An ASP Semantics for Default Reasoning with Constraints

Pedro Cabalar[1] **Roland Kaminski**[2] **Max Ostrowski**[2] **Torsten Schaub**[2,3]
[1]University of Corunna, Spain [2]University of Potsdam, Germany [3]INRIA, France

Abstract

We introduce the logic of *Here-and-There with Constraints* in order to capture constraint theories in the non-monotonic setting known from Answer Set Programming (ASP). This allows for assigning default values to constraint variables or to leave them undefined. Also, it provides us with a semantic framework integrating ASP and Constraint Processing in a uniform way. We put some emphasis on logic programs dealing with linear constraints on integer variables, where we further introduce a directional assignment operator. We elaborate upon the formal relation and implementation of these programs in terms of Constraint ASP, sketching an existing system.

1 Introduction

Although Answer Set Programming (ASP; [Lifschitz, 2008]) has become a prime candidate for knowledge representation and reasoning, it falls short of succinctly representing variables over large numeric domains. So far, this was addressed by hybridizing ASP with Constraint Processing (CP; [Dechter, 2003]), leading to the subarea of Constraint ASP (CASP; [Lierler, 2014]). In fact, the design of most CASP approaches is inspired by the algorithmic framework of Satisfiability modulo Theories (SMT; [Nieuwenhuis *et al.*, 2006]) and thus leads to hybrid semantics combining non-monotonic aspects of ASP with monotonic ones of CP. This yields an inevitable blind spot, namely, the incapacity of providing defaults for constraint variables (or even leaving them undefined). Such features must be addressed on the ASP side, which brings us back to the aforementioned problem.

We address this dilemma by introducing a new approach that integrates ASP and CP in the uniform semantic framework of the logic of Here-and-There (HT; [Heyting, 1930]), extending the Equilibrium Logic [Pearce, 1997] characterization of ASP to theories with constraint atoms. This puts both ASP and CP on the same semantic footing, being non-monotonic in nature. The new logic of *Here-and-There with constraints*, HT_C for short, is built from variables over associated domains, whose valid valuations are determined by the interpretation of constraint atoms. HT_C is not only a proper generalization of HT, and hence also ASP, but it also tolerates undefined constraint variables and lets them take default values. Moreover, the logic programming fragment of HT_C also subsumes the CASP approach of [Gebser *et al.*, 2009]. Interestingly, the monotonic nature of constraint variables in CASP can be obtained by adding simple axioms, similar to tertium non datur in HT (or choice rules in ASP). Finally, we elaborate upon the fragment of logic programs with linear constraints on integer variables, LC, and introduce directional assignments in rule heads in order to guarantee foundedness in the presence of undefinedness. Furthermore, we develop a translation of LC-programs into CASP that forms the backbone of our implementation by means of off-the-shelf CASP solvers.

2 Here-and-There with Constraints

In this section, we introduce the logic of *Here-and-There with Constraints*, HT_C for short.

We begin by recalling the definition of a *constraint satisfaction problem* as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ a domain of values, and $\mathcal{C}$ a set of constraints. Each constraint is a pair $\langle \overline{x}, R \rangle$ where $\overline{x}$ is an n-tuple of variables and R an n-ary relation on $\mathcal{D}$. A *valuation* of the variables is a function from the set of variables to the domain of values $v : \mathcal{X} \to \mathcal{D}$. A valuation v satisfies a constraint $\langle (x_1, \ldots, x_n), R \rangle$ if $(v(x_1), \ldots, v(x_n)) \in R$. A *solution* of $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a valuation v that satisfies all constraints in $\mathcal{C}$.

The syntax of HT_C starts from a similar signature $\langle \mathcal{X}, \mathcal{D}, \mathcal{A} \rangle$ where, as before, $\mathcal{X}$ are variables and $\mathcal{D}$ domain values, but $\mathcal{A}$ are now *constraint atoms*, or just *atoms* for short. The syntax of a constraint atom is left open and depends on the respective type of constraints. We assume that it always has a set of associated variables from $\mathcal{X}$ and sometimes refers to elements in $\mathcal{D}$. Examples of constraint atoms are $x + y \leq 3$, $x = y$, $\mathit{all\_diff}(\{x, y, z\})$, or $x \in \{\boldsymbol{t}, \boldsymbol{f}\}$ where x, y, z are variables and $3, \boldsymbol{t}, \boldsymbol{f}$ are values. We sometimes refer to a subset of variables $\mathcal{P} \subseteq \mathcal{X}$ as *propositions* and let the subset $\{\boldsymbol{t}, \boldsymbol{f}\} \subseteq \mathcal{D}$ of values stand for Boolean truth values. Also, for each proposition $p \in \mathcal{P}$, we include a constraint atom $(p = \boldsymbol{t}) \in \mathcal{A}$ that we call *regular atom* and usually abbreviate by p. We also allow for atoms of form $(p = \boldsymbol{f}) \in \mathcal{A}$, standing for the *strong negation* of p, and alternatively write them as $\sim p$. A *formula* is any propositional combination of atoms and logical connectives $\bot, \wedge, \vee, \rightarrow$.

We define negation as $\neg\varphi \stackrel{\text{def}}{=} \varphi \rightarrow \bot$ and double implication as $\varphi \leftrightarrow \psi \stackrel{\text{def}}{=} (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

The semantics of HT_C is defined as follows. A *partial valuation* v is a function $v : \mathcal{X} \rightarrow \mathcal{D} \cup \{\boldsymbol{u}\}$ assigning to each variable in $\mathcal{X}$ either a domain value from $\mathcal{D}$ or the special value $\boldsymbol{u} \notin \mathcal{D}$ standing for "undefined." A partial valuation can be alternatively represented as a set $v \subseteq \mathcal{X} \times \mathcal{D}$ that does not contain two different pairs (x, c) and (x, d) with $c \neq d$ for the same variable x and that does not include any pair $(x, \cdot)$ when $v(x) = \boldsymbol{u}$. With this set representation, we use the standard $\subseteq$ relation to compare partial evaluations. We define $\mathcal{V}_{\mathcal{X},\mathcal{D}}$ as the set of all possible partial valuations for $\mathcal{X}$ and $\mathcal{D}$ and remove the subindices when clear from the context. We use letters v, v' to denote elements of $\mathcal{V}$. The closure of a set $\mathcal{S}$ of partial valuations is defined as $\mathcal{S}^{\uparrow} \stackrel{\text{def}}{=} \{v \mid v \supseteq v', v' \in \mathcal{S}\}$. A set $\mathcal{S}$ of partial valuations is said to be *closed* if $\mathcal{S}^{\uparrow} = \mathcal{S}$. Given two variables $x, y \in \mathcal{X}$ and a domain $\mathcal{D} = \{1, 2\}$, an example of a non-closed set of valuations is $\{v \in \mathcal{V} \mid v(x) = v(y)\}$ since $v = \emptyset$ satisfies $v(x) = v(y) = \boldsymbol{u}$ but its superset $v' = \{(x, 1), (y, 2)\}$ does not satisfy $v'(x) = v'(y)$. On the other hand, $\{v \in \mathcal{V} \mid v(x) = v(y) \neq \boldsymbol{u}\}$ is closed.

Satisfaction of formulas is defined wrt a fixed function, called *atom denotation*, $[\![\cdot]\!] : \mathcal{A} \rightarrow 2^{\mathcal{V}}$ that maps each atom to a closed set of partial valuations. As examples of atom denotations, we have the following:

$$
\begin{aligned}
[\![\, |x - y| = d \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid v(x), v(y) \in \mathbb{Z}, \\
&\qquad |v(x) - v(y)| = d\} \\
[\![\, x = y \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid v(x) = v(y) \neq \boldsymbol{u}\} \\
[\![\, x \neq y \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid \boldsymbol{u} \neq v(x) \neq v(y) \neq \boldsymbol{u}\} \\
[\![\, \mathit{all\_diff}(X) \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid \text{for any pair } x, y \in X \\
&\qquad \boldsymbol{u} \neq v(x) \neq v(y) \neq \boldsymbol{u}\} \\
[\![\, x \in D \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid v(x) \in D\} \\
[\![\, \mathit{some\_zero}(x, y) \,]\!] &\stackrel{\text{def}}{=} \{v \in \mathcal{V} \mid v(x) = 0 \text{ or } v(y) = 0\}
\end{aligned}
$$

As mentioned, denotations must be closed. For instance, all above examples satisfy $[\![A]\!]^{\uparrow} = [\![A]\!]$. This property allows us to prove the following result. Given a denotation $[\![\cdot]\!]$ and a partial valuation v, we define the set of atoms that hold in v as $At_{[\![\cdot]\!]}(v) \stackrel{\text{def}}{=} \{A \in \mathcal{A} \mid v \in [\![A]\!]\}$ or $At(v)$ for short.

Proposition 1 *Let $v \subseteq v'$ be a pair of partial valuations and $[\![\cdot]\!]$ a denotation for atoms in $\mathcal{A}$. Then $At(v) \subseteq At(v')$.*

The denotation for regular atoms is fixed as expected:

$$
[\![\, p \,]\!] = [\![\, p = \boldsymbol{t} \,]\!] \stackrel{\text{def}}{=} \{v \mid v(p) = \boldsymbol{t}\}
$$
$$
[\![\, {\sim}p \,]\!] = [\![\, p = \boldsymbol{f} \,]\!] \stackrel{\text{def}}{=} \{v \mid v(p) = \boldsymbol{f}\}
$$

Let $X = \{x_1, \ldots, x_n\}$ be a subset of variables $X \subseteq \mathcal{X}$. A constraint $C = \langle (x_1, \ldots, x_n), R \rangle$ can be understood as the following set of partial valuations:

$$
\{v \mid \{(x_1, d_1), \ldots, (x_n, d_n)\} \subseteq v \text{ and } (d_1, \ldots, d_n) \in R\} \,.
$$

These are all valuations fixing variables in X to some tuple in R while varying the remaining variables $\mathcal{X} \setminus X$ in all possible ways (including being undefined). Notice that $v(x_i)$ is always defined for all variables in X and any valuation v in the constraint. A set of partial valuations is said to be *strict* if it corresponds to some constraint C. Otherwise, it is said to be non-strict. For instance, the denotation $[\![\, x = y \,]\!]$ provided before is strict because we can represent it as a constraint $\langle (x, y), R \rangle$ where $R = \{(d, d) \mid d \in \mathcal{D}\}$. However, $[\![\, \mathit{some\_zero}(x \cdot y) \,]\!]$ is non-strict because we may have $v(x) = 0$ but $v(y)$ undefined or the other way around. This cannot be captured by a constraint: we cannot cover this set of valuations as any $\langle (x, y), R \rangle$, $\langle (x), R \rangle$ or $\langle (y), R \rangle$ because we must include both $v_1 = \{(x, 0)\}$, where y is undefined, and $v_2 = \{(y, 0)\}$, where x is undefined. In this paper, we focus on strict denotations for atoms.

An *interpretation* in HT_C is a pair $\langle H, T \rangle$ of partial valuations such that $H \subseteq T$.

Definition 1 *Given a fixed denotation $[\![\cdot]\!]$, we say that an interpretation $\langle H, T \rangle$* satisfies *a formula φ, written $\langle H, T \rangle \models \varphi$, when the following recursive conditions hold:*

(i) $\langle H, T \rangle \models A$ *iff* $H \in [\![A]\!]$

(ii) $\bot, \vee, \wedge$ *as usual*

(iii) $\langle H, T \rangle \models \varphi \rightarrow \psi$ *iff for both* $v = H$ *and* $v = T$ *it holds:* $\langle v, T \rangle \not\models \varphi$ *or* $\langle v, T \rangle \models \psi$.

As usual, an interpretation $\langle H, T \rangle$ is a *model* of a theory Γ, written $\langle H, T \rangle \models \Gamma$, if it satisfies all formulas in Γ, that is, $\langle H, T \rangle \models \varphi$ for all $\varphi \in \Gamma$. A theory (or a single formula) Γ *entails* a formula φ, written $\Gamma \models \varphi$, when all models of Γ are models of φ. We write $\varphi \equiv \psi$ to represent that φ and ψ are *equivalent*, that is, have the same HT_C models.

Observation 1 *The logic of Here-and-There can be obtained as a case of HT_C with a signature $\langle \mathcal{X}, \mathcal{D}, \mathcal{A} \rangle$ where $\mathcal{X}$ represents propositions, $\mathcal{D} = \{\boldsymbol{t}\}$ and $\mathcal{A} = \mathcal{X}$, understanding each $p \in \mathcal{A}$ as an abbreviation of the constraint atom $(p = \boldsymbol{t})$ as explained above. Moreover, this can be generalized to any arbitrary singleton $\mathcal{D} = \{d\}$ and corresponding constraint atoms $(p = d)$ and the relationship still holds.*

The following is an interesting connection between HT_C and HT:

Proposition 2 *Let Γ be some HT_C theory for signature $\langle \mathcal{X}, \mathcal{D}, \mathcal{A} \rangle$ and let $\langle H, T \rangle$ be some model of Γ. Then, $\langle At(H), At(T) \rangle$ is an HT model of Γ under signature $\langle \mathcal{A}, \{\boldsymbol{t}\}, \{p = \boldsymbol{t} \mid p \in \mathcal{A}\} \rangle$.*[1]

As a result, we directly derive these properties from HT:

Proposition 3 *For any formula φ:*

- $\langle H, T \rangle \models \varphi$ *implies* $\langle T, T \rangle \models \varphi$
- $\langle H, T \rangle \models \neg\varphi$ *iff* $\langle T, T \rangle \not\models \varphi$
- *Any tautology in HT is also a tautology in HT_C.*

In the light of Proposition 2, one might wonder whether, to capture HT_C semantics, it would suffice to exclusively consider HT theories built with constraint atoms used as propositional variables without entering into their internal semantics. This is not the case, since we cannot obtain a similar correspondence in the opposite direction. Namely, not any arbitrarily chosen pair of sets of atoms $H' \subseteq T' \subseteq \mathcal{A}$ necessarily corresponds to an HT_C interpretation $\langle H, T \rangle$ such

[1] Or simply $\langle \mathcal{A}, \{\boldsymbol{t}\}, \mathcal{A} \rangle$ by abbreviating all $(p = \boldsymbol{t})$ by p.

that $H' = At(H)$ and $T' = At(T)$. As an example, take $H' = \{x = y\} \subset \{x = y, x = 0\} = T'$. Clearly, to obtain $At(T) = T'$ we need $T = \{(x, 0), (y, 0)\}$. Now, the only subset of T that satisfies $x = y$ is $H = T$ itself. But then $At(H) = At(T) = T' \neq H'$.

We sometimes write $T \models \varphi$ to stand for $\langle T, T\rangle \models \varphi$. Extending the equilibrium model definition [Pearce, 1997] to HT_C theories is straightforward.

Definition 2 *An 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 $\langle H, T\rangle \models \Gamma$.*

In this case, we also say that T is a *stable model* of Γ. Again, if we restrict the signature to $\langle \mathcal{A}, \{\boldsymbol{t}\}, \mathcal{A}\rangle$, we obtain standard equilibrium/stable models.

For logic programming syntax, we use comma ',' and semicolon ';' as alternative representations of $\wedge$ and $\vee$, respectively. Similarly, we write $\varphi \leftarrow \psi$ to stand for $\psi \rightarrow \varphi$, as expected. An HT_C-literal is an atom A or its default negation $\neg A$. An HT_C program is a set of rules of the form:

$$L_1; \ldots; L_n \leftarrow L_{n+1}, \ldots, L_m$$

where each L_i is an HT_C-literal.

Example 1 *For solving the 8-queens puzzle, we define the variables $\mathcal{X} = \{q_1, \ldots, q_8\}$ where q_i represents the column of the queen located at row i. We are given some queens already placed and, by default, the first queen should be located at column 1. A possible way to encode this problem is as follows. We use the domain values $\mathcal{D} = \{1, \ldots, 8\}$ and use the atoms $all\_diff(X)$ and $|x - y| = d$ as given above. Then, we specify the problem as the HT_C-program Π_1:*

$$\bot \leftarrow \neg all\_diff(\mathcal{X}) \quad (1)$$

$$\bot \leftarrow |q_i - q_j| = d_{i,j} \quad (2)$$

$$q_1 = 1 \leftarrow \neg(q_1 \neq 1) \quad (3)$$

$$q_k \in \mathcal{D} \quad (4)$$

where $i, j, k \in \mathcal{D}$, $i \neq j$, $k > 1$ and $d_{i,j}$ is the constant $|i - j|$.

Without further information, the program Π_1 yields four solutions corresponding to the possible 8-queens arrangements with $q_1 = 1$. However, if we add the fact $q_1 = 4$, we obtain the 18 possible solutions where queen 1 is located at row 4.

As we can see, constraints can be used to encode default reasoning, such as the default value 1 for variable q_1 in the example. This feature of HT_C cannot be represented with the usual semantics for CASP [Gebser *et al.*, 2009] which separates regular ASP atoms (that allow for defaults) from constraint atoms, that only permit monotonic reasoning. As a result, any CASP program that does not contain regular atoms is monotonic. Note the difference wrt HT_C where, due to Observation 1, it is always possible to encode any standard logic program only using constraint variables and picking an arbitrary singleton domain. For instance, the ASP program $\{p \leftarrow \neg q\}$ can be directly encoded as $\{p = 1 \leftarrow \neg(q = 1)\}$ using integer variables instead of Boolean atoms.

Capturing CASP constraints in HT_C can be easily achieved. Take the following pair of axioms:

$$\neg\neg(x = x) \quad (5)$$

$$x = x \vee \neg(x = x) \quad (6)$$

A variable x is said to be *defined* (resp. *rigid*) in a theory Γ if the axiom (5) (resp. (6)) is entailed by Γ.

Proposition 4 *For any model $\langle H, T\rangle$ of Γ:*

- *If x is defined in Γ, then $T(x) \neq \boldsymbol{u}$*
- *If x is rigid in Γ, then $H(x) = T(x)$*

Intuitively, (5) acts as a constraint forbidding stable models with x undefined. However, x can be undefined in H, that is, during models minimization. Thus, a defined variable may be assigned a default value, as we did with q_1 in Example 1. On the other hand, (6) forces a monotonic behavior for x, so that we can freely choose its value beforehand, including the case in which we decide to leave it undefined. When a variable is both defined and rigid it satisfies (5) and (6), whose conjunction amounts to the axiom $x = x$. This axiom acts as a choice rule allowing to pick any arbitrary value in $\mathcal{D}$ for x.

Theorem 1 *The definition of CASP provided in [Gebser* et al.*, 2009] exactly corresponds to HT_C programs where all variables are defined and rigid.*

In fact, we can apply this same technique (adding axiom $x = x$) to selectively fix a CASP behavior only for some variable x. This is analogous to the addition of the ASP choice[2] $p \vee \neg p$ to make proposition p behave classically.

3 Logic programs with Linear Constraints

In this section, we focus on a family of constraint atoms for dealing with linear constraints on integer variables, studying some useful syntactic constructions for logic programs with this kind of atoms. A *linear constraint* is a constraint atom of the form $\alpha \leq \beta$ where α and β are in their turn linear expressions defined as follows. A *linear expression* α is a sum $t_1 + \cdots + t_n$ where each addend t_i can be a product $d_i \cdot x_i$ or simply a constant d_i, being $d_i \in \mathbb{Z}$ and $x_i \in \mathcal{X}$. By $Vars(\alpha)$ we denote the set of variables occurring in α and we sometimes write $Vars(\alpha, \beta) \stackrel{\text{def}}{=} Vars(\alpha) \cup Vars(\beta)$ when dealing with two linear expressions. 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_i instead of $1 \cdot x_i$ and we directly replace the '$+$' symbol by '$-$' for negative constants. Moreover, when clear from the context, we sometimes omit the '$\cdot$' symbol. As an example, $-x+3\,y-2\,z$ stands for $(-1) \cdot x + 3 \cdot y + (-2) \cdot z$. Other abbreviations must be handled with care. In particular, we neither remove products of form $0 \cdot x$ nor replace them by 0.

To define the denotation of a linear constraint, we extend any partial valuation v on integer variables to any arbitrary arithmetic term t in the following way:

$$v(d) \stackrel{\text{def}}{=} d \quad \text{if } d \in \mathbb{Z}$$

$$v(x) \stackrel{\text{def}}{=} \begin{cases} d & \text{if } (x, d) \in v, d \in \mathbb{Z} \\ \boldsymbol{u} & \text{otherwise} \end{cases}$$

$$v(t_1 \oplus t_2) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{u} & \text{if } v(t_1) = \boldsymbol{u} \text{ or } v(t_2) = \boldsymbol{u} \\ v(t_1) \oplus v(t_2) & \text{otherwise} \end{cases}$$

for any variable $x \in \mathcal{X}$ and any operator $\oplus \in \{\cdot, +\}$. As before, we write $v(t) = \boldsymbol{u}$ to represent that $v(t)$ is undefined.

[2] This HT-formula is frequently denoted as $\{p\}$ in ASP.

In other words, an arithmetic expression is evaluated as usual, except that it is undefined if it contains some undefined subterm (or eventually, some undefined variable).

Proposition 5 *For any arithmetic expression t and HT_C interpretation $\langle H, T\rangle$, $H(t) \neq \boldsymbol{u}$ implies $H(t) = T(t)$.*

The denotation of a linear constraint $\alpha \leq \beta$ is defined as:

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

This collects interpretations assigning some integer both to α and β, and additionally $v(\alpha) \leq v(\beta)$. Therefore, $\alpha \leq \beta$ does not hold in interpretations where some variable in $Vars(\alpha, \beta)$ is undefined (or assigned a non-integer value). We can also observe that $[\![\, \alpha \leq \beta \,]\!]$ is strict, since it can be represented as the constraint $\langle (x_1, \ldots, x_n), R\rangle$ with $Vars(\alpha, \beta) = \{x_1, \ldots, x_n\}$ and R containing all the n-tuples of integer values that assigned to the variables fulfill $v(\alpha) \leq v(\beta)$.

We use some abbreviations: we write $\alpha = \beta$ to stand for the conjunction[3] $\alpha \leq \beta \wedge \beta \leq \alpha$. Given an inequality $A : (\alpha \leq \beta)$, we write $\overline{A}$ to stand for $(\beta < \alpha) \stackrel{\text{def}}{=} \beta \leq \alpha \wedge \neg(\alpha \leq \beta)$. We also define the formula $\alpha \neq \beta$ as $\alpha < \beta \vee \beta < \alpha$. 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.

One interesting result is that we can fully capture propositional HT and equilibrium logic in HT_C with integer variables and linear constraints. To do so, it suffices to replace each occurrence of a Boolean variable p in a propositional HT theory by the constraint $x_p = 1$ for a corresponding integer variable x_p. Then, we get an obvious one-to-one mapping where each assignment $(p, \boldsymbol{t})$ in an HT model corresponds to $(x_p, 1)$ in HT_C and vice versa.

For any linear expression α, we define $df\,\alpha \stackrel{\text{def}}{=} \alpha \leq \alpha$ to stand for "α is defined," that is, α has a value. It is easy to see that $df\,\alpha$ is equivalent to the conjunction $\bigwedge_{x \in Vars(\alpha)} df\,x$. Therefore, if α does not contain integer variables, $df\,\alpha = \top$.

Constraints in rule heads must be handled with care because they treat all variables, in principle, in a non-directional way. For instance, imagine we want to assign to x some value in the range from 0 to y, and that we have the rule $y = 10 \leftarrow p$ but currently no evidence about p so y should be undefined. Adding the formula $0 \leq x \wedge x \leq y$ would not yield the desired effect because, as we force both constraints to be true, it would also allow for justifying *any arbitrary value* for y. To allow for directional assignments, we introduce the following construction. An *assignment* A for variable x is an expression of the form $x := \alpha \,..\, \beta$ (with α, β linear expressions) standing for the formula:

$$\neg\neg df\,A \wedge (df\,A \rightarrow \alpha \leq x \wedge x \leq \beta) \tag{7}$$

where $df\,A \stackrel{\text{def}}{=} df\,\alpha \wedge df\,\beta$. We say that A is *applicable* in $\langle H, T\rangle$ when $\langle H, T\rangle \models df\,A$. We define $\Phi(A)$ to be the non-directional version of assignment A, that is, $\Phi(x := \alpha \,..\, \beta) \stackrel{\text{def}}{=} \alpha \leq x \wedge x \leq \beta$. As we can see, A makes some additional checks regarding the definedness of α and β before imposing any condition on x. In particular, $(df\,A \rightarrow \alpha \leq x \wedge x \leq \beta)$ 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, $\neg\neg df\,A$ 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 $(x := \alpha) \stackrel{\text{def}}{=} (x := \alpha \,..\, \alpha)$, that is, $\neg\neg df\,\alpha \wedge (df\,\alpha \rightarrow x = \alpha)$. Note that, as a result, $\Phi(x := \alpha) = (x = \alpha)$.

The following proposition relates A and its non-directional version, $\Phi(A)$, in some particular cases.

Proposition 6 *Given an assignment $A = (x := \alpha \,..\, \beta)$ then:*

(i) $A \wedge df\,A \equiv \Phi(A)$

(ii) $\neg A \equiv \neg\Phi(A)$

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

We now define an interesting syntactic subclass of HT_C logic programs. A *linear constraint rule*, or LC-rule for short, is a rule of the form:

$$A_1; \ldots; A_n \leftarrow B_1, \ldots, B_m, \neg B_{m+1}, \ldots, \neg B_k \tag{8}$$

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 (8), we let $Head(r)$ stand for the set of assignments $\{A_1, \ldots, A_n\}$ and $Body(r)$ be the set of linear constraints $\{B_1, \ldots, B_m, \neg B_{m+1}, \ldots, \neg B_k\}$. An HT_C program consisting of LC-rules only is called *LC-program*.

Notice that an LC-rule does not directly correspond to an HT_C program rule since the assignments in the head contain nested implications like (7). However, the following theorem allows us to rewrite any LC-rule as a set of HT_C rules, and helps us to illustrate the intuitive behavior of assignments:

Theorem 2 *A rule r as in* (8) *is equivalent*[4] *to the conjunction $\bigwedge_{\Delta \subseteq Head(r)} \Psi_\Delta$ where Ψ_Δ is the implication:*

$$\bigvee_{A \in \Delta} \Phi(A) \leftarrow \bigwedge_{A \in Body(r)} A \wedge \bigwedge_{A \in \Delta} df\,A \wedge \bigwedge_{A \in Head(r) \setminus \Delta} \neg\Phi(A)$$

Due to Proposition 6, each implication in Theorem 2 can be written as a set of HT_C rules, because $\Phi(A)$ is a conjunction in the head $\alpha \leq x \wedge x \leq \beta$ and, by De Morgan, $\neg\Phi(A)$ becomes a disjunction in the body $\neg(\alpha \leq x) \vee \neg(x \leq \beta)$, and both cases can be unfolded in HT into different rules. Let us informally illustrate this result with the following example.

Example 2 *The LC-rule*

$$y := x - 1 \leftarrow \neg(1 \leq z) \tag{9}$$

corresponds to the set of HT_C rules:

$$y = x - 1 \leftarrow \neg(1 \leq z), df\,x \tag{10}$$

$$\bot \leftarrow \neg(1 \leq z), \neg(y = x - 1) \tag{11}$$

[3] When we write $x = y$ for two variables, we deal with some syntactic ambiguity wrt '=' used as identity constraint atom. In fact, for integer variables, the semantics of both formulas coincide.

[4] A more succinct translation is used in Section 4.

Suppose our LC-program consists of rule (9) only. The intuition is that $\neg(1 \leq z)$ should hold, as z is undefined and we cannot prove $1 \leq z$, but then $y := x - 1$ cannot be fulfilled, since there are no assignments for x, so it is left undefined and $x-1$ cannot be evaluated. As a result, we get no stable model. Note how, if we replaced $y := x - 1$ by just $y = x - 1$ in the head, we would get a stable model $T = \{(y, d), (x, d - 1)\}$ per each $d \in \mathbb{Z}$ so the rule would also be fixing values for x. Looking at the translation in (10)-(11), the behavior of the assignment becomes clearer. As z does not occur in any head, it is left undefined. Variable x occurs in the head of (10), but it depends on dfx in the body, and so, this rule cannot be used to provide a founded value for x. Thus, x is undefined and $y = x - 1$ is also false, so the constraint (11) becomes applicable, and we get no stable model.

To illustrate non-monotonicity, suppose we add the rule $x := 1$ whose translation from Theorem 2 amounts to the fact $x = 1$. Then, we obtain a unique stable model $\{(x, 1), (y, 0)\}$. Moreover, assume now that together with $x := 1$, we also add the assignment $z := 0..3$. This last version of the program yields four stable models: one with $z = 0$ and $y = 0$ and the other three with y undefined and z varying from 1 to 3.

The next example illustrates the behavior of an LC-rule with a disjunction in the head.

Example 3 *The LC-rule:*

$$z := x; t := y \tag{12}$$

corresponds to the conjunction of the HT_C-rules:

$$z = x; t = y \leftarrow dfx, dfy \tag{13}$$
$$z = x \leftarrow dfx, \neg(t = y) \tag{14}$$
$$t = y \leftarrow dfy, \neg(z = x) \tag{15}$$
$$\bot \leftarrow \neg(z = x), \neg(t = y) \tag{16}$$

If we only have (12) in our program, then x and y are undefined and the rule cannot be satisfied – constraint (16) is applicable. If we add, for instance, the assignment $x := 1$, then y is still undefined, but (14) becomes applicable and we get the stable model $\{(x, 1), (z, 1)\}$. Then, if we further add $y := 2$, we obtain the two expected stable models $\{(x, 1), (y, 2), (z, 1)\}$ and $\{(x, 1), (y, 2), (t, 2)\}$. To illustrate how disjunction interacts with positive cycles, let us look at the following variation.

Example 4 *Take the program containing* (12) *and the rules:*

$$x := 1 \tag{17}$$
$$y := 1 \leftarrow z = 1 \tag{18}$$
$$z := 1 \leftarrow y = 1 \tag{19}$$

If we apply the first disjunct in (12), we get $z = 1$ and then $y = 1$ by (18) leading to the stable model $\{(x, 1), (y, 1), (z, 1)\}$. This is indeed the only stable model of the program. If we tried to apply the second disjunct in (12) instead, we would need to establish a founded value for y first. However, y depends on z which, in its turn, depends on x through the first disjunct of (12). But then the solution $\{(x, 1), (y, 1), (z, 1), (t, 1)\}$ would not be minimal.[5]

[5]As in standard ASP, stable models of a positive HT_C program are always minimal wrt set inclusion.

We show next that LC-programs can be translated into ASP with linear constraints, viz. CASP [Gebser *et al.*, 2009], by introducing some auxiliary propositional variables. CASP semantics was based on the assumption that all constraint variables were defined and rigid, that is, the choice axiom $x = x$ is satisfied for any variable x. Let $\boldsymbol{DF}$ stand for the set of choice axioms $x = x$ for all variables in $\mathcal{X}$.

Proposition 7 *For any linear expression α: $\boldsymbol{DF} \models df\alpha \equiv \top$*

Let Π be an LC-program for signature $\Sigma = \langle \mathcal{X}, \mathbb{Z}, \mathcal{A} \rangle$ where $\mathcal{A}$ is a set of linear constraints. We define a set of auxiliary propositions $\mathcal{P} = \{x^\delta \mid x \in \mathcal{X}\}$. Intuitively, a proposition x^δ represents the fact that variable x has a defined value in the original program Π. The translation of Π gives a new HT_C program $\tau(\Pi)$ for the extended signature $\tau(\Sigma) = \langle \mathcal{X} \cup \mathcal{P}, \mathbb{Z} \cup \{\boldsymbol{t}\}, \mathcal{A} \cup \mathcal{P} \rangle$. For any linear expression α, we write α^δ to stand for the conjunction of all propositions y^δ for all $y \in \mathit{Vars}(\alpha)$ and $(\alpha \leq \beta)^\delta$ to stand for the conjunction $\alpha^\delta \wedge \beta^\delta$. Using this notation, the translation of a linear constraint A is the formula $\tau(A) \stackrel{\text{def}}{=} A \wedge A^\delta$. Intuitively, due to the choice axiom $\boldsymbol{DF}$ (applied only on $\mathcal{X}$), A can hold due to an arbitrary assignment of variable values, but $\mathcal{A}^\delta$ guarantees that all variables have been assigned a founded value wrt program Π. Notice that the translation of $df\alpha$ corresponds to $\tau(df\alpha) = \tau(\alpha \leq \alpha) = \alpha \leq \alpha \wedge \alpha^\delta$ and, under the assumption $\boldsymbol{DF}$, the latter is equivalent to α^δ (Proposition 7). For any arbitrary formula ϕ, $\tau(\phi)$ stands for the replacement of every constraint atom A in ϕ by $\tau(A)$. The translation of an LC-program Π corresponds to the set of formulas $\tau(\Pi) \stackrel{\text{def}}{=} \{\tau(r) \mid r \in \Pi\}$.

As we see below, the models of the translation $\tau(\Pi)$ are isomorphic to the original models of Π. Thus, we can apply τ on rules of the form (8) or on their decomposition through Theorem 2. As an example, if we apply the translation on the decomposition of (12) as (13)-(16), we obtain, after some minor simplifications, the rules:

$$z = x \wedge z^\delta; t = y \wedge t^\delta \leftarrow x^\delta, y^\delta \tag{20}$$
$$z = x \wedge z^\delta \leftarrow x^\delta, \neg(t = y \wedge t^\delta \wedge y^\delta) \tag{21}$$
$$t = y \wedge t^\delta \leftarrow y^\delta, \neg(z = x \wedge z^\delta \wedge x^\delta) \tag{22}$$
$$\bot \leftarrow \neg(z = x \wedge z^\delta \wedge x^\delta), \neg(t = y \wedge t^\delta \wedge y^\delta) \tag{23}$$

Given a valuation v for the extended signature $\tau(\Sigma)$, we define its corresponding "defined" subset in signature Σ as $v|_\delta \stackrel{\text{def}}{=} \{(x, d) \in v \mid (x^\delta, \boldsymbol{t}) \in v\}$.

Proposition 8 *Given a pair of partial valuations $H \subseteq T$ for signature $\tau(\Sigma)$, we have $H|_\delta \subseteq T|_\delta$.*

Lemma 1 *Let $\langle H, T \rangle$ be an HT_C interpretation for signature $\tau(\Sigma)$. Then, for any constraint atom $A \in \mathcal{A}$, $\langle H, T \rangle \models \tau(A)$ iff $\langle H|_\delta, T|_\delta \rangle \models A$.*

By a simple application of structural induction, we get:

Corollary 1 *Let $\langle H, T \rangle$ be an HT_C interpretation for signature $\tau(\Sigma)$. Then, for any formula φ, $\langle H, T \rangle \models \tau(\varphi)$ iff $\langle H|_\delta, T|_\delta \rangle \models \varphi$.*

Theorem 3 (Soundness) *Let T be a stable model of $\tau(\Pi) \cup \boldsymbol{DF}$. Then, $T|_\delta$ is a stable model of LC-program Π.*

Theorem 4 (Completeness) *Let T be a stable model of LC-program Π. Then, any T' such that $T' \models \boldsymbol{DF}$ and $T'|_\delta = T$ is a stable model of $\tau(\Pi) \cup \boldsymbol{DF}$.*

That is, each stable model T of Π is in one-to-one correspondence to a class of stable models T' of $\tau(\Pi) \cup \boldsymbol{DF}$ that coincide with T in the valuation of its defined variables, making x^δ true for all of them, and freely varying the other variables. For instance, the above program Π consisting of (12) plus the facts $x := 1$ and $y := 2$ has two stable models $T_1 = \{(x,1),(y,2),(z,1)\}$ and $T_2 = \{(x,1),(y,2),(t,2)\}$. Then, $\tau(\Pi)$ includes formulas (20)-(23) plus the translation of the facts, viz. $x = 1 \wedge x^\delta$ and $y = 2 \wedge y^\delta$. Also, $\boldsymbol{DF}$ includes axioms $x = x$, $y = y$ and $z = z$. The resulting translation $\tau(\Pi) \wedge \boldsymbol{DF}$ yields two sets of stable models: one of the form $\{(x,1),(y,2),(z,1),(x^\delta,\boldsymbol{t}),(y^\delta,\boldsymbol{t}),(z^\delta,\boldsymbol{t}),(t,d)\}$ and another $\{(x,1),(y,2),(t,2),(x^\delta,\boldsymbol{t}),(y^\delta,\boldsymbol{t}),(t^\delta,\boldsymbol{t}),(z,d)\}$ in both cases for any $d \in \mathcal{D}$.

4 An LC-solver implementation

We implemented our approach (see [LC2CASP, 2016]) as an extension of the CASP solver CLINGCON 3 [Banbara *et al.*, 2016]. Our system computes the stable models of an LC-program by implementing a polynomial-size variant of the translation described in the previous section. This is accomplished by using auxiliary atoms to avoid the exponential blow-up in Theorem 2 (similar to [Tseitin, 1968]).

In the input language, rule heads are formed by means of the functor `&assign`. More precisely, a disjunctive head $A_1; \ldots; A_n$ as in (8) is represented as '`&assign { A1;...;An }`'. Similarly, linear expressions are formed using the `&sum` functor. A linear constraint of form '$\alpha_1 + \cdots + \alpha_n \prec \beta$' is written as '`&sum { a1;...;an } ≺ b`', where $\prec$ is among `<=`, `=`, `>=`, `<`, `>`, or `!=`. Moreover, the language contains an all-different constraint, `&distinct`, as well as a `&show` and `&minimize` directive with the same meaning as in ASP yet applied to linear expressions. As with ASP, undefined variables are not shown (eg. t and z above, respectively); also, they do not contribute to minimization.

For illustration, consider the HT_C-program in (1) to (4) expressed as an LC-program:

```
1  n(1..8).
2  :- not &distinct { q(X) : n(X) }.
3  :- &sum { q(X); -q(Y) } = X-Y, n(X), n(Y), X != Y.
4  :- &sum { q(X); -q(Y) } = Y-X, n(X), n(Y), X != Y.
5  &assign { q(1) := 1 } :- not &sum { q(1) } != 1.
6  &assign { q(X) := 1..n } :- n(X), X > 1.
```

Note that atoms, like `n(X)` and `X > 1`, are Boolean propositions, mixed with constraint atoms. The above LC-program has 4 stable models, all assigning `1` to `q(1)` according to the default expressed in Line 5. However, once '`&assign { q(1) := 4 }.`' is added, the default is overwritten, and we obtain 18 models, yet all assigning `4` to `q(1)`.

5 Discussion

We introduced the logic HT_C in order to capture constraint theories in the non-monotonic setting known from ASP. As a result, HT_C allows for assigning default values to constraint variables or to leave them undefined. To the best of our knowledge, HT_C constitutes the first logical account of non-monotonic constraint theories. Since HT and thus also ASP constitute special cases of HT_C, we obtain a uniform framework integrating ASP and CP on the same semantic footing. In view of this, we particularly elaborated on the HT_C fragment of LC-programs dealing with linear constraints on integer variables. A central concept is that of assignments (in rule heads) because they are the only way to attribute values to constraint variables – unassigned variables stay undefined.

Our approach is different from traditional CASP [Baselice *et al.*, 2005; Balduccini, 2009; Gebser *et al.*, 2009], where logic programs are hybridized with constraint atoms having standard monotonic CP semantics. In such approaches, constraint atoms in rule heads are merely shortcuts for the complementary body literal. Rather, the monotonic CP semantics assigns each variable all feasible values. In fact, we have identified the HT_C fragment corresponding to the approach of [Gebser *et al.*, 2009] and pinpointed the axioms characterizing the aforementioned feature. Although we have not proven it, the result should also extend in a slightly different form to the approaches in [Baselice *et al.*, 2005; Balduccini, 2009] due to their close correspondence to [Gebser *et al.*, 2009] established in [Lierler, 2014]. A noteworthy exception among CASP approaches is Bound Founded ASP [Aziz *et al.*, 2013] that imports the notion of non-circular value derivations into CP. Informally, constraints can have a distinguished variable (akin to a head) over a totally ordered domain. The singular value of a lower-bound[6] variable is the smallest derivable value or the smallest domain element. This yields also a non-monotonic approach that comprises ASP as a special case. However, it remains future work to identify the fragment of HT_C that captures this approach and its notion of value minimization.

Our semantics captures a fragment of ASP with partial functions [Cabalar, 2011; Balduccini, 2012] where constraint variables correspond to 0-ary functions. This fragment is expressive enough to cover the general case, since arbitrarily nested partial functions can be reduced to the 0-ary case by a process called *flattening* [Cabalar, 2011] or *unfolding* [Bartholomew and Lee, 2013]. Moreover, our approach extends functional ASP by generalizing equality among terms to arbitrary relations. In this paper, we have focused on linear constraints, but other extensions will be studied in the future.

For implementing the fragment of LC-programs, we have devised a translation into CASP programs in accord with [Gebser *et al.*, 2009] and shown its soundness and completeness. The key role in this translation is played by propositions warranting the non-circularity of constraint assignments. Although our system uses CLINGCON as back-end, our translational approach applies also to other CASP solvers sharing the same semantics. Our system along with several examples and additional material is available at [LC2CASP, 2016].

[6] And analogously for upper-bound variables.

Acknowledgments This work was partially funded by DFG grant SCHA 550/9.

References

[Aziz *et al.*, 2013] R. Aziz, G. Chu, and P. Stuckey. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming*, 13(4-5):517–532, 2013.

[Balduccini, 2009] M. Balduccini. Representing constraint satisfaction problems in answer set programming. In W. Faber and J. Lee, editors, *Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, pages 16–30, 2009.

[Balduccini, 2012] M. Balduccini. A "conservative" approach to extending answer set programming with non-herbrand functions. In E. Erdem, J. Lee, Y. Lierler, and D. Pearce, editors, *Correct Reasoning: Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 24–39. Springer-Verlag, 2012.

[Banbara *et al.*, 2016] M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Submitted for publication*, 2016.

[Bartholomew and Lee, 2013] M. Bartholomew and J. Lee. On the stable model semantics for intensional functions. *Theory and Practice of Logic Programming*, 13(4-5):863–876, 2013.

[Baselice *et al.*, 2005] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

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

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

[Gebser *et al.*, 2009] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In P. Hill and D. Warren, editors, *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*, pages 235–249. Springer-Verlag, 2009.

[Heyting, 1930] A. Heyting. Die formalen Regeln der intuitionistischen Logik. In *Sitzungsberichte der Preussischen Akademie der Wissenschaften*, page 42–56. 1930. Reprint in Logik-Texte: Kommentierte Auswahl zur Geschichte der Modernen Logik, Akademie-Verlag, 1986.

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

[Lifschitz, 2008] V. Lifschitz. What is answer set programming? In D. Fox and C. Gomes, editors, *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 1594–1597. AAAI Press, 2008.

[Nieuwenhuis *et al.*, 2006] 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.

[Pearce, 1997] D. Pearce. A new logical characterisation of stable models and answer sets. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Proceedings of the Sixth International Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*, volume 1216 of *Lecture Notes in Computer Science*, pages 57–70. Springer-Verlag, 1997.

[LC2CASP, 2016] `http://www.cs.uni-potsdam.de/lc2casp`, 2016.

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