defined for arbitrary HT_C theories, not only LC-programs. Given a conditional expression $\tau = (s|s'\colon \varphi)$, we define formula $\delta(\tau)$ as the conjunction of the following implications

$$\varphi \wedge def(s) \;\rightarrow\; x_\tau = s \tag{17}$$
$$\neg\varphi \wedge def(s') \;\rightarrow\; x_\tau = s' \tag{18}$$
$$\varphi \wedge def(x_\tau) \;\rightarrow\; x_\tau = s \tag{19}$$
$$\neg\varphi \wedge def(x_\tau) \;\rightarrow\; x_\tau = s' \tag{20}$$
$$def(x_\tau) \;\rightarrow\; \varphi \vee \neg\varphi \tag{21}$$

where x_τ is a fresh variable locally occurring in $\delta(\tau)$. These implications are used to guarantee that the new auxiliary variable x_τ gets exactly the same value as τ under any $\langle h,t\rangle$ interpretation. In that way, the conditional expression can be safely replaced by x_τ in the presence of $\delta(\tau)$. In particular, (17) and (18) alone suffice to guarantee that x_τ gets the value of s when φ holds, or the value of s' if $\neg\varphi$ instead. To illustrate the effect of (17)-(18), suppose we have the formula $sum\{x,y\} > 1 \rightarrow p$ and the fact $y = 5$. This amounts to the theory:

$$y = 5 \qquad (x|0\colon def(x)) + (y|0\colon def(y)) > 1 \;\rightarrow\; p$$

whose unique stable model is $t = \{(p,\mathbf{t}),(y,5)\}$ where p becomes true even though x has no value. If we replace, say, $\tau = (y|0\colon def(y))$ by x_τ we get

$$y = 5 \qquad (x|0\colon def(x)) + x_\tau > 1 \;\rightarrow\; p$$

and that (17) and (18) respectively correspond to:

$$def(y) \rightarrow x_\tau = y \qquad\qquad \neg def(y) \rightarrow x_\tau = 0$$

after minor simplifications. The resulting theory also has a unique stable model $t' = t \cup \{(x_\tau,5)\}$ that precisely coincides with t when projected on the original set of variables $\{x,y\}$.

We see that (17) and (18) provide the expected behavior in this case and, in fact, are enough to cover the translation $\delta(\tau)$ of any conditional expression inside an LC-program. This is because defined variables in LC-programs need to be supported and, by construction, x_τ cannot occur in the left hand side of any assignment. Hence, the only way in which x_τ can be defined is because the body of either (17) or (18) is satisfied.

Implications (19)-(21) are additionally required for the translation of arbitrary theories. Their need is best illustrated when the constraint atom is used as a rule head or a fact, since this may cause some effect on the involved variables. The formulas (19) and (20) ensure that variables in s or s' take the correct value when x_τ is defined. Take, for instance, the theory only containing a conditional constraint atom $(y|0\colon \top) = 5$. This formula is logically equivalent to $y = 5$ and, thus, it has the stable model $\{(y,5)\}$. If we replace it by some x_τ and only add (17) and (18), we get the theory:

$$x_\tau = 5 \qquad \top \wedge def(y) \rightarrow x_\tau = y \qquad \bot \wedge def(y) \rightarrow x_\tau = 0$$

(where the last formula is tautological) whose unique stable model is $\{(x_\tau,5)\}$ with y undefined. Now, adding (19) and (20) we also get:

$$\top \wedge def(x_\tau) \rightarrow x_\tau = y \qquad \bot \wedge def(x_\tau) \rightarrow x_\tau = 0$$

(again, the last formula is a tautology) whose unique stable model is now $\{(x_\tau,5),(y,5)\}$ as expected. Finally, (21) is added to ensure that x_τ is only defined in an interpretation $\langle h,t\rangle$ if either $\langle h,t\rangle \models \varphi$ or $\langle h,t\rangle \models \neg\varphi$. This correspond to the "otherwise" case in Definition 3. To show its effect, take the example:

$$(y|y\colon p) = 5 \qquad\qquad \neg p \rightarrow \bot$$

This program has a unique stable model $t = \{(p,\mathbf{t}),(y,5)\}$. Interpretation $\langle h,t\rangle$ with $h = \emptyset$ is not a model because it does not satisfy $eval_{\langle h,t\rangle}((y|y\colon p) = 5)$ since $(y|y\colon p)$ is evaluated to $\mathbf{u}$ and $\mathbf{u} = 5$ is not satisfied. On the other hand,

$$x_\tau = 5 \qquad\qquad p \wedge def(y) \;\rightarrow\; x_\tau = y$$
$$\neg p \rightarrow \bot \qquad\qquad p \wedge def(x_\tau) \;\rightarrow\; x_\tau = y$$

has no stable model. Note that $\langle h',t'\rangle$ with $h' = \{(y,5),(x_\tau,5)\}$ and $t' = \{(p,\mathbf{t}),(y,5),(x_\tau,5)\}$ is a model. This is solved by adding the following rule corresponding to (21)

$$def(x_\tau) \rightarrow p \vee \neg p$$

which is not satisfied by $\langle h',t'\rangle$.

Let us now formalize these intuitions. We assume that if $c[\tau] \in \mathcal{C}$ is a constraint atom then $x_\tau = s$ and $x_\tau = s'$ and also $c[\tau/x_\tau]$ are constraint atoms. We also assume that for every pair of subexpressions s, s', if $s = s'$ is a constraint atom, so they are $s' = s$ and $s = s$ and that if $s = s'$ and $c[s] \in \mathcal{C}$ are a constraint atoms, then $c[s/s'] \in \mathcal{C}$ is also a constraint atom. In other words, if two expressions are of a type that can be syntactically compared, then replacing one expression by the other also results in a syntactically valid expression. Furthermore, we require that our denotation behaves as expected wrt. equality atoms "=" and substitutions of subexpressions, that is, that it satisfies the following property

5. $v \in [\![\, s = s' \,]\!]$ implies $v \in [\![\, c[s] \,]\!]$ iff $v \in [\![\, c[s/s'] \,]\!]$

for any expressions s, s' such that $c[s]$ and $s = s'$ are constraint atoms. This condition is similar to Property 2 in Section 2 but relates equal subexpressions instead of a variable with its held value. It is easy to see that the denotation for linear constraints discussed in Section 4 does satisfy this property. We also extend the definition of $def(s)$ to arbitrary (non-linear) expressions: if s is an expression which is not linear, then $def(s)$ is an abbreviation for $s = s$.

Given a theory Γ, by $\delta(\Gamma)$, we denote the theory resulting from replacing in Γ every occurrence of every conditional expression τ by a corresponding fresh variable x_τ and adding to the result of this replacement the formula $\delta(\tau)$ for every conditional expression τ occurring in Γ. Furthermore, given an interpretation $\langle h,t\rangle$, by $\langle h,t\rangle_\tau = \langle h_\tau, t_\tau\rangle$, we denote an interpretation that satisfies the following two conditions:

$$v_\tau(x) = v(x) \text{ for } x \in \mathcal{X}\setminus\{x_\tau\} \qquad v_\tau(x_\tau) = \begin{cases} v(s) & \text{if } \langle v,t\rangle \models \varphi \\ v(s') & \text{if } \langle t,t\rangle \not\models \varphi \\ \mathbf{u} & \text{otherwise} \end{cases}$$

with $v \in \{h, t\}$. It is easy to see the correspondence of $\langle h, t\rangle_\tau$ with the *eval* function (3): it ensures that the value of x_τ in the valuation v_τ is the same as the conditional expression τ in v for $v \in \{h, t\}$.

Observation 1 *Any interpretation $\langle h, t\rangle$ and conditional expression τ satisfy $h_\tau(x_\tau) = h(eval_{\langle h,t\rangle}(\tau))$.*

Now we can relate the construction of interpretation $\langle h, t\rangle_\tau$ with the set of implications $\delta(\tau)$.

Proposition 5 *Any conditional expression τ and any model $\langle h, t\rangle$ of $\delta(\tau)$ satisfy $\langle h, t\rangle = \langle h, t\rangle_\tau$.*

In other words, $\delta(\tau)$ ensures that x_τ and τ have the same evaluation in all models of the resulting theory. Combining Proposition 5 and Observation 1, we immediately obtain the following result.

Corollary 1 *Let Γ be some theory and τ be some conditional expression. Then, $\Gamma \cup \{\delta(\tau)\} \equiv \Gamma[\tau/x_\tau] \cup \{\delta(\tau)\}$.*

In other words, replacing τ by x_τ has no effect in a theory that contains the set of implications $\delta(\tau)$. To finish the formalization of the correspondence between Γ and $\delta(\Gamma)$, we resort to the notion of *projected strong equivalence* [2, 10]. Given a set of variables X, let $SM(\Gamma)|_X \stackrel{def}{=} \{v|_X \mid v \in SM(\Gamma)\}$ as expected. Two theories Γ and Γ' for alphabet $\mathcal{X}$ are said to be *strongly equivalent for a projection* onto $X \subseteq \mathcal{X}$, denoted $\Gamma \equiv_s^X \Gamma'$, iff the equality $SM(\Gamma \cup \Delta)|_X = SM(\Gamma \cup \Delta')|_X$ holds for any theory Δ over subalphabet X.

Proposition 6 *Let Γ be some theory, τ be some conditional expression and $X = \mathcal{X} \setminus \{x_\tau\}$. Then, $\Gamma \equiv_s^X \Gamma \cup \{\delta(\tau)\}$.*

Theorem 2 *Let Γ be some theory, τ be some conditional expression and $X = \mathcal{X} \setminus \{x_\tau\}$. Then, $\Gamma \equiv_s^X \Gamma[\tau/x_\tau] \cup \{\delta(\tau)\}$.*

Theorem 2 is a strongly equivalence result and, thus, the replacement can be made independently of the rest of the theory. Therefore, this confirms that our translation is strongly faithful and modular. It is also easy to see that this translation is linear in the size of the program. In fact, we only introduce as many auxiliary variables as conditional expressions exist in the program and two new constraint atoms, namely $x_\tau = s$ and $x_\tau = s'$ for each new variable. Furthermore, the only requirement for the underlying CP solver is to allow the condition-free versions of the constraints appearing in our theory plus equality constraint atoms. These two requirements are satisfied by off-the-self CASP solvers when our theory deals with linear constraints. Finally, note that the result of translation δ is a condition-free HT_C-theory. In particular, when applied to an LC-program, the result is a set containing condition-free LC- and HT_C-rules. LC-rules can be translated into HT_C-implications as illustrated by Theorem 1. Although, in general, this translation requires exponential space, a polynomial-size variation is possible. This is achieved by using auxiliary variables in the fashion of [24]. The resulting condition-free theory can then be translated into CASP by using further auxiliary Boolean variables to capture when constraints are defined [7].

So far, we have only dealt with *sum* aggregate functions, though ASP systems usually allow for *count*, *max* and *min* operations too. Compiling $count\dot{\{}\varphi_1, \varphi_2, \dots\dot{\}}$ into sums is easy: we just transform it into $sum\dot{\{}1 : \varphi_1, 1 : \varphi_2, \dots\dot{\}}$. The encoding for the *min* aggregate is more involved: We replace an expression of the form

$$min\dot{\{}s_1 : \varphi_1,\ s_2 : \varphi_2, \dots\dot{\}} \tag{22}$$

by a fresh variable x_{min} and add the formulas:

$$def(x_{min}) \leftrightarrow count\dot{\{}\varphi_1 \wedge def(s_1),\ \varphi_2 \wedge def(s_2), \dots\dot{\}} \geq 1 \tag{23}$$

$$def(x_{min}) \rightarrow \alpha_{min} \wedge \beta_{min} \tag{24}$$

where α_{min} and β_{min} are the following respective expressions

$$count\dot{\{}\varphi_1 \wedge (s_1 < x_{min}), \varphi_2 \wedge (s_2 < x_{min}), \dots\dot{\}} \leq 0$$

$$count\dot{\{}\varphi_1 \wedge (s_1 \leq x_{min}),\ \varphi_2 \wedge (s_2 \leq x_{min}), \dots\dot{\}} \geq 1$$

We can then simply encode $max\dot{\{}s_1 : \varphi_1, s_2 : \varphi_2, \dots\dot{\}}$ as the expression $min\dot{\{} - s_1 : \varphi_1, -s_2 : \varphi_2, \dots\dot{\}}$. Intuitively, (23) requires that x_{min} is defined iff the multiset has at least one value. As a result, when the aggregate is defined, x_{min} can take any value. Expression α_{min} in (24) ensures that no element in the aggregate is strictly smaller than x_{min} while β_{min} guarantees that at least one element is smaller or equal to x_{min}.

This translation is similar to the one for *min* aggregates in regular ASP [3], but taking care of the definiteness of variables. It is worth mentioning that in most approaches to aggregates in regular ASP [11, 12, 22, 21, 23], replacing the aggregate expression by an auxiliary variable results in a non-equivalent formula, while it is safe in our framework. In this sense, our approach behaves similarly to ASP aggregates as defined in [6, 14]. We discuss this relation in more detail in the next Section.

As a simple example, consider the translation of the expression $min\dot{\{}x, y\dot{\}}$. If both x and y are undefined, so it is the right hand side of (23). This forces x_{min} to be undefined as well. Otherwise, the right hand side of (23) is defined and so it is x_{min}. In such case, a valuation t satisfies the following two formulas:

$$sum\dot{\{}1 : x < x_{min},\ 1 : y < x_{min}\dot{\}} \leq 0$$

$$sum\dot{\{}1 : x \leq x_{min},\ 1 : y \leq x_{min}\dot{\}} \geq 1$$

If either $t(x)$ or $t(y)$ are strictly smaller than $t(x_{min})$, then the first formula corresponding to α_{min} is violated. Similarly, if both $t(x)$ and $t(y)$ are strictly greater than $t(x_{min})$, then β_{min} is violated. Hence, $t(x_{min})$ is the minimum of $t(x)$ and $t(y)$. That is, the value of x_{min} indeed is the minimal value of x and y whenever both x and y are defined. Finally, in any interpretation $\langle h, t\rangle$ in which any of the variables is defined in t but not in h, we get that that the right hand side of (23) is also defined in t but not in h. The same applies to x_{min} as a result. This last case happens as a result of a cyclic dependence as, for instance, in $x = 1 \leftarrow min\dot{\{}x, y\dot{\}} \geq 1$.

7 Discussion

HT_C is a logic to capture non-monotonic constraint theories that permits assigning default values to constraint variables. Since HT and thus also ASP are special cases of this logic, it provides a uniform framework integrating ASP and CP on the same semantic footing. We elaborate on this logic by incorporating *aggregate expressions*, one of the essential elements in ASP's modeling language. This was missing so far. We accomplished this by introducing the construct of *conditional expressions* that allow us to consider two alternatives while evaluating constraints. With it, we can also deal with aggregate expressions on the constraint side. To the best of our knowledge, this is the first account that allows for the use of ASP-like aggregate expressions within constraints. In particular, we focus on a fragment of HT_C that constitutes an extension of logic programs with conditional linear constraints, called LC-programs. We show that *sum*,

$count$, max and min aggregate atoms can be regarded as special cases of conditional linear constraints and, in fact, our formalism permits their use as terms inside linear constraints.

Condition-free HT_C captures a fragment of ASP with partial functions [4, 5] where constraint variables correspond to 0-ary evaluable functions. This work was extended with intensional sets in [6], where it is shown to capture Gelfond-and-Zhang semantics for ASP with aggregate atoms [14]. Recall that a characteristic feature of this approach is the adherence to the *vicious circle principle*, stating that "*no object or property may be introduced by a definition that depends on that object or property itself.*" As a result, the ASP program consisting of the single rule

$$p(a) \ \leftarrow \ count\{\, X : p(X) \,\} \geq 0. \tag{25}$$

has no stable model under this semantics. This distinguishes it from other alternatives [11, 12, 22, 21, 23] that consider $\{p(a)\}$ as a stable model. Note that the only rule supporting $p(a)$ depends on a set which contains it as one of its elements. Thus, in accordance to the above rationality principle, this is rejected. In our framework, we can write a very simple rule which reflects a similar behavior

$$x := 1 \ \leftarrow \ sum\{\, x : \top \,\} \geq 0. \tag{26}$$

As above, this theory has no stable model, showing that our framework adheres to this rationality principle as well. As happens with the relation between condition-free HT_C and ASP with partial functions, we conjecture that our framework captures a fragment of [6]. As a result, an *instantiation* process similar to the one in [11] would also allow us to capture [14]. Confirming this conjecture is ongoing work.

Recall that [14] showed that Gelfond-and-Zhang semantics coincides with the other alternatives [11, 12, 23] on programs which are *stratified on aggregates*, which is the fragment covered by the ASP Core 2 semantics [8]. We have also considered the definition of a semantics for constraint aggregates closer to Ferraris' [12] but the implementation for the latter is not so straightforward as the one shown in this paper and is still under study.

Despite the close relation with [6], a distinctive feature of our approach is its orientation as a general abstraction of a hybrid solver with the ASP solver in charge of evaluating the Boolean part of the theory while relegating the evaluation of constraint atoms to dedicated CP solvers. Though, we focus here on reasoning with linear constraints, our formalism can also be regard as an abstraction of a multi-theory solver where the semantics of different constraint atoms are evaluated by different CP solvers. Interestingly, we provide a polynomial translation from HT_C with conditional constraints to (condition-free) CASP theories. This allows us to use off-the-shelf CASP solvers as back-ends for implementing our approach. This is also ongoing work.

ACKNOWLEDGEMENTS

This work was partially supported by Ministry of Science and Innovation, Spain (TIC2017-84453-P), Xunta de Galicia, Spain (GPC ED431B 2019/03 and 2016-2019 ED431G/01, CITIC Research Center), and German Research Foundation, Germany (SCHA 550/11).

REFERENCES

[1] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, and P. Wanko, 'Train scheduling with hybrid ASP', in *Proceedings of the Fifteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'19)*, pp. 3–17. Springer, (2019).

[2] F. Aguado, P. Cabalar, J. Fandinno, D. Pearce, G. Pérez, and C. Vidal, 'Forgetting auxiliary atoms in forks', *Artificial Intelligence*, **275**, 575–601, (2019).

[3] M. Alviano, W. Faber, and M. Gebser, 'Rewriting recursive aggregates in answer set programming: Back to monotonicity', *Theory and Practice of Logic Programming*, **15**(4-5), 559–573, (2015).

[4] M. Balduccini, 'A "conservative" approach to extending answer set programming with nonherbrand functions', in *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, 24–39, Springer, (2012).

[5] P. Cabalar, 'Functional answer set programming', *Theory and Practice of Logic Programming*, **11**(2-3), 203–233, (2011).

[6] P. Cabalar, J. Fandinno, L. Fariñas del Cerro, and D. Pearce, 'Functional ASP with intensional sets: Application to Gelfond-Zhang aggregates', *Theory and Practice of Logic Programming*, **18**(3-4), 390–405, (2018).

[7] P. Cabalar, R. Kaminski, M. Ostrowski, and T. Schaub, 'An ASP semantics for default reasoning with constraints', in *Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, pp. 1015–1021. IJCAI/AAAI Press, (2016).

[8] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format, 2012.

[9] R. Dechter, *Constraint Processing*, Morgan Kaufmann Publishers, 2003.

[10] T. Eiter, H. Tompits, and S. Woltran, 'On solution correspondences in answer set programming', in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, pp. 97–102. Professional Book Center, (2005).

[11] W. Faber, G. Pfeifer, and N. Leone, 'Semantics and complexity of recursive aggregates in answer set programming', *Artificial Intelligence*, **175**(1), 278–298, (2011).

[12] P. Ferraris, 'Logic programs with propositional connectives and aggregates', *ACM Transactions on Computational Logic*, **12**(4), 25, (2011).

[13] C. Frioux, T.Schaub, S. Schellhorn, A. Siegel, and P. Wanko, 'Hybrid metabolic network completion', *Theory and Practice of Logic Programming*, **19**(1), 83–108, (2019).

[14] M. Gelfond and Y. Zhang, 'Vicious circle principle, aggregates, and formation of sets in ASP based languages', *Artificial Intelligence*, **275**, 28–77, (2019).

[15] A. Heyting, 'Die formalen Regeln der intuitionistischen Logik', in *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, 42–56, Deutsche Akademie der Wissenschaften zu Berlin, (1930).

[16] Y. Lierler, 'Relating constraint answer set programming languages and algorithms', *Artificial Intelligence*, **207**, 1–22, (2014).

[17] V. Lifschitz, 'What is answer set programming?', in *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pp. 1594–1597. AAAI Press, (2008).

[18] K. Neubauer, P. Wanko, T. Schaub, and C. Haubelt, 'Exact multi-objective design space exploration using ASPmT', in *Proceedings of the Twenty-first Conference on Design, Automation and Test in Europe (DATE'18)*, pp. 257–260. IEEE Computer Society Press, (2018).

[19] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, 'Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T)', *Journal of the ACM*, **53**(6), 937–977, (2006).

[20] D. Pearce, '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)*, pp. 57–70. Springer, (1997).

[21] N. Pelov, M. Denecker, and M. Bruynooghe, 'Well-founded and stable semantics of logic programs with aggregates', *Theory and Practice of Logic Programming*, **7**(3), 301–353, (2007).

[22] P. Simons, I. Niemelä, and T. Soininen, 'Extending and implementing the stable model semantics', *Artificial Intelligence*, **138**(1-2), 181–234, (2002).

[23] T. Son and E. Pontelli, 'A constructive semantic characterization of aggregates in answer set programming', *Theory and Practice of Logic Programming*, **7**(3), 355–375, (2007).

[24] G. Tseitin, 'On the complexity of derivation in the propositional calculus', *Zapiski nauchnykh seminarov LOMI*, **8**, 234–259, (1968).