Constraint Answer Set Programming versus Satisfiability Modulo Theories Or Constraints versus Theories[⋆]

Yuliya Lierler[1] and Benjamin Susman[2]

[1] University of Nebraska at Omaha ylierler@unomaha.edu
[2] University of Nebraska at Omaha bsusman@unomaha.edu

Abstract. Constraint answer set programming is a promising research direction that integrates answer set programming with constraint processing. It is often informally related to the field of Satisfiability Modulo Theories. Yet the exact formal link is obscured as the terminology and concepts used in these two research areas differ. In this paper, we make the link between these two areas precise.

1 Introduction

Constraint answer set programming (CASP) [12] is a promising research direction that integrates answer set programming (ASP), a powerful knowledge representation paradigm, with constraint processing. Typical answer set programing tools start their computation with grounding, a process that substitutes variables for passing constants in respective domains. Thus large domains often form an obstacle for classical ASP. CASP enables a mechanism to model constraints over large domains so that they are processed in a non-typical way for ASP tools by delegating the solving of constraints to constraint solver systems specifically designed to handle large and sometimes infinite domains. CASP solvers including CLINGCON [7] and EZCSP [1] already put the CASP on the map of efficient automated reasoning tools.

CASP often cites itself as a related initiative to Satisfiability Modulo Theories (SMT) solving [3]. Yet, the exact link is obscured as the terminology and concepts used in both fields differ. To add to the complexity of the picture several answer set programming modulo theories formalisms have been proposed. Liu et.al. [14], Janhunen et.al [10], and Lee and Meng [11] introduced logic programs modulo linear constraints, logic programs modulo difference constraints, and ASPMT programs respectively.

In this work we attempt to unify the terminology used in CASP and SMT so that the differences and similarities of logic programs with constraints versus logic programs modulo theories becomes apparent. At the same time, we introduce the notion of constraint formulas which is similar to that of logic programs with constraints. We characterize a special class of SMT theories that we call "uniform theories" so that when uniform theories are used the two formalisms, logic programs with constraints

⋆ We would like to thank Martin Brain for the discussions that lead us to undertake this research. We are also grateful to Vladimir Lifschitz, Cesare Tinelli, and Mirek Truszczynski for the conversations related to the topic of the paper.

and logic programs modulo theories, become the same. This unified outlook allows us not only to better understand the landscape of CASP languages and systems, but also to foster new ideas for CASP solvers design as well as SMT solvers design.

Section 2 reviews concepts of logic programs, completion, and (input) answer sets. In Section 3, we introduce (i) generalized constraint satisfaction problems, (ii) constraint answer set programs, and (iii) constraint formulas. Section 4 starts by introducing satisfiability modulo theories and respective SMT programs. We also identify a special class of uniform theories that are commonly used in satisfiability modulo solving. This class of theories helps us to establish precise links (i) between constraint formulas and SMT programs, and (ii) between CASP and SMT. In Section 5, we conclude by relating a family of distinct constraint answer set programming formalisms.

2 Logic Programs, Completion, and Input Answer Sets

A *vocabulary* is a set of propositional symbols also called atoms. A *(propositional) logic program*, denoted by Π, over vocabulary σ is a set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m, \tag{1}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq m$, is an atom in σ. We will sometime use the abbreviated form for a rule (1)

$$a \leftarrow B \tag{2}$$

where B stands for $b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m$, and is also called a *body*.

The expression a is the *head* of the rule. When a is $\bot$, we often omit it and say that the head is empty. We write $hd(\Pi)$ for the set of nonempty heads of rules in Π. The *reduct* Π^X of a program Π relative to a set X of atoms is the set of rules

$$a \leftarrow b_1, \ldots, b_\ell \tag{3}$$

for all rules (1) in Π such that $b_{\ell+1}, \ldots, b_m \notin X$. A set X of atoms *satisfies* a (positive) rule (3) if $a \in X$ whenever $\{b_1, \ldots, b_\ell\} \subseteq X$. We say that a set X of atoms is an *answer set* when X is the smallest set of atoms that satisfies all rules in Π^X [8].

We call a rule whose body is empty a *fact*. In such cases, we drop the arrow. We sometimes may identify a set X of atoms with a set of facts $\{a. \mid a \in X\}$. Also, it is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value $true$ to every atom in X and $false$ to every atom in $\sigma \setminus X$.

For a program Π over vocabulary σ, the *completion* of Π [4], denoted by $Comp(\Pi)$, is the set of classical formulas that consists of the implications $B \to a$ for all rules (2) in Π and the implications

$$a \to \bigvee_{a \leftarrow B \in \Pi} B \tag{4}$$

for all atoms a in σ.

It is well known that for the large class of logic programs, referred to as *tight* programs, its answer sets coincide with models of its completion, as shown by Fages [6]. Tightness is a syntactic condition on a program that can be verified by means of program's dependency graph. The *dependency graph* of Π is the directed graph G such that (i) the vertices of G are the atoms occurring in Π, and (ii) for every rule (1) in Π whose head is not $\bot$, G has an edge from atom a to each atom $b_1 \dots b_\ell$. A program is called *tight* if its dependency graph is acyclic.

We now introduce a generalization of a concept of an input answer set by Lierler and Truszczynski [13]. In this work, we consider input answer sets relative to input vocabularies. We then extend the definition of completion so that we can state the result by Fages for the case of input answer sets. These concepts are essential for introducing constraint answer set programs as well as constraint formulas and establishing a formal relation between them.

Definition 1. *For a logic program Π over vocabulary σ, a set X of atoms over σ is an* input answer set *of Π relative to vocabulary $\iota \subseteq \sigma$ when X is an answer set of the program $\Pi \cup ((X \cap \iota) \setminus hd(\Pi))$.*

Definition 2. *For a program Π over vocabulary σ, the* input-completion *of Π relative to vocabulary $\iota \subseteq \sigma$, denoted by $IComp(\Pi, \iota)$, is defined as the set of propositional formulas (formulas in propositional logic) that consists of the implications $B \rightarrow a$ for all rules* (2) *in Π and the implications* (4) *for all atoms a occurring in $(\sigma \setminus \iota) \cup hd(\Pi)$.*

Theorem 1. *For a tight program Π over vocabulary σ and vocabulary $\iota \subseteq \sigma$, a set X of atoms from σ is an input answer set of Π relative to ι if and only if X satisfies program's input-completion $IComp(\Pi, \iota)$.*

3 Generalized Constraint Answer Set Programs

We start this section by presenting primitive constraints as defined by Marriott and Stuckey [15, Section 1.1] using the notation convenient for our purposes. We refer to this concept as a constraint dropping the word "primitive". We use constraints to define a notion of a generalized constraint satisfaction problem that Marriott and Stuckey refer to as "constraint". We then review constraint satisfaction problems as commonly defined in artificial intelligence literature and illustrate that they are special case of generalized constraint satisfaction problems.

Constraints and Generalized Constraint Satisfaction Problem We adopt the following convention: for a function ν and an element x, by x^ν we denote the value assigned by ν to x. A *domain* is a *nonempty* set of elements (values). A *signature* Σ is a set of *variables*, *function symbols (or f-symbols)*, and *predicate symbols*. Function and predicate symbols are associated with a positive integer called *arity*. By $\Sigma_{|v}$, $\Sigma_{|r}$, and $\Sigma_{|f}$ we denote the subsets of Σ that contain all variables, all predicate symbols, and all f-symbols respectively.

For instance, we can define signature $\Sigma_1 = \{s, r, E, Q\}$ by saying that s and r are variables, E is a predicate symbol of arity 1, and Q is a predicate symbol of arity 2. Then, $\Sigma_{1|v} = \{s, r\}$, $\Sigma_{1|r} = \{E, Q\}$, $\Sigma_{1|f} = \emptyset$.

Let D be a domain. For a set V of variables, we call a function $\nu : V \to D$ a $[V, D]$ *valuation*. For a set F of f-symbols, we call a total function on F *an* $[F, D]$ *f-denotation*, when it maps an n-ary f-symbol into a function $D^n \to D$. For a set R of predicate symbols, we call a total function on R *an* $[R, D]$ *r-denotation*, when it maps an n-ary predicate symbol into an n-ary relation on D.

A table below presents definitions of sample domain D_1, valuations ν_1, ν_2, and r-denotations ρ_1 and ρ_2.

D_1	$\{1, 2, 3\}$
ν_1	$[\Sigma_{1\vert v}, D_1]$ valuation, where $s^{\nu_1} = r^{\nu_1} = 1$
ν_2	$[\Sigma_{1\vert v}, D_1]$ valuation, where $s^{\nu_2} = 2$ and $r^{\nu_2} = 1$
ρ_1	$[\Sigma_{1\vert r}, D_1]$ r-denotation, where $E^{\rho_1} = \{\langle 1\rangle\}$, $Q^{\rho_1} = \{\langle 1, 1\rangle, \langle 2, 2\rangle, \langle 3, 3\rangle\}$
ρ_2	$[\Sigma_{1\vert r}, D_1]$ r-denotation, where $Q^{\rho_2} = Q^{\rho_1}$ and $E^{\rho_2} = \{\langle 2\rangle, \langle 3\rangle\}$.

A *constraint vocabulary (c-vocabulary)* is a pair $[\Sigma, D]$, where Σ is a signature and D is a domain. A *term* over a c-vocabulary $[\Sigma, D]$ is either
- a variable in $\Sigma_{|v}$, or
- a domain element in D, or
- an expression $f(t_1, \ldots, t_n)$, where f is an f-symbol of arity n in $\Sigma_{|f}$ and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$

A *constraint atom* over a c-vocabulary $[\Sigma, D]$ is an expression

$$P(t_1, \ldots, t_n), \tag{5}$$

where P is a predicate symbol from $\Sigma_{|r}$ of arity n and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$. A *constraint literal* over a c-vocabulary $[\Sigma, D]$ is either a constraint atom (5) or an expression

$$\neg P(t_1, \ldots, t_n), \tag{6}$$

where $P(t_1, \ldots, t_n)$ is a constraint atom over $[\Sigma, D]$. For instance, expressions $\neg E(s)$, $\neg E(2)$, and $Q(r, s)$ are constraint literals over $[\Sigma_1, D_1]$.

Let $[\Sigma, D]$ be a c-vocabulary, ν be a $[\Sigma_{|v}, D]$ valuation, ϕ be a $[\Sigma_{|f}, D]$ f-denotation, and ρ be a $[\Sigma_{|r}, D]$ r-denotation. First, we define recursively a value that valuation ν assigns to a term τ over $[\Sigma, D]$ w.r.t. ϕ. We denote this value by $\tau^{\nu,\phi}$. For a term that is a variable x in $\Sigma_{|v}$, $x^{\nu,\phi} = x^{\nu}$. For a term that is a domain element d in D, $d^{\nu,\phi}$ is d itself. For a term τ of the form $f(t_1, \ldots, t_n)$, $\tau^{\nu,\phi}$ is defined recursively by the formula $f(t_1, \ldots, t_n)^{\nu,\phi} = f^{\phi}(t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi})$. Second, we define what it means for valuation to be a solution of a constraint literal w.r.t. given f-and-r-denotations. We say that ν *satisfies (is a solution to)* constraint literal (5) over $[\Sigma, D]$ *w.r.t.* ϕ *and* ρ when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi}\rangle \in P^{\rho}$. Let $\mathcal{R}$ be an n-ary relation on D. By $\overline{\mathcal{R}}$ we denote *complement* relation of $\mathcal{R}$ constructed as $D^n \setminus \mathcal{R}$. Valuation ν *satisfies (is a solution to)* constraint literal of the form (6) *w.r.t.* ϕ *and* ρ when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi}\rangle \in \overline{P^{\rho}}$. For instance, valuation ν_1 satisfies constraint literal $Q(r, s)$ w.r.t. ρ_1, while valuation ν_2 does not satisfy this constraint literal w.r.t. ρ_2 (when a signature contains no function symbols no reference to f-denotation is necessary in the definitions above).

We are now ready to define constraints, their syntax and semantics. To begin we introduce a *lexicon*, which is a tuple $([\Sigma, D], \rho, \phi)$, where $[\Sigma, D]$ is a c-vocabulary, ρ is

$[\Sigma_{|r}, D]$ r-denotation, and ϕ is $[\Sigma_{|f}, D]$ f-denotation. For a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, we call any function that is $[\Sigma_{|v}, D]$ valuation, a *valuation over* $\mathcal{L}$. We will omit the last element of the tuple if the signature Σ of the lexicon contains no f-symbols. A *constraint* is defined over lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$. Syntactically, it is a constraint literal over $[\Sigma, D]$ (lexicon $\mathcal{L}$, respectively). Semantically, we say that valuation ν over $\mathcal{L}$ *satisfies (is a solution to)* the constraint c when ν satisfies c w.r.t. ϕ and ρ. For instance, the table below presents definitions of sample lexicons $\mathcal{L}_1$, $\mathcal{L}_2$, and constraints c_1, c_2, c_3, and c_4.

$\mathcal{L}_1$	$([\Sigma_1, D_1], \rho_1)$
$\mathcal{L}_2$	$([\Sigma_1, D_1], \rho_2)$
c_1	a literal $Q(r, s)$ over lexicon $\mathcal{L}_1$
c_1	a literal $Q(r, s)$ over lexicon $\mathcal{L}_2$
c_3	a literal $\neg E(s)$ over lexicon $\mathcal{L}_2$
c_4	a literal $\neg E(2)$ over lexicon $\mathcal{L}_2$.

Valuation ν_1 is a solution to c_1, c_2, c_3, but not a solution to c_4. Valuation ν_2 is a solution to c_3, but not a solution to c_1, c_2, and c_4. In fact, constraint c_4 has no solutions. We sometimes omit the explicit mention of the lexicon when talking about constraints: we then may identify a constraint with its syntactic form of a constraint literal.

Definition 3. *A* generalized constraint satisfaction problem (GCSP) $\mathcal{C}$ *is a finite set of constraints over a lexicon* $\mathcal{L} = ([\Sigma, D], \rho, \phi)$*. We say that a valuation* ν *over* $\mathcal{L}$ satisfies (is a solution to) *GCSP* $\mathcal{C}$ *when* ν *is a solution to every constraint in* $\mathcal{C}$*.*

For example, any subset of set $\{c_2, c_3, c_4\}$ of constraints forms a GCSP.

From GCSP to Constraint Satisfaction Problem We say that a lexicon is *finite-domain* if it is defined over a c-vocabulary that refers to a domain whose set of elements is finite. Trivially, lexicons $\mathcal{L}_1$ and $\mathcal{L}_2$ are finite-domain. Consider a special case of a constraint of the form (5) over finite-domain lexicon $\mathcal{L} = ([\Sigma, D], \rho)$ so that each t_i is a variable. (For instance, constraints c_1, c_2, and c_3 satisfy the stated requirements, while c_4 does not.) In this case, we can syntactically identify (5) with the pair

$$\langle (t_1, \dots, t_n), P^\rho \rangle. \tag{7}$$

A *constraint satisfaction problem* (CSP) is a set of pairs (7), where $\Sigma_{|v}$ and D of the finite-domain lexicon $\mathcal{L}$ are called variables and domain of CSP, respectively. Saying that valuation ν over $\mathcal{L}$ satisfies (5) is the same as saying that $\langle t_1^\nu, \dots, t_n^\nu \rangle \in P^\rho$. The latter is the way in which a solution to expressions (7) in CSP is typically defined. As in the definition of semantics of GCSP, a valuation is a *solution* to a CSP problem C when it is a solution to every pair (7) in C. In conclusion, GCSP generalizes CSP by (i) elevating the restriction of finite domain, and (ii) allowing us more elaborate syntactic expressions (e.g., recall f-symbols).

Constraint Answer Set Programs and Constraint Formulas Let σ_r and σ_i be two disjoint vocabularies. We refer to their elements as *regular* atoms and *irregular* atoms. For the rest of this paper we will assume the convention that σ_r and σ_i represent disjoint vocabularies of so called regular and irregular atoms. For a program Π, by $At(\Pi)$ we denote the set of atoms occurring in it.

Definition 4. *A* constraint answer set program *(CAS program) over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle \Pi, \mathcal{B}, \gamma \rangle$, where Π is a logic program over the vocabulary σ such that $hd(\Pi) \cap \sigma_i = \emptyset$, $\mathcal{B}$ is a set of constraints over the same lexicon, and γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.*

For a CAS program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ so that $\mathcal{L}$ is the lexicon of the constraints in $\mathcal{B}$, a set $X \subseteq \sigma$ is an answer set *of P if*
- *$X \subseteq At(\Pi)$*
- *X is an input answer set of Π relative to σ_i, and*
- *the following GCSP over $\mathcal{L}$ has a solution*

$$\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(\Pi) \cap \sigma_i) \setminus X\}.$$

These definitions are generalizations of CAS programs introduced by Gebser et al. [7] as they refer to the concept of GCSP in place of CSP in the original definition.

Just as we defined constraint answer set programs, we can define constraint formulas. For a propositional formula F, by $At(F)$ we denote the set of atoms (propositional symbols) occurring in it.

Definition 5. *A* constraint formula *over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, \mathcal{B}, \gamma \rangle$, where F is a propositional formula over the vocabulary σ, $\mathcal{B}$ is a set of constraints over the same lexicon, and γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.*

For a constraint formula $\mathcal{F} = \langle F, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ so that $\mathcal{L}$ is the lexicon of the constraints in $\mathcal{B}$, a set $X \subseteq \sigma$ is a model *of $\mathcal{F}$ if*
- *$X \subseteq At(F)$*
- *X is a model of F, and*
- *the following GCSP over $\mathcal{L}$ has a solution*

$$\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(F) \cap \sigma_i) \setminus X\}.$$

Theorem 2. *For a CAS program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ and a set X of atoms over σ, when Π is tight, X is an answer set of P if and only if X is a model of constraint formula $\langle IComp(\Pi, \sigma_i), \mathcal{B}, \gamma \rangle$ over $\sigma = \sigma_r \cup \sigma_i$.*

4 Satisfiability Modulo Theories versus Constraint Formulas

First, in this section we introduce the notion of a "theory" in Satisfiability Modulo Theories (SMT) [3]. Second, we present the definition of a "restriction formula" and state the conditions under which such formulas are satisfied by a given interpretation. The introduced restriction formulas are syntactically restricted classical ground predicate logic formulas. The presented notions of interpretation and satisfaction are usual but are stated in terms convenient for our purposes. In the literature on SMT, a more sophisticated syntax, than restriction formulas provide, is typically discussed. Yet, SMT solvers often rely on the so called propositional abstractions of predicate logic formulas which, in their most commonly used case, coincide with restriction formulas discussed here.

An *interpretation* I for a signature Σ, or Σ-interpretation, is a tuple (D, ν, ρ, ϕ), where

– D is a domain,
– ν is a $[\Sigma_{|v}, D]$ valuation,
– ρ is a $[\Sigma_{|r}, D]$ r-denotation, and
– ϕ is a $[\Sigma_{|f}, D]$ f-denotation.

For signatures that contains no f-symbols, we omit the reference to the last element of the interpretation tuple.

For a signature Σ, a *Σ-theory* is a set of interpretations over Σ. For instance, for signature Σ_1, by $\mathcal{I}_1$ and $\mathcal{I}_2$ we denote the following sample interpretations (D_1, ν_1, ρ_1) and (D_1, ν_2, ρ_1) respectively. Any subset of interpretations $\{\mathcal{I}_1, \mathcal{I}_2\}$ exemplifies a unique Σ_1-theory.

In literature on predicate logic and SMT, the term "object constant" or "function symbol of arity 0" is commonly used to refer to elements in the signature that we call variables. Using the terms that stem from definitions related to constraint satisfaction processing will facilitate uncovering the precise link between CASP-like formalisms and SMT-like formalisms. Also it is typical for predicate logic signatures to contain propositional symbols (predicate symbols of arity 0). It is easy to extend the notion of signature introduced here to allow propositional symbols. Yet it will complicate the presentation, which is the reason we avoid this extension.

A *restriction formula* over signature Σ is a finite set of constraint literals over c-vocabulary $[\Sigma, \emptyset]$. A sample restriction formula over Σ_1 follows

$$\{\neg E(s), \neg Q(r, s)\}. \tag{8}$$

We now state the semantics of restriction formulas. Consider a Σ-interpretation $I = (D, \nu, \rho, \phi)$. To each term τ over c-vocabulary $[\Sigma, \emptyset]$, I assigns a value $\tau^{\nu,\phi}$ that we denote by r^I. We say that I *satisfies* restriction formula Φ over Σ when ν satisfies every constraint literal in Φ w.r.t. ϕ and ρ. Interpretation $\mathcal{I}_2$ satisfies restriction formula (8), while $\mathcal{I}_1$ does not satisfy (8).

We say that a restriction formula Φ over signature Σ is *satisfiable* in a Σ-theory T, or is T-satisfiable, when there is an element of the set T that satisfies Φ. For example, restriction formula (8) is satisfiable in any Σ_1-theory that contains interpretation $\mathcal{I}_2$. On the other hand, restriction formula (8) is not satisfiable in Σ_1-theory $\{\mathcal{I}_1\}$.

SMT often considers theories of a special kind. For instance, the presented definition of a theory places no restrictions on the domains, r-denotations, or f-denotations being identical across the interpretations defining the theory. In practice, such restrictions are very common. Later in the presentation we will define so called "uniform" theories that follow typical restrictions. We will then show how such uniform theories and restriction formulas can practically be seen as syntactic variants of GCSPs.

SMT and ASPT Programs For signature Σ, domain D, and a vocabulary σ', a $[\sigma', \Sigma, D]$*-mapping* is an injective mapping μ from atoms over σ' to constraint atoms over c-vocabulary $[\Sigma, D]$. For a $[\sigma', \Sigma, D]$-mapping μ, a consistent set M of propositional literals over vocabulary $\sigma \supseteq \sigma'$ is a *T-model of μ (or, T, μ-model)* if a restriction formula

$$\{\mu(a) |\ a \in M \cap \sigma'\} \cup \{\neg\mu(a) \mid \neg a \in M \text{ and } a \in \sigma'\}$$

is satisfiable in Σ-theory T.

Definition 6. *An* SMT program *P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, T, \mu \rangle$, where F is a propositional formula over σ, μ is a $[\sigma_i, \Sigma, \emptyset]$-mapping, and T is a Σ-theory.*

For an SMT program $\langle F, T, \mu \rangle$ over σ, a set $X \subseteq \sigma$ is its model *if*
1. *$X \subseteq At(F)$,*
2. *X is a model of F, and*
3. *$X \cup \{\neg a \mid a \in At(F) \setminus X\}$ is a T, μ-model.*

We now define the concept of logic programs modulo theories.

Definition 7. *A* logic program modulo theories *(or* ASPT program*) P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle \Pi, T, \mu \rangle$, where Π is a logic program over σ, μ is a $[\sigma_i, \Sigma, \emptyset]$-mapping, and T is a Σ-theory.*

For an ASPT program $\langle \Pi, T, \mu \rangle$ over σ, a set $X \subseteq \sigma$ is its model *if*
1. *$X \subseteq At(\Pi)$,*
2. *X is an input answer set of Π relative to σ_i, and*
3. *$X \cup \{\neg a \mid a \in At(\Pi) \setminus X\}$ is a T, μ-model.*

Uniform Theories We now identify a special class of theories that we call “uniform”. We then relate the question of verifying satisfiability of formulas in such theories to the question of solving relevant generalized constraint satisfaction problem. This connection brings us to a straightforward relation between SMT program formalism over uniform theories and constraint formula formalism as well as between CAS programs and ASPT programs. In the following section we list several common SMT fragments such as satisfiability modulo difference logic and satisfiability modulo linear arithmetic whose theories are, in fact, uniform theories. We then use these findings to relate several ASP modulo theories approaches such as ASP(DL) introduced in [14] and ASP(LC) introduced in [14] to CASP approaches.

For a signature Σ, we call a Σ-theory T *uniform* over lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$ when (i) all interpretations in T are of the form (D, ν, ρ, ϕ) (note how valuation ν is the only not fixed element in the interpretations), and (ii) for every possible $[\Sigma_{|v}, D]$ valuation ν, there is an interpretation (D, ν, ρ, ϕ) in T.

To illustrate a concept of a uniform theory, a table below defines sample domain D_2, valuations ν_3 and ν_4, and r-denotation ρ_3.

D_2	$\{1, 2\}$
ν_3	$[\Sigma_{1\,|v}, D_2]$ valuation, where $s^{\nu_3} = 1$ and $r^{\nu_3} = 2$
ν_4	$[\Sigma_{1\,|v}, D_2]$ valuation, where $s^{\nu_4} = r^{\nu_4} = 2$
ρ_3	$[\Sigma_{1\,|r}, D_2]$ r-denotation, where $E^{\rho_3} = \{\langle 2 \rangle\}$, $Q^{\rho_3} = \{\langle 1, 1 \rangle, \langle 2, 2 \rangle\}$

Valuations ν_1 and ν_2 can be seen not only as $[\Sigma_{1\,|v}, D_1]$ valuations but also as $[\Sigma_{1\,|v}, D_2]$ valuations. The set

$$\{(D_2, \nu_1, \rho_3), (D_2, \nu_2, \rho_3), (D_2, \nu_3, \rho_3), (D_2, \nu_4, \rho_3)\}$$

of Σ_1 interpretations is an example of a uniform theory over lexicon $([\Sigma_1, D_2], \rho_3)$. We denote this theory by T_1. Neither of Σ_1-theories $\{\mathcal{I}_1\}$, $\{\mathcal{I}_1, \mathcal{I}_2\}$ is uniform over lexicon $([\Sigma_1, D_1], \rho_1)$.

It is easy to see that for uniform theories we can identify their interpretations of the form (D, ν, ρ, ϕ) with their second element valuation ν. Indeed, the other three elements are fixed by the lexicon over which the uniform theory is defined. In the following we will sometimes use this convention. For example, we may refer to interpretation (D_2, ν_1, ρ_3) of uniform theory T_1 as ν_1.

For uniform Σ-theory T over lexicon $([\Sigma, D], \rho, \phi)$ we can extend the syntax of restriction formulas by saying that a *restriction formula* is defined over c-vocabulary $[\Sigma, D]$ as a finite set of constraint literals over $[\Sigma, D]$ (earlier we considered constraint literals over $[\Sigma, \emptyset]$). The earlier definition of semantics is still applicable. In the following for the uniform theories we assume such a more general syntax. Also we can extend the definition of SMT program given a constraint Σ-theory T over lexicon $([\Sigma, D], \rho, \phi)$ as follows: an *SMT program* P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, T, \mu \rangle$, where F is a propositional formula over σ, μ is a $[\sigma_i, \Sigma, D]$-mapping, and T is a Σ-theory. Note how μ-mapping refers do the domain of lexicon now in place of an empty set in the earlier definition. The definition of ASPT program can be extended in the same style. For the case of uniform theories we will assume the definition of SMT programs as stated in this paragraph. The same applies to the case of ASPT programs module uniform theories.

We now present a theorem that makes the connection between GCSPs over some lexicon $\mathcal{L}$ and restriction formulas interpreted using the uniform theory T over the same lexicon $\mathcal{L}$ apparent: the question whether a given GSCP over $\mathcal{L}$ has a solution translates into the question whether the set of constraint literals of GSCP forming a restriction formula is T-satisfiable. Furthermore, any solution to such GSPC is also an interpretation in T that satisfies the respective restriction formula, and the other way around. We then relate SMT programs "modulo uniform theories" and constraint formulas, as well as ASPT programs and CAS programs.

Theorem 3. *For a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, a set Φ of constraint literals over c-vocabulary $[\Sigma, D]$, a uniform Σ-theory T over lexicon $\mathcal{L}$, the following holds*
1. *for any $[\Sigma_{|v}, D]$ valuation ν, there is an interpretation ν in T,*
2. *$[\Sigma_{|v}, D]$ valuation ν is a solution to GCSP Φ over lexicon $\mathcal{L}$ if and only if interpretation ν in T satisfies Φ.*
3. *GCSP Φ over lexicon $\mathcal{L}$ has a solution if and only if Φ is T-satisfiable.*

Let $\mathcal{L}$ denote a lexicon $([\Sigma, D], \rho, \phi)$. By $\mathcal{B}_{\mathcal{L}}$ we denote the set of all constraints over $\mathcal{L}$ of the form (5) (in other words all constraints that syntactically are represented by constraint atoms rather than constraint literals). By $T_{\mathcal{L}}$ we denote the uniform Σ-theory over $\mathcal{L}$.

Theorem 4. *For a signature Σ, a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, vocabularies $\sigma = \sigma_i \cup \sigma_r$, a $[\sigma_i, \Sigma, D]$-mapping μ, and a set $X \subseteq \sigma$, X is a model of an SMT program $\langle F, T_{\mathcal{L}}, \mu \rangle$ over σ if and only if X is a model of a constraint formula $\langle F, \mathcal{B}_{\mathcal{L}}, \mu \rangle$ over σ, where μ is identified with the function from irregular atoms to constraints over $\mathcal{L}$ in a trivial way.*

This theorem illustrates that for uniform theories the language of SMT programs and constraint formulas coincide. Or, that the language of constraint formulas is a special case of SMT programs that are defined over uniform theories. We now show similar relation between CAS and ASPT programs.

Theorem 5. *For a signature Σ, a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, vocabularies $\sigma = \sigma_i \cup \sigma_r$, a $[\sigma_i, \Sigma, D]$-mapping μ, and a set $X \subseteq \sigma$, X is a model of an ASPT program $\langle \Pi, T_{\mathcal{L}}, \mu\rangle$ over σ if and only if X is a model of a CAS program $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \mu\rangle$ over σ, where μ is identified with the function from irregular atoms to constraints over $\mathcal{L}$ in a trivial way.*

5 CAS over Numeric Constraints or ASPT over Numeric Theories

In this section we first illustrate how GCSP model constraint problems over so called numeric constraints. Next we proceed at defining "numeric" uniform theories. These definitions will allow us to precisely define the languages used by various constraint answer set solvers. We conclude with the discussion of the variety of solving techniques used in logic programming community.

Linear, Integer Linear, and Finite-Domain Constraints Let $\mathbb{Z}$ and $\mathbb{R}$ denote the sets of integers and real numbers respectively. We say that a signature is *numeric* when it satisfies the following requirements (i) its only f-symbols are $+$, $\times$ of arity 2, and (ii) its only predicate symbols are $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ also of arity 2. In the following we use the symbol $\mathcal{A}$ to denote a numeric signature. Let $\phi_{\mathbb{Z}}$ and $\rho_{\mathbb{Z}}$ be $[\{+, \times\}, \mathbb{Z}]$ f-denotation and $[\{<, >, \leq, \geq, =, \neq\}, \mathbb{Z}]$ r-denotation respectively where they map their function and predicate symbols into usual arithmetic operations and relations over integers. We call any lexicon of the form $([\mathcal{A}, \mathbb{Z}], \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$ *integer*. Similarly, $f_{\mathbb{R}}$ and $r_{\mathbb{R}}$ denote $[\{+, \times\}, \mathbb{R}]$ f-denotation and $[\{<, >, \leq, \geq, =, \neq\}, \mathbb{R}]$ r-denotation respectively where they map their function an predicate symbols into usual arithmetic operations and relations over reals. We call any lexicon of the form $([\mathcal{A}, \mathbb{R}], r_{\mathbb{R}}, f_{\mathbb{R}})$ *numeric*.

A *linear expression* has the form

$$a_1 x_1 + \cdots + a_n x_n \tag{9}$$

where $a_1, \ldots, a_n$ are real numbers and $x_1, \ldots, x_n$ are variables over real numbers. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. We view this expression as an abbreviation for the following term

$$+(\times(a_1, x_1), +(\times(a_2, x_2), \cdots + (\times(a_{n-1}, x_{n-1}), \times(a_n, x_n)) \ldots),$$

over some c-vocabulary $[\mathcal{A}, \mathbb{R}]$, where $\mathcal{A}$ contains $x_1, \ldots, x_n$ as its variables. For instance, $2x_2 + 3x_3$ is an abbreviation for the expression $+(\times(2, x_2), \times(3, x_3))$. We say that a linear expression is *integer* if it has the form (9), $a_1, \ldots, a_n$ are integers, and $x_1, \ldots, x_n$ are variables over integers.

We call a constraint *linear (integer linear)* when it is defined over some numeric (integer) lexicon and has the form

$$\bowtie (e, k) \tag{10}$$

where e is a linear (integer linear) expression, k is a real number (an integer), and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. We can write (10) as an expression in usual infix notation $e \bowtie k$. We call GCSP a *linear constraint satisfaction problem* when it is composed of linear constraints. Similarly we call GCSP an *integer linear constraint satisfaction problem* when it is composed of integer linear constraints.

By $\hat{\mathbb{Z}}$ we denote any finite subset of $\mathbb{Z}$. Any lexicon of the form $([\mathcal{A}, \hat{\mathbb{Z}}], \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$ is called *finite-domain integer*. We can specialize definitions of linear expressions, linear constraints to the case of finite-domain integer lexicons in an obvious way. We will refer to these concepts as finite-domain linear-expressions, finite-domain linear constraints.

Linear (Integer Linear) Arithmetic and Difference Logic as Uniform Theories Recall that by $\mathcal{A}$ we denote a numeric signature. By $T^{\mathcal{A}}_{\mathbb{Z}}$ we denote the uniform $\mathcal{A}$-theory over integer lexicon $([\mathcal{A}, \mathbb{Z}], \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$. We call theories of this kind *integer*. By $T^{\mathcal{A}}_{\mathbb{R}}$ we denote the constraint $\mathcal{A}$-theory over numeric lexicon $([\mathcal{A}, \mathbb{R}], \rho_{\mathbb{R}}, \phi_{\mathbb{R}})$. We call theories of this kind *numeric*.

Two examples of integer theories commonly implemented in SMT solvers are called *integer linear arithmetic* and *difference logic arithmetic*. *Linear arithmetic* is an example of numeric theory commonly supported by SMT solver. We note that difference logic is a special case of integer linear arithmetic that introduces syntactic requirements on considered expressions [16]. SMT solver CVC4[3] [2] supports all three mentioned arithmetics.

We are now ready to define SMT(DL), SMT(IL), and SMT(L) programs that capture common SMT formalisms used in practice. For a vocabulary σ, we say that $[\sigma, \mathcal{A}, \mathbb{Z}]$-mapping μ is *integer linear (difference-logic) friendly* if any element in σ is mapped to an integer linear formula (difference logic formula) over integer lexicon whose signature is $\mathcal{A}$ (i.e., lexicon $([\mathcal{A}, \mathbb{Z}], \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$). Similarly, for vocabulary σ, we say that $[\sigma, \mathcal{A}, \mathbb{R}]$-mapping μ is *linear friendly* if any element in σ is mapped to a linear formula over numeric lexicon whose signature is $\mathcal{A}$.

We call an SMT program $\langle F, T^{\mathcal{A}}_{\mathbb{Z}}, \mu\rangle$ an *SMT(DL)* or *SMT(IL)* program when μ is difference-logic friendly or integer linear friendly respectively. We call an SMT program $\langle F, T^{\mathcal{A}}_{\mathbb{R}}, \mu\rangle$ an *SMT(L)* program when μ is linear friendly. In the same style we can define the notions of ASPT(DL), ASPT(IL), ASPT(L) programs.

Outlook on Constraint Answer Set Solvers and ASPT Solvers We say that a CAS program $\langle \Pi, \mathcal{B}, \gamma\rangle$ is *(finite-domain) integer* when $\mathcal{B}$ is the set of all (finite-domain) integer linear constraints of the form (5) over some (finite-domain) integer lexicon. Similarly, we say that a CAS program $\langle \Pi, \mathcal{B}, \gamma\rangle$ is *numeric* when $\mathcal{B}$ is the set of all linear constraints of the form (5) over some numeric lexicon. From Theorem 5, it immediately follows that integer CAS programs and numeric CAS programs are in fact the same objects as ASPT(IL) and ASPT(L) programs respectively.

Constraint answer set solver CLINGCON [7] is a system that specializes on finite-domain integer CAS programs. Constraint answer set solver EZCSP [1] is a system that can function in three modes as (i) a solver for finite-domain integer CAS programs, (ii) a solver for integer CAS programs (or, ASPT(IL) programs), (iii) a solver for numeric CAS programs (or, ASPT(L) programs).

At a high-level abstraction one may summarize the architectures of these two solvers as *ASP-base solver plus theory solver*. Given a CAS program $\langle \Pi, \mathcal{B}, \gamma\rangle$, both CLINGCON and EZCSP first use an answer set solver to compute an input answer set of Π. Second, they contact a theory solver to verify whether respective constraint satisfaction problem has a solution. In case of CLINGCON, finite domain constraint solver GECODE

[3] http://cvc4.cs.nyu.edu/web/

is used as a theory solver. System EZCSP, on the other hand, uses constraint logic programming tools such as SICSTUS PROLOG and SWI PROLOG.

Liu et.al. [14] introduced logic programs with linear constraints that they call ASP(LC), while Janhunen et.al [10] introduced logic programs with difference constraints that they call ASP(DL). It turns out that ASP(LC) programs coincide with ASPT(L) programs (or numeric CAS programs). On the other hand, ASP(DL) programs are captured by ASPT(DL) and obviously can be seen as a special case of ASPT(IL) programs (or integer CAS programs).

To process ASP(LC) programs, Liu et.al. [14] introduce the solver called MINGO, which translates the formalism into mixed integer programming formula and then uses the solver CPLEX [9] to solve these formulas. To process ASP(DL) programs, Janhunen et.al [10] implement a system that is called DINGO, which translates the formalism into an SMT(DL) program and applies SMT solver Z3 [5]. Note how EZCSP is an alternative system for computing solutions to the ASP(LC) and ASP(DL) programs.

The diversity of solutions used in constraint answer set programming paradigms also suggests that solutions of the kind are available for SMT technology. Typical SMT architecture is in style of that of the systems CLINGON or EZCSP. One difference is that a satisfiability solver is used as a base solver. Another difference is that theory solvers are typically implemented within an SMT solver and are as such custom solutions. The fact that CLINGON and EZCSP use tools available in constraint programming community suggests that these tools could be of use in SMT community also. The solution exhibited by system MINGO, where mixed integer programming is used for solving ASPT(L) programs, hints that a similar strategy can be implemented for solving SMT(L) programs.

Obviously, Theorems 2 and 3 pave the way for using SMT systems that solve SMT(IL), SMT(L) problems as is for solving ASPT(IL), ASPT(L) programs whose first member of a triple is a tight logic program. It is sufficient to compute input completion of the program relative to irregular atoms.

6 Conclusions

In this paper we unified the terminology stemming from the fields of constraint answer set programming and satisfiability modulo theories solving. This unification helped us identify the special class of so called uniform theories widely used in SMT practice. Given such theories, CASP and SMT solving share more in common than meets the eye. We expect this work to be a strong building block that will bolster the cross-fertilization between three different, even if related, automated reasoning communities: constraint answer set programming, constraint (satisfaction processing) programming, and SMT.

References

1. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. in: Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP), https://www.mat.unical.it/ASPOCP09/ (2009)

2. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4
3. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Clarke, E., Henzinger, T., Veith, H. (eds.) Handbook of Model Checking. Springer (2014)
4. Clark, K.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1978)
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
6. Fages, F.: Consistency of Clark's completion and existence of stable models. Journal of Methods of Logic in Computer Science 1, 51–60 (1994)
7. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of 25th International Conference on Logic Programming (ICLP). pp. 235–249. Springer (2009)
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium. pp. 1070–1080. MIT Press (1988)
9. IBM: IBM ILOG AMPL Version 12.1 User's Guide
10. Janhunen, T., Liu, G., Niemela, I.: Tight integration of non-ground answer set programming and satisfiability modulo theories. In: Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables (2001)
11. Lee, J., Meng, Y.: Answer set programming modulo theories and reasoning about continuous changes. In: IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013 (2013)
12. Lierler, Y.: Relating constraint answer set programming languages and algorithms. Artificial Intelligence 207C, 1–22 (2014)
13. Lierler, Y., Truszczynski, M.: Transition systems for model generators — a unifying approach. Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP'11) Special Issue 11, issue 4-5 (2011)
14. Liu, G., Janhunen, T., Niemela, I.: Answer set programming via mixed integer programming. In: Knowledge Representation and Reasoning Conference (2012), https://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516
15. Marriott, K., Stuckey, P.J.: programming with Constraints: An Introduction. MIT Press (1998)
16. Nieuwenhuis, R., Oliveras, A.: Dpll(t) with exhaustive theory propagation and its application to difference logic. In: In Proc. CAV05, volume 3576 of LNCS. Springer (2005)