An ASP Semantics for Default Reasoning with Constraints

Pedro Cabalar\textsuperscript{1}  Roland Kaminski\textsuperscript{2}  Max Ostrowski\textsuperscript{2}  Torsten Schaub\textsuperscript{2,3}

\textsuperscript{1}University of Corunna, Spain \textsuperscript{2}University of Potsdam, Germany \textsuperscript{3}INRIA, France

1 Introduction

Although Answer Set Programming (ASP; \cite{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 Programming (CP; \cite{Dechter:2003}), leading to the subarea of Constraint ASP (CASP; \cite{Lierler:2014}). In fact, the design of most CASP approaches is inspired by the algorithmic framework of Satisfiability modulo Theories (SMT; \cite{Nieuwenhuis: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; \cite{Heyting:1930}), extending the Equilibrium Logic \cite{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 \textit{Here-and-There with constraints}, \textit{HT}_C for short, is built from variables over associated domains, whose valid valuations are determined by the interpretation of constraint atoms. \textit{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 \textit{HT}_C also subsumes the CASP approach of \cite{Gebser: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, \textit{LC}, and introduce directional assignments in rule heads in order to guarantee groundedness in the presence of undefinedness. Furthermore, we develop a translation of \textit{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 \textit{Here-and-There with Constraints}, \textit{HT}_C for short.

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

The syntax of \textit{HT}_C starts from a similar signature \(\langle \mathcal{X}, D, \mathcal{A} \rangle\) where, as before, \(\mathcal{X}\) are variables and \(D\) 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 \(D\). Examples of constraint atoms are \(x + y \leq 3\), \(x = y\), \(\text{all}\diff\{x, y, z\}\), or \(x \in \{t, f\}\) where \(x, y, z\) are variables and \(3, t, f\) are values. We sometimes refer to a subset of variables \(\mathcal{P} \subseteq \mathcal{X}\) as propositions and let the subset \(\{t, f\} \subseteq D\) of values stand for Boolean truth values. Also, for each proposition \(p \in \mathcal{P}\), we include a constraint atom \(p = t\) of \(\mathcal{A}\) that we call \textit{regular atom} and usually abbreviate by \(p\). We also allow for atoms of form \((p = f) \in \mathcal{A}\), standing for the \textit{strong negation} of \(p\), and alternatively write them as \(\neg p\). A formula is any propositional combination of atoms and logical connectives \(\bot, \land, \lor, \rightarrow, \neg\).
We define negation as \( \neg \varphi \overset{\text{def}}{=} \varphi \rightarrow \bot \) and double implication as \( \varphi \leftrightarrow \psi \overset{\text{def}}{=} (\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi) \).

The semantics of \( HT_C \) is defined as follows. A partial valuation \( v \) is a function \( v : X \rightarrow D \cup \{u\} \) assigning to each variable in \( X \) either a domain value from \( D \) or the special value \( u \notin D \) standing for “undefined.” A partial valuation can be alternatively represented as a set \( v \subseteq X \times D \) 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 \( (\cdot, \cdot) \) when \( v(x) = u \). With this set representation, we use the standard \( \subseteq \) relation to compare partial evaluations. We define satisfaction of formulas is defined w.r.t. a fixed function, \( \cdot \). Proposition 1: Let \( \cdot \) be a subset of variables \( \subseteq \). Then \( \cdot = \cdot \). Proposition 2: A theory (or a single formula) \( \Gamma \) is a tautology if it satisfies all formulas in \( \Gamma \), and \( \Gamma \) entails a formula \( \varphi \), written \( \Gamma \models \varphi \), when all models of \( \Gamma \) are models of \( \varphi \). We write \( \varphi \equiv \psi \) to represent that \( \varphi \) and \( \psi \) are equivalent, that is, have the same \( HT_C \) models.

As mentioned, denotations must be closed. For instance, all above examples satisfy \( [A] = [A] \). 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 \( At_1([v]) = \{A \in A \mid v \in [A]\} \) or \( At_1(v) \) for short.

Let \( v \subseteq v' \) be a pair of partial valuations and \( [\cdot] \) a denotation for atoms. Then \( At_1(v) \subseteq At_1(v') \).

The denotation for regular atoms is fixed as expected:

\[
\begin{align*}
[x-y] & = \{v \in V \mid v(x), v(y) \in Z, (v(x) - v(y)) = d\} \\
[x = y] & = \{v \in V \mid v(x) = v(y) \neq u\} \\
[x \neq y] & = \{v \in V \mid u \neq v(x) \neq v(y) \neq u\} \\
[\text{all_diff}(X)] & = \{v \in V \mid \text{for any pair } x, y \in X \} \\
[u \neq v(x) \neq v(y) \neq u] & = \{v \in D \mid v \in V \mid v(x) \in D\} \\
[some\_zero(x, y)] & = \{v \in V \mid v(x) = 0 \text{ or } v(y) = 0\}
\end{align*}
\]

As mentioned, denotations must be closed. For instance, all above examples satisfy \( [A] = [A] \). 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 \( At_1([v]) = \{A \in A \mid v \in [A]\} \) or \( At_1(v) \) for short.

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

As usual, an interpretation \( \langle H, T \rangle \) is a model of a theory \( \Gamma \), written \( \langle H, T \rangle \models \Gamma \), if it satisfies all formulas in \( \Gamma \), that is, \( \langle H, T \rangle \models \varphi \) for all \( \varphi \in \Gamma \). A theory (or a single formula) \( \Gamma \) entails a formula \( \varphi \), written \( \Gamma \models \varphi \), when all models of \( \Gamma \) are models of \( \varphi \). We write \( \varphi \equiv \psi \) to represent that \( \varphi \) and \( \psi \) 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 \( \langle X, D, A \rangle \) where \( X \) represents propositions, \( D = \{t\} \) and \( A = X \), understanding each \( p \in A \) as an abbreviation of the constraint atom \( (p = t) \) as explained above. Moreover, this can be generalized to any arbitrary singleton \( D = \{d\} \) 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 \( \Gamma \) be some \( HT_C \) theory for signature \( \langle X, D, A \rangle \) and let \( \langle H, T \rangle \) be some model of \( \Gamma \). Then, \( \langle At(H), At(T) \rangle \) is an \( HT \) model of \( \Gamma \) under signature \( \langle A, \{t\}, \{p = t \mid p \in A\}\rangle \).

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

Proposition 3 For any formula \( \varphi \):

- \( \langle H, T \rangle \models \varphi \) implies \( \langle T, T \rangle \models \varphi \)
- \( \langle H, T \rangle \not\models \neg \varphi \iff \langle T, T \rangle \models \varphi \)
- 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 \( H' \subseteq T' \subseteq A \) necessarily corresponds to an \( HT_C \) interpretation \( \langle H, T \rangle \) such

\(^1\)Or simply \( \langle A, \{t\}, A \rangle \) by abbreviating all \( (p = t) \) by \( p \).
that $H' = At(H)$ and $T' = At(T)$. As an example, take $H' = \{x = y\} \subseteq \{x = y, x = 0\} = T'$. Clearly, to obtain $At(H) = T'$ 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 $HTC$ theories is straightforward.

**Definition 2** An interpretation $\langle T, T \rangle$ is an equilibrium model of a theory $\Gamma$ 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 $\Gamma$. Again, if we restrict the signature to $\langle A, \{\}, A \rangle$, we obtain standard equilibrium/stable models.

For logic programming syntax, we use commas ‘,’ and semicolon ‘;’ as alternative representations of $\land$ and $\lor$, respectively. Similarly, we write $\varphi \iff \psi$ to stand for $\varphi \rightarrow \psi$ as expected. An $HTC$-literal is an atom $\alpha$ or its default negation $\neg \alpha$. An $HTC$ program is a set of rules of the form:

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

where each $L_i$ is an $HTC$-literal.

**Example 1** For solving the 8-queens puzzle, we define the variables $X = \{q_1, \ldots, q_8\}$ 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 $D = \{1, \ldots, 8\}$ and use the atoms $\text{all}\_\text{diff}(X)$ and $|x - y| = d$ as given above. Then, we specify the problem as the $HTC$-program $\Pi_1$:

$$\bot \leftarrow \neg \text{all}\_\text{diff}(X) \iff (1)$$

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

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

$$q_k \in D \iff (4)$$

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

Without further information, the program $\Pi_1$ 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 $HTC$ 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 $HTC$ 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 $HTC$ can be easily achieved. Take the following pair of axioms:

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

$$x = x \lor \neg (x = x) \iff (6)$$

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

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

- If $x$ is defined in $\Gamma$, then $T(x) \neq u$.
- If $x$ is rigid in $\Gamma$, 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 $D$ for $x$.

**Theorem 1** The definition of CASP provided in [Gebser et al., 2009] exactly corresponds to $HTC$ 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 $p \lor \neg p$ 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 $\alpha$ and $\beta$ are in their turn linear expressions defined as follows. A linear expression $\alpha$ 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 $x_i \in X$. By $\text{Vars}(\alpha)$ we denote the set of variables occurring in $\alpha$ and we sometimes write $\text{Vars}(\alpha, \beta) \triangleq \text{Vars}(\alpha) \cup \text{Vars}(\beta)$ 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 ‘·’ symbol. As an example, $-x+3y-2z$ 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) \triangleq d \quad \text{if } d \in \mathbb{Z} \$$

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

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

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

---

2This $HT$-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 \( HT \) interpretation \((H,T)\), \( H(t) \neq a \) implies \( H(t) = T(t) \).

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

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

This collects interpretations assigning some integer both to \( \alpha \) and \( \beta \), and additionally \( v(\alpha) \leq v(\beta) \). Therefore, \( \alpha \leq \beta \) does not hold in interpretations where some variable in \( \text{Vars}(\alpha, \beta) \) is undefined (or assigned a non-integer value). We can also observe that \( \llbracket \alpha \leq \beta \rrbracket \) is strict, since it can be represented as the constraint \( \{(x_1, \ldots, x_n), R \} \) with \( \text{Vars}(\alpha, \beta) = \{x_1, \ldots, x_n\} \) 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\(^3\) \( \alpha \leq \beta \land \beta \leq \alpha \). Given an inequality \( A : (\alpha \leq \beta) \), we write \( \bar{A} \) to stand for \( (\beta \leq \alpha) \overset{\text{def}}{=} \beta \land \neg (\alpha \leq \beta) \). We also define the formula \( \alpha \neq \beta \) as \( \alpha < \beta \lor \beta < \alpha \). Notice that \( \alpha \neq \beta \) is stronger than \( \neg (\alpha = \beta) \) since the former requires \( \alpha \) and \( \beta \) 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 \( HT \) 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 \( \langle p, t \rangle \) in an \( HT \) model corresponds to \( \langle x_p, 1 \rangle \) in \( HT \) and vice versa.

For any linear expression \( \alpha \), we define \( df(\alpha) \overset{\text{def}}{=} \alpha \leq \alpha \) to stand for "\( \alpha \) is defined," that is, \( \alpha \) has a value. It is easy to see that \( df(\alpha) \) is equivalent to the conjunction \( \bigwedge_{x \in \text{Vars}(\alpha)} df(x) \).

Therefore, if \( \alpha \) does not contain integer variables, \( df(\alpha) \equiv \top \).

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 \( 0 \leq x \land x \leq y \) 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 \( x := \alpha \ldots \beta \) (with \( \alpha, \beta \) linear expressions) standing for the formula:

\[
\neg df(A) \land (df(A) \rightarrow \alpha \leq x \land x \leq \beta)
\]

(7)

where \( df(A) \overset{\text{def}}{=} df(\alpha) \land df(\beta) \). We say that \( A \) is applicable in \((H,T)\) when \((H,T) \models df(A) \). We define \( \Phi(\alpha) \) to be the non-directional version of assignment \( A \), that is, \( \Phi(\alpha) \overset{\text{def}}{=} \alpha \leq x \land x \leq \beta \). As we can see, \( A \)

makes some additional checks regarding the definedness of \( \alpha \) and \( \beta \) before imposing any condition on \( x \). In particular, \( (df(A) \rightarrow \alpha \leq x \land x \leq \beta) \) guarantees that \( \alpha \) and \( \beta \) can be used to fix the value of \( x \), but not of variables in \( \alpha \) and \( \beta \) themselves. On the other hand, \( \neg df(A) \) can be seen as a constraint checking that \( \alpha \) and \( \beta \) 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 \( (x := \alpha) \overset{\text{def}}{=} (x := \alpha \ldots \alpha) \), that is, \( \neg df(A) \land (df(A) \rightarrow x = \alpha) \).

Note that, as a result, \( \Phi(x := \alpha) = (x = \alpha) \).

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

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

(i) \( A \land df(A) \equiv \Phi(A) \)

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

In particular, if \( A = (x := \alpha \ldots \beta) \) 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 \( HT \) logic programs. A linear constraint rule, or \( LC \)-rule for short, is a rule of the form:

\[
A_1; \ldots; A_n \leftarrow B_1, \ldots, B_m, \neg B_{m+1}, \ldots, \neg B_k
\]

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

Notice that an \( LC \)-rule does not directly correspond to a \( HT \) 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 \( HT \) 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 \( \bigwedge_{\Delta \subseteq \text{Head}(r)} \Phi(\Delta) \) where \( \Phi(\Delta) \) is the implication:

\[
\bigwedge_{\Delta \subseteq \text{Head}(r)} \Phi(\Delta) \iff A \land \bigwedge_{\Delta \subseteq \text{Body}(r)} df(A) \land \bigwedge_{\Delta \subseteq \text{Head}(r)} \neg \Phi(\Delta)
\]

Due to Proposition 6, each implication in Theorem 2 can be written as a set of \( HT \) rules, because \( \Phi(\Delta) \) is a conjunction in the head \( \alpha \leq x \land \beta \land \beta \) and, by De Morgan, \( \neg \Phi(\Delta) \) becomes a disjunction in the body \( \neg (\alpha \leq x) \lor \neg (x \leq \beta) \), 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

\[
y := x - 1 \leftarrow \neg(1 \leq z)
\]

corresponds to the set of \( HT \) rules:

\[
y = x - 1 \leftarrow \neg(1 \leq z), df(x)
y \leftarrow \neg(1 \leq z), \neg(y = x - 1)
\]
Suppose our \(LC\)-program consists of rule (9) only. The intuition is that \(-1 \leq z\) 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 \(s\) and \(z\) 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 \(df\). For \(x\) to be provided a founded value for \(x\), \(x\) 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:
\[
z := x; t := y
\]

**Example 4** Take the program containing (12) and the rules:
\[
x := 1
\]
\[
y := 1 \leftarrow z = 1
\]
\[
z := 1 \leftarrow y = 1
\]

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 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 \(HTC\)-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 \(DF\) stand for the set of choice axioms \(x = x\) for all variables in \(X\).

**Proposition 7** For any linear expression \(\alpha\): \(DF \models df\alpha \equiv \top\)

Let \(\Pi\) be an \(LC\)-program for signature \(\Sigma = \{X, Z, A\}\) where \(A\) is a set of linear constraints. We define a set of auxiliary propositions \(P = \{x^\delta | x \in X\}\). Intuitively, a proposition \(x^\delta\) represents the fact that variable \(x\) has a defined value in the original program \(\Pi\). The translation of \(\Pi\) gives a new \(HTC\) program \(\tau(\Pi)\) for the extended signature \(\tau(\Sigma) = \{X \cup P, Z \cup \{t\}, A \cup P\}\). For any linear expression \(\alpha\), we write \(\alpha^0\) to stand for the conjunction of all propositions \(y^\delta\) for all \(y \in \text{Vars}(\alpha)\) and \((\alpha \leq \beta)^0\) to stand for the conjunction \(\alpha^0 \land \beta^0\). Using this notation, the translation of a linear constraint \(A\) is the formula \(\tau(A) \equiv A \land A^\delta\). Intuitively, due to the choice axiom \(DF\) (applied only on \(X\)), \(A\) can hold due to an arbitrary assignment of variable values, but \(A^\delta\) guarantees that all variables have been assigned a founded value wrt program \(\Pi\). Notice that the translation of \(df\alpha\) corresponds to \(\tau(df\alpha) = \tau(\alpha \leq \alpha) = \alpha \leq \alpha \land \alpha^\delta\) and, under the assumption \(DF\), the latter is equivalent to \(\alpha^0\) (Proposition 7).

For any arbitrary formula \(\phi\), \(\tau(\phi)\) stands for the replacement of every constraint atom \(A\) in \(\phi\) by \(\tau(A)\). The translation of an \(LC\)-program \(\Pi\) corresponds to the set of formulas \(\tau(\Pi) \equiv \{\tau(\phi) | \phi \in \Pi\}\).

As we see below, the models of the translation \(\tau(\Pi)\) are isomorphic to the original models of \(\Pi\). Thus, we can apply \(\tau\) 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:
\[
z = x \land z^\delta; t = y \land t^\delta \leftarrow x^\delta, y^\delta
\]
\[
z = x \land z^\delta \leftarrow x^\delta, \neg(t = y \land t^\delta \land y^\delta)
\]
\[
t = y \land t^\delta \leftarrow y^\delta, \neg(z = x \land x^\delta \land x^\delta)
\]
\[
\bot \leftarrow \neg(z = x \land x^\delta \land x^\delta),
\]
\[
\neg(t = y \land t^\delta \land y^\delta)
\]

Given a valuation \(v\) for the extended signature \(\tau(\Sigma)\), we define its corresponding “defined” subset in signature \(\Sigma\) as \(v|_{\delta} \equiv \{(x, d) \in v | (x^\delta, t) \in v\}\).

**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 \((H, T)\) be an \(HTC\) interpretation for signature \(\tau(\Sigma)\). Then, for any constraint atom \(A \in A\), \((H, T) \models \tau(A)\) iff \((H|_{\delta}, T|_{\delta}) \models A\).

By a simple application of structural induction, we get:

**Corollary 1** Let \((H, T)\) be an \(HTC\) interpretation for signature \(\tau(\Sigma)\). Then, for any formula \(\phi\), \((H, T) \models \tau(\phi)\) iff \((H|_{\delta}, T|_{\delta}) \models \phi\).

**Theorem 3** (Soundness) Let \(T\) be a stable model of \(\tau(\Pi) \cup DF\). Then, \(T|_{\delta}\) is a stable model of \(LC\)-program \(\Pi\).
Theorem 4 (Completeness) Let \( T \) be a stable model of LC-program \( \Pi \). Then, any \( T' \) such that \( T' \models DF \) and \( T'|_{\emptyset} = T \) is a stable model of \( \tau(\Pi) \cup DF \).

That is, each stable model \( T \) of \( \Pi \) is in one-to-one correspondence to a class of stable models \( T' \) of \( \tau(\Pi) \cup DF \) that coincide with \( T \) in the valuation of its defined variables, making \( x^d \) true for all of them, and freely varying the other variables. For instance, the above program \( \Pi \) 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. \( x = 1 \wedge x^5 \) and \( y = 2 \wedge y^5 \). Also, \( DF \) includes axioms \( x = x, y = y \) and \( z = z \). The resulting translation \( \tau(\Pi) \cup DF \) yields two sets of stable models: one of the form \( \{(x,1),(y,2),(z,1),(x^5,t),(y^5,t),(z^5,t),(t,d)\} \) and another \( \{(x,1),(y,2),(t,2),(x^5,t),(y^5,t),(t^5,t),(z,d)\} \) in both cases for any \( d \in D \).

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 
\[ \text{'&assign } A_1;\ldots;A_n \text{'} \]. Similarly, linear expressions are formed using the &sum functor. A linear constraint of form \( \alpha_1 + \cdots + \alpha_n < \beta \) is written as 
\[ \text{'&sum } \{ \alpha_1;\ldots;\alpha_n \text{'} < \beta \text{'} \]. Where \( \prec \) is among \( <,=,\rightarrow \), \( \leftarrow \). 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 (e.g. \( t \) 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:

\begin{verbatim}
1 \n(1,8).
2 :- not \&distinct \{ q(X) \mid n(X) \}. 
3 :- \&sum \{ q(X) \mid -q(V) \} \leftarrow X-Y, n(X), n(V), X \neq Y.
4 :- \&sum \{ q(X) \mid -q(V) \} \leftarrow X-Y, n(X), n(V), X \neq Y.
5 \&assign \{ q(X) \leftarrow \} :- not \&sum \{ q(Y) \} \leftarrow 1.
6 \&assign \{ q(X) \leftarrow \_..\_ \} :- n(X), X > 1.
\end{verbatim}

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 \( \text{'&assign } q(1) \leftarrow 4 \text{'} \) 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; Balducci, 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; Balducci, 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; Balducci, 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


