Constraint Answer Set Programming Based on HEX-Programs*

Alessandro F. De Rosis[1], Thomas Eiter[2], Christoph Redl[2], and Francesco Ricca[1]

[1] Department of Mathematics and Computer Science, Università della Calabria
Via P. Bucci Cubo 31B, 87036 Rende (CS), Italy alessandrof.derosis@gmail.com
ricca@mat.unical.it
[2] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,redl}@kr.tuwien.ac.at

Abstract. Constraint Answer Set Programming (CASP) is a convenient integration of the Answer Set Programming (ASP) paradigm with constraint programming (CP), which was exploited for a range of applications. HEX-programs are another extension of ASP towards integration of arbitrary external sources through so-called external atoms. In this work, we integrate HEX-programs with CP, which results in a strict generalization of CASP, called *constraint* HEX-*programs*. We then present a translation of constraint HEX-programs to (standard) HEX-programs using special external sources for constraint solving. In contrast to native CASP solvers, this not only allows for reusing existing algorithms, but also for combining constraints with other external sources. We further show how to integrate advanced techniques used in CASP solving algorithms; while dedicated solvers are currently faster as not all such techniques have been integrated yet, our experiments show that this significantly improves efficiency and promises that further optimization integrated in future work will yield an efficient yet more general system.

Keywords: Answer Set Programming, Constraint Programming, External Sources

1 Introduction

In recent years, Answer Set Programming (ASP) has emerged as a popular approach to declarative problem solving for a range of applications [3], thanks to expressive and efficient systems like SMODELS [18], DLV [13], cmodels [12], and CLASP [10]. HEX-programs [7] extend ASP with external atoms, through which the user can couple any external data source with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions, into an external source which returns output values of an (abstract) function. This extension has been utilized for applications such as multi-context reasoning, and reasoning about actions and planning (cf. [5]). Notably, recursive data exchange between the rules and the external sources is supported.

Another declarative formalism is *constraint programming (CP)* [1]. Intuitively, a constraint satisfaction problem (CSP) consists of a set of variables from given domains

* This research was supported by the Austrian Science Fund (FWF) projects P24090 and P27730

and restrictions of their value combinations. Solutions to CSPs are assignments of values to the variables such that the given restrictions are respected.

The combination of ASP with CP is called Constraint Answer Set Programming (CASP), see [15, 14], as implemented e.g. in the CLINGCON system [17], which integrates GRINGO, CLASP and the constraint solver GECODE. While constraints might be encoded in plain ASP thanks to builtin predicates, this easily produces groundings of unmanageable sizes. Hence, a direct support of constraints within ASP is useful for avoiding this bottleneck. This is because instances of constraint variables can be hidden in the constraint solver without their explicit representation in the (ground) program. In addition to the *SMT-like* [2] approach for CASP solving as adopted e.g. by CLINGCON, there are also *translation approaches* (e.g. to SAT) [4]; however, as our host formalism of HEX-programs handles external sources similar to SMT, we stick with this former approach also for constraint handling.

Dedicated CASP solvers, however, do not allow for integration with background theories other than constraints. This motivates our work on integrating CASP with HEX-programs by translating the input programs into a native HEX-program. This allows for easy combination with arbitrary background theories without the need for adopting the reasoning algorithms, and results in a strict generalization of CASP, which we call *constraint* HEX-*programs*.

In more detail, after introducing the preliminaries in the Section 2, the further organization of the paper and our main contributions are as follows:

- We define an integration of HEX-programs with constraint programming and a translation to (standard) HEX-programs (Section 3). To this end, we develop an encoding based on a dedicated external atom which serves as an interface to a constraint solver. The truth values of the constraint atoms are guessed by disjunctive rules and passed as input parameters to the external atom, which checks the consistency wrt. the constraints.
- As this approach comes at the price of a large search space which undercuts efficient evaluation, we provide means for search space pruning by *conflict-driven learning* (Section 4). Whenever the constraint solver identifies an inconsistency, a reason for this conflict in terms of a small set of constraints is identified. This reason is added to the internal representation of the HEX-program to avoid invalid guesses in the further search.
- Extending a preliminary contribution [19], we provide means for *theory propagation* (Section 5). That is, reasons for conflicts are not only identified after inconsistencies occurred, but already guide the ASP solver before the interpretation is inconsistent.
- We present our implementation and consider some benchmarks (Section 6). Although it can currently not compete with native solvers (dedicated CASP algorithms are not yet fully integrated), the improvements can increase the efficiency significantly. We also discuss an application that uses other external sources and does not fit existing solvers.

In conclusion, the results of our ongoing work let us expect that by integrating and advancing CASP techniques into our system the gap to dedicated systems can be sensibly reduced (but not closed), at the gain of increased problem solving capacity.

2 Preliminaries

We start with basic definitions, syntax and semantics of HEX-programs and constraint programming. Our vocabulary consists of sets $\mathcal{C} \supseteq \mathbb{N}$, $\mathcal{V}$, $\mathcal{P}$ and $\mathcal{X}$ of constants, variables, predicates and external predicates, respectively, where $\mathbb{N}$ denotes the natural numbers. Constants $\mathcal{C}$ are also used as function symbols. The set of terms $\mathcal{T}$ is the least set $\mathcal{T} \supseteq \mathcal{C}$ s.t. whenever $f \in \mathcal{C}$ and $t_1, \ldots, t_\ell \in \mathcal{T}$, then $f(t_1, \ldots, t_\ell) \in \mathcal{T}$. Predicates $p \in \mathcal{P}$ may occur with multiple arities $ar(p) \subseteq \mathbb{N}$ in a program. While constants in our examples are usually strings which start with lower case letters, we formally allow them to consist of any characters except comma, parentheses and $\leftarrow$, as this is sufficient for unambiguous reading (e.g. c, but also $\leqslant$). We let predicates be alphanumeric strings starting with lower case letters (e.g. a, p), and variables be alphanumeric strings starting with upper case letters (e.g. Z). External predicates start with lower case letters preceded by character *&* in order to distinguish them from ordinary predicates (e.g. *&ext*).

For a list $\mathbf{x} = (x_1, \ldots, x_k)$, we write $x \in \mathbf{x}$ iff $x_i = x$ for some $1 \leq i \leq k$; the empty list is denoted ϵ. We drop the parentheses whenever this does not cause confusion. Moreover, for a list $\mathbf{x}$ we let x_i implicitly denote the i-th element even if the elements of $\mathbf{x}$ are not explicitly listed. We further use lists as sets when appropriate.

A *(signed) literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground atom of form $p(\mathbf{x})$, with predicate p and terms $\mathbf{x} = x_1, \ldots, x_\ell$. For a literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e., $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where a expresses that a is true and $\mathbf{F}a$ that a is false; an assignment $\mathbf{A}$ is called *complete for a program* (or *interpretation*), if $\mathbf{T}a \in \mathbf{A}$ or $\mathbf{F}a \in \mathbf{A}$ for all atoms a in the program.

A *nogood* is a set $\{L_1, \ldots, L_n\}$ of literals $L_i, 1 \leq i \leq n$. An interpretation $\mathbf{A}$ is a *solution* to a nogood δ (resp. a set Δ of nogoods), iff $\delta \not\subseteq \mathbf{A}$ (resp. $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$).

2.1 HEX-Programs

HEX-programs were introduced in [7] as a generalization of (disjunctive) extended logic programs under the answer set semantics [11]; for details and background see [7].

Syntax. HEX-programs extend ordinary ASP programs by *external atoms*, which enable a bidirectional interaction between a program and external sources of computation. External atoms have a list of input parameters (terms or predicate names) and a list of output parameters. Informally, to evaluate an external atom, the reasoner passes the terms and extensions of the predicates in the input tuple to the external source associated with the external atom. The external source computes output tuples that are matched with the output list. Formally, an *external atom* is of the form *&g* $[\mathbf{Y}]$ $(\mathbf{X})$, where $\mathbf{Y} = Y_1, \ldots, Y_k$ are input parameters from $\mathcal{T} \cup \mathcal{V} \cup \mathcal{P}$, and $\mathbf{X} = X_1, \ldots, X_l$ are output terms. Input $Y_i \in \mathbf{Y}$ is called *predicate input* if $Y_i \in \mathcal{P}$ and *term input* otherwise.

Definition 1 **(HEX-programs).** *A* HEX*-program consists of rules*

$$a_1 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \text{not } b_{m+1}, \ldots, \text{not } b_n \;, \qquad (1)$$

where each a_i is an (ordinary) atom $p(X_1, \ldots, X_\ell)$ with $X_i \in \mathcal{T} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, each b_j is either an ordinary atom or an external atom, and $k + n > 0$. [3]

The *head* of a rule r is $H(r) = \{a_1, \ldots, a_n\}$ and the *body* is $B(r) = \{b_1, \ldots, b_m,$ $\text{not } b_{m+1}, \ldots, \text{not } b_n\}$. We call b or $\text{not } b$ in a rule body a *default literal*; $B^+(r) =$

[3] Strong negation can be introduced as usual (use new atoms $\neg a$ and constraints $\leftarrow a, \neg a$).

$\{b_1, \dots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \dots, b_n\}$ is the *negative body*. For a rule r or a program Π, let $A(r)$ and $A(\Pi)$ be the set of all ordinary (i.e., non-external) atoms occurring in r or Π, respectively.

Semantics. We first define the semantics of ground (variable-free) HEX-programs. Intuitively, a ground external atom $\&g\,[\mathbf{y}]\,(\mathbf{x})$ is true, if the external source $\&g$ yields output tuple $\mathbf{x}$ when evaluated with input $\mathbf{y}$. Formally, the semantics of a ground external atom $\&g\,[\mathbf{y}]\,(\mathbf{x})$ wrt. an assignment $\mathbf{A}$ is given by a $1{+}k{+}l$-ary Boolean *oracle function* $f_{\&g}$ that is defined for all possible values of $\mathbf{A}$, $\mathbf{y}$ and $\mathbf{x}$, where k is the length of $\mathbf{y}$ and l is the length of $\mathbf{x}$. Thus, $\&g\,[\mathbf{y}]\,(\mathbf{x})$ is true relative to $\mathbf{A}$ if and only if $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1$. Satisfaction of ordinary ground rules and ground ASP programs [11] is extended to ground HEX-rules and programs in the obvious way. We assume that predicates, which do not occur in the input of an external atom, do not influence its semantics, i.e., $f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = f_{\&g}(\mathbf{A}', \mathbf{y}, \mathbf{x})$ whenever $\mathbf{A}$ and $\mathbf{A}'$ coincide on all atoms over $p \in \mathbf{y} \cap \mathcal{P}$.

Definition 2 (FLP-Reduct [9]). *For an interpretation $\mathbf{A}$ over a ground program Π, the* FLP-reduct $f\Pi^{\mathbf{A}}$ *of Π wrt. $\mathbf{A}$ is the set $\{r \in \Pi \mid \mathbf{A} \models b,$ for all $b \in B(r)\}$.*

An assignment $\mathbf{A}_1$ is smaller than or equal to another assignment $\mathbf{A}_2$ wrt. a program Π, denoted $\mathbf{A}_1 \leq_\Pi \mathbf{A}_2$, if $\{\mathbf{T}a \in \mathbf{A}_1 \mid a \in A(\Pi)\} \subseteq \{\mathbf{T}a \in \mathbf{A}_2 \mid a \in A(\Pi)\}$.

Definition 3 (Answer Set). *An answer set of ground program Π is a $\leq_\Pi$-minimal (complete) model $\mathbf{A}$ of $f\Pi^{\mathbf{A}}$; the set of all answer sets of Π is denoted $\mathcal{AS}(\Pi)$.*

Example 1. Consider the program $\Pi = \{p \leftarrow \&id\,[p]\,()\}$, where $\&id\,[p]\,()$ is true iff p is true. It has the answer set $\mathbf{A}_1 = \emptyset$, which is a $\leq_\Pi$-minimal model of $f\Pi^{\mathbf{A}_1} = \emptyset$. $\square$

As for possibly non-ground programs, let $grnd_{\mathcal{T}}(\Pi)$ be the ground program resulting from Π if the variables in Π are replaced by terms from $\mathcal{T}$ in all possible ways. The answer sets of a program Π are then given by $\mathcal{AS}(\Pi) = \mathcal{AS}(grnd_{\mathcal{T}}(\Pi))$. Safety criteria ensure that a finite subset of $grnd_{\mathcal{T}}(\Pi)$ suffices in practice [8].

Evaluation. Describing the evaluation algorithms for HEX-programs in detail is beyond this paper; we give here only an overview. The answer sets of a HEX-program Π are to be determined by the DLVHEX solver using a transformation to ordinary ASP programs as follows. Each ground external atom $\&g\,[\mathbf{y}]\,(\mathbf{x})$ in Π is replaced by an ordinary ground *external replacement atom* $e_{\&g[\mathbf{y}]}(\mathbf{x})$ and a rule $e_{\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow$ is added to the program. The answer sets of the resulting *guessing program* $\hat{\Pi}$ are computed with an ordinary ASP solver and projected to non-replacement atoms. However, the resulting interpretations are not necessarily models of Π, as the values of $\&g[\mathbf{y}](\mathbf{x})$ under $f_{\&g}$ and $e_{\&g[\mathbf{y}]}(\mathbf{x})$ can be different. Each answer set of $\hat{\Pi}$ thus merely yields a *candidate* which must be checked against the external sources. If no discrepancy is found, the candidate is a *compatible set* of Π. Finally, for each compatible set also minimality wrt. the reduct is checked; if true, then the compatible set yields an answer set. The ordinary ASP solver at the backend solves the problem by encoding the input program as a set of nogoods.

In addition to this procedure, additional nogoods are added during solving to describe parts of the behavior of external atoms in terms of input-output relationships [6]. Although not strictly necessary, it turned out to be highly effective in search space pruning as it eliminates guesses of external atoms that violate known behavior in advance.

2.2 Constraint programming

Basic formalism. Constraint programming is a declarative programming paradigm where variables are assigned values from domains, and the allowed combinations of variable values are restricted by *constraints*.

Definition 4 (Constraint Satisfaction Problem). *A* constraint satisfaction problem (CSP) *is a triplet $\langle V, D, C\rangle$, where V is a set of variables, D is a domain, and C is a set of constraints of form $\langle X, R\rangle$ where $X \in V^k$ and $R \subseteq D^k$ for some $k > 0$.*

Definition 5 (Solutions to Constraint Satisfaction Problems). *For a CSP $\langle V, D, C\rangle$, an* interpretation *is a mapping $I \colon V \mapsto D$, and a* solution *is an interpretation I such that for all $\langle \mathbf{v}, R\rangle \in C$ with $\mathbf{v} = v_1, \ldots, v_n$ it holds that $(I(v_1), \ldots, I(v_n)) \in R$.*

A CSP is called *consistent*, if it has a solution and *inconsistent* otherwise. Note that we assume all variables in V have the same domain D; this is for the sake of simplicity, indeed w.l.o.g. different domains for the variables can be simulated by appropriate constraints. Moreover, in the following we assume to have a fixed domain D and only specify it explicitly when relevant in examples.

Example 2. Consider the CSP $\langle V, D, C\rangle$ where $V = \{x, y\}$, $D = \{1, \ldots, 10\}$, and $C = \{c_1 \colon \langle (x, y), \{(1, 8), (2, 9), (2, 10)\}\rangle, c_2 \colon \langle y, \{9, 10\}\rangle, c_3 \colon \langle x, \{1, 2\}\rangle, c_4 \colon \langle x, \{4, 5\}\rangle\}$ One can easily see that the CSP is inconsistent due to c_3 and c_4. □

Constraint expressions. We now introduce arithmetic expressions to denote sets of value combinations and provide an interpretation of such expressions as by Definition 4.

We start with a set of *constraint terms* $\mathcal{CT}$, which are (possibly non-ground) atoms $p(X_1, \ldots, X_n)$ with $p \in \mathcal{P}$ and $X_i \in \mathcal{V} \cup \mathcal{T}$ for $1 \leq i \leq n$, variables $\mathcal{V}$ or terms $\mathcal{T}$, e.g. $p(c)$, $p(X)$, Y and 10; variables allow for exploiting the grounder when writing constraint expressions. The set of *constraint operators* consists of *arithmetic operators* $\mathcal{CA} = \{+, -, *, /\}$ and *comparison operators* $\mathcal{CC} = \{\equiv, \not\equiv, >, \geqslant, \leqslant, <\}$.[4]

The set of *arithmetic expressions* is $\mathcal{CE} = \{o_1 \circ_1 o_2 \circ_2 \cdots \circ_{n-1} o_n \mid o_1, \ldots, o_n \in \mathcal{CT} \cup \mathbb{N}, \circ_1, \ldots, \circ_{n-1} \in \mathcal{CA}\} \cup \{sum(p, i) \mid p \in \mathcal{P}, i \in \mathbb{N}\}$; expressions of kind $sum(p, i)$ are useful for the integration with logic programming, as they allow to query program predicates in constraint expressions. We call an arithmetic expression *ground*, if it does not contain variables from $\mathcal{V}$. Every ground constraint term is either a string (viewed as a constraint variable) or an integer (viewed as a numeric constant). The set of all *constraint expressions* is $\mathcal{CO} = \{l \circ r \mid l, r \in \mathcal{CE}, \circ \in \mathcal{CC}\}$; we call a constraint expression *ground*, if both l and r are ground. For a constraint expression $e \in \mathcal{CO}$ of form $e = l \circ r$, let $\overline{e} = l \overline{\circ} r$ be the negated constraint atom, where $\overline{\circ}$ is the negation of an operator $\circ$ such that $\overline{\equiv} = \not\equiv$, $\overline{>} = \leqslant$, $\overline{\geqslant} = <$, etc.

We now provide a translation of ground constraint expressions $e \in \mathcal{CO}$ to constraints as by Definition 4. Once the domain D is fixed, we can use a set of ground constraint expressions $E \subseteq \mathcal{CO}$ to define a CSP of kind $\langle V, D, C\rangle$, where V are the variables occurring in E, and the constraints C are constructed as follows.

[4] In the implementation, operators in constraint expressions are prefixed with '\$' to distinguish them from builtin predicates. As we do not use builtins in this paper, we drop '\$' for simplicity.

Definition 6. *For a ground $e \in \mathcal{CO}$ with the lexicographically ordered list of distinct constraint variables $\mathbf{v}$ of length k, and interpretation $\mathbf{A}$, the* constraint given by e and $\mathbf{A}$ *is $\Gamma(e, \mathbf{A}) = \langle \mathbf{v}, \{\mathbf{d} \in D^k \mid \gamma(e, \mathbf{d})$ holds$\}\rangle$, where $\gamma(e, \mathbf{d})$ results from e when:*

- *all constraint variables v_i with $1 \le i \le k$ are replaced by d_i;*
- *operators from $\mathcal{CA}$ and $\mathcal{CC}$ have the standard semantics;*
- *all expressions of form $sum(p, i)$ are replaced by $\sum_{\{\mathbf{T}p(x_1,\dots,x_n)\in\mathbf{A}\}} x_i$.*

For $E = \{e_1, \dots, e_n\} \subseteq \mathcal{CO}$, let $\Gamma(E, \mathbf{A}) = \langle \bigcup_{1\le i\le n} V_i, D, \bigcup_{1\le i\le n} R_i\rangle$ be the CSP given by D, E and $\mathbf{A}$, *where $\langle V_i, R_i\rangle = \Gamma(e_i, \mathbf{A})$ for all $1 \le i \le n$.*

Intuitively, $\mathbf{A}$ serves to interpret constraint terms $sum(p, i)$; this prepares for the integration with HEX-programs below. If e resp. E contains no constraint terms $sum(p, i)$, functions γ and Γ are independent of $\mathbf{A}$ and we might drop it from the argument list.

Example 3. Let $D = \{0, \dots, 3\}$ be the domain. The constraint expression $e_1 = x+y < 3$ gives the constraint $\Gamma(e_1) = \langle(x, y), \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (2, 0)\}\rangle$. Let $\mathbf{A} = \{\mathbf{T}p(1), \mathbf{T}p(2)\}$, the expression $e_2 = x + y > sum(p, 1)$ gives $\Gamma(e_2, \mathbf{A}) = \langle(x, y), \{(1, 3), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}\rangle$, i.e., $sum(p, 1)$ is replaced by 3. $\Box$

3 Integrating HEX-Programs and Constraint Programming

We now introduce an integration of HEX-programs with constraint programming, called *constraint* HEX-*programs*.[5] As HEX-programs subsume ASP-programs, this is a strict generalization of the integration of ASP and CP. Afterwards, we provide a translation of constraint HEX-programs to standard HEX-programs using a dedicated external atom for interfacing the constraint solver, which allows for reusing existing evaluation algorithms.

3.1 Constraint HEX-Programs

We now extend HEX-programs as by Definition 1 to constraint HEX-programs by allowing constraint expressions to be used as atoms; we sometimes call HEX-programs without constraint expressions *standard* HEX-*programs* to stress the absence of such atoms.

Syntax. We first extend Definition 1 to capture constraint expressions.

Definition 7 (Constraint HEX-programs). *A constraint* HEX-*program consists of rules*

$$a_1 \vee \dots \vee a_k \leftarrow b_1, \dots, b_m, \mathrm{not}\ b_{m+1}, \dots, \mathrm{not}\ b_n\ , \tag{2}$$

where each a_i is an (ordinary) atom or a constraint expression, each b_j is either an ordinary atom, an external atom or a constraint expression, and $k + n > 0$. A HEX-*program may define the domain for constraint variables; default it is the set of integers.*

We assume w.l.o.g. that default-negated constraint atoms $\mathrm{not}\, b$ in rule bodies are rewritten to positive atoms $\bar{b}$. Constraint expressions occurring within programs are also called *constraint atoms*. The sets $H(r)$, $B(r)$, $B^+(r)$ and $B^-(r)$ are defined akin to

[5] The content of this section is based on the preliminary work done in [19], which was a first less thorough step toward the combination of HEX-programs with constraint programming.

HEX-programs. As before, $A(r)$ and $A(\Pi)$ are the sets of ordinary atoms in a rule and in a program, respectively; they include neither external atoms nor constraint atoms.

Semantics. We extend the definition of interpretations to encompass constraint atoms. A *constraint assignment* is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where a is an ordinary ground atom or a ground constraint atom from $\mathcal{CO}$. Satisfaction of rules and answer sets are then extended from HEX-programs to constraint HEX-programs in the obvious way.

Definition 8. *Let Π be a ground constraint* HEX*-program. A constraint interpretation $\mathbf{A}$ is a* constraint answer set *of Π, if it is an answer set of Π and the CSP $\Gamma(E, \mathbf{A})$ is satisfiable, where $E = \{e \in \mathcal{CO} \mid \mathbf{T}e \in \mathbf{A}\} \cup \{\overline{e} \in \mathcal{CO} \mid \mathbf{F}e \in \mathbf{A}\}$.*

The set of all constraint answer sets of a constraint HEX-program Π is denoted $\mathcal{CAS}(\Pi)$. The answer sets $\mathcal{AS}(\Pi)$ of a non-ground constraint HEX-program are given by the answer sets $\mathcal{AS}(grnd_{\mathcal{T}}(\Pi))$ of $grnd_{\mathcal{T}}(\Pi)$ obtained by substituting variables by terms in all possible ways; the substitution includes variables in constraint expressions. Intuitively, the CSP consists of the constraints represented by the constraint atoms.

Example 4. Suppose Alice owns a restaurant offering different daily menus. The menus are chosen on the basis of the prices of foods and drinks such that the price of the food should be greater than the price of the drink. The prices of all menus should be no greater than 20; menus with a price equal to 20 are called *exclusive*.

This can be encoded by the following program Π with domain $\mathbb{N}$.

$r_1\colon$ *food*$(P) \leftarrow$ *&sql* ["Select price from Food"] (P)

$r_2\colon$ *drink*$(P) \leftarrow$ *&sql* ["Select price from Drink"] (P)

$r_3\colon max\_price(20)$

$r_4\colon inMenu(F, D) \vee outMenu(F, D) \leftarrow drink(D), food(F)$

$r_5\colon \leftarrow D > F, inMenu(F, D)$

$r_6\colon F + D \leqslant P \leftarrow inMenu(F, D), max\_price(P)$

$r_7\colon exclusive\_menu \leftarrow inMenu(F, D), max\_price(P), F + D \equiv P$

Here, the prices of food and drink are represented by atoms $food(\cdot)$ and $drink(\cdot)$, respectively. We use an external atom *&sql* $[\cdot]$ $(\cdot)$ in order to load all prices from the database of the restaurant. Rule r_4 creates all possible combinations of available menus. Rule r_5 enforces the price of the food to be greater than the prices of the drink. Rule r_6 implements the maximum price and rule r_7 checks the existence of an exclusive menu.

Let the facts be $F = \{max\_price(20), food(30), food(10), drink(8)\}$, the unique constraint answer set of the program contains, besides facts, $\mathbf{T}inMenu(10, 8)$, and $\mathbf{T}outMenu(30, 8)$ as positive atoms, and there is no exclusive menu. □

3.2 Encoding Constraint HEX-Programs in Standard HEX-Programs

We now provide a translation to standard HEX-programs. The basic idea is to encode constraint atoms by ordinary atoms whose truth values are guessed. For each guess, the atoms representing constraints serve as input to an external atom that checks the consistency of the CSP encoded by constraint expressions by employing a constraint solver. To encode constraint atoms from $\mathcal{CO}$ as ordinary atoms, their operators and operands are stored as individual arguments of the ordinary atoms. Since constraint atoms may contain ASP variables, the grounding component of the HEX-solver is reused for generating ground constraint atoms. In the following, let con be a new predicate.

Definition 9. *For $E = a_1 \circ_1 \cdots \circ_{n-1} a_n \in \mathcal{CO}$ with $a_i \in \mathcal{CE}$ for $1 \leq i \leq n$, $\circ_j \in \mathcal{CC}$ for exactly one $1 \leq j < n$, and $\circ_i \in \mathcal{CA}$ for all $1 \leq i < n, i \neq j$, let $toAtom(E) = con(a_1, \circ_1, \ldots, \circ_{n-1}, a_n)$. For ordinary or external atoms A, let $toAtom(A) = A$.*

Note that all components of the constraint expression occur as separate terms in the rewritten atom, which is possible as operators such as $\equiv$ or $\leqslant$ can be used as constants (cf. Section 2). Further note that the result of $toAtom$ might result in a function term, e.g., $e = p(X) > 2$ is translated to $toAtom(e) = con(p(X), >, 2)$. That is, the original atom $p(X)$ is encoded as a function term with function symbol p and argument X, which is possible as sets $\mathcal{P}$ and C are not necessarily disjoint. For a rule r let $toAtom(r)$ be result of atom-wise application of $toAtom$ to all atoms in r. Importantly, also the inverse function $toAtom^{-1}$ exists as for any atom of kind $con(\mathbf{x})$, $toAtom^{-1}(con(\mathbf{x}))$ only needs to concatenate the elements of $\mathbf{x}$, while for atoms over predicates different from con it is the identity function. For an assignment $\mathbf{A}$ let $toAtom^{-1}(\mathbf{A}) = \{toAtom^{-1}(con(\mathbf{x})) \mid \mathbf{T}con(\mathbf{x}) \in \mathbf{A}\}$ be the set of constraint expressions encoded as ordinary atoms in $\mathbf{A}$.

Towards an encoding of constraint HEX-programs in standard HEX-programs, we introduce a function which generates guessing rules for all constraints in a given rule r.

Definition 10. *Let the* guessing rules *guess(r) for the constraint atoms in rule r be: $guess(r) = \{toAtom(e) \vee toAtom(\overline{e}) \leftarrow B(r) \setminus \mathcal{CO} \mid$ for all $e \in \mathcal{CO} \cap (H(r) \cup B(r))\}$.*

Intuitively, for a rule r, rule $guess(r)$ guesses the truth value of constraint atoms in r, where the (possibly) non-empty body guarantees safety.

Moreover, the sums of predicate arguments used in sum expressions are computed using auxiliary predicates. The idea is to collect all required sums (occurring in some constraint expression) in a new predicates sum, whose parameters are the predicate and argument position to be summed up, and the value of this sum.

Definition 11. *For a constraint HEX-program Π, let $sumdef(\Pi) = \{sum(p, i, X_i) \leftarrow p(X_1, \ldots, X_\ell) \mid$ for all $sum(p, i) \in \mathcal{CA}$ occurring in $\Pi, \ell \in ar(p), \ell \geq i\}$.*

Finally, we define an external source that performs the consistency check of the guessed values of the (rewritten) constraint expressions in the program. For the sake of simplicity we do not communicate the values of constraint variables back to the HEX-program, but an according extension would be possible by adding output values.

Definition 12. *For a constraint interpretation $\mathbf{A}$ over Π, let $f_{\&check}(\mathbf{A}, con, sum, \epsilon) = 1$ ($= 0$) if the CSP $\Gamma(toAtom^{-1}(\mathbf{A}), \mathbf{A})$ has a (has no) solution.*

Intuitively, the external atom internally recovers the underlying CSP given by the atoms over con in $\mathbf{A}$ and checks its consistency; the constraint expressions are retrieved by function $toAtom^{-1}$ and directly define a CSP using function Γ from Definition 6.

Now we can define a translation of a constraint HEX-program, which consists of the original program with constraint atoms replaced by ordinary ones, the guessing rules for constraint atoms, the computation of the sums, and the consistency check.

Definition 13. *Given a constraint* HEX*-program Π, we define:*

$$\begin{aligned}\mathit{translation}(\Pi) = \{\mathit{toAtom}(r) \mid r \in \Pi\} \cup \{\mathit{guess}(r) \mid r \in \Pi\} \cup \\ \mathit{sumdef}(\Pi) \cup \{\leftarrow \text{not } \&\mathit{check}\,[\mathit{con}, \mathit{sum}]\,()\}\end{aligned}$$

Example 5. Consider the previous constraint HEX-program Π from Example 4 with constraint domain $\mathbb{N}$. We now show how $\mathit{translate}(\Pi)$ is constructed. The set $\{\mathit{toAtom}(r) \mid r \in \Pi\}$ made by ordinary atoms that represent constraint atoms is as follows:

$r_1\colon \mathit{food}(P) \leftarrow \&\mathit{sql}\,[\text{“Select price from Food”}]\,(P)$
$r_2\colon \mathit{drink}(P) \leftarrow \&\mathit{sql}\,[\text{“Select price from Drink”}]\,(P)$
$r_3\colon \mathit{max\_price}(20)$
$r_4\colon \mathit{inMenu}(F,D) \vee \mathit{outMenu}(F,D) \leftarrow \mathit{drink}(D), \mathit{food}(F)$
$r_5'\colon \leftarrow \mathit{con}(D,>,F), \mathit{inMenu}(F,D)$
$r_6'\colon \mathit{con}(F,+,D,\leqslant,P) \leftarrow \mathit{inMenu}(F,D), \mathit{max\_price}(P)$
$r_7'\colon \mathit{exclusive\_menu} \leftarrow \mathit{inMenu}(F,D), \mathit{max\_price}(P), \mathit{con}(F,+,D,\equiv,P)$

Next, the set of guessing rules $\{g_5, g_6, g_7\} = \{\mathit{guess}(r) \mid r \in \Pi\}$, stemming from the constraint atoms in r_5, r_6 and r_7, respectively, is as follows:

$g_5\colon \mathit{con}(D,>,F) \;\vee\; \mathit{con}(D,\leqslant,F) \leftarrow \mathit{inMenu}(F,D)$
$g_6\colon \mathit{con}(F,+,D,\leqslant,P) \vee \mathit{con}(F,+,D,>,P) \leftarrow \mathit{inMenu}(F,D), \mathit{max\_price}(P)$
$g_7\colon \mathit{con}(F,+,D,\equiv,P) \vee \mathit{con}(F,+,D,\not\equiv,P) \leftarrow \mathit{inMenu}(F,D), \mathit{max\_price}(P)$

We do not have any sum constraint expressions, thus $\mathit{sumdef}(\Pi) = \emptyset$.
Finally, rule $\leftarrow \text{not } \&\mathit{check}\,[\mathit{con}, \mathit{sum}]\,()$ is added for consistency checking. □

The following proposition states a one-to-one correspondence between the constraint answer sets of a constraint HEX-program Π and the answer sets of the HEX-program $\mathit{translation}(\Pi)$; thus HEX-solvers may be used for constraint HEX-program solving.

Proposition 1. *Given a constraint* HEX*-program Π, let $\Pi' = \mathit{translation}(\Pi)$. Then:*

1. *If $\mathbf{A}' \in \mathcal{AS}(\Pi')$, then $\mathbf{A} \in \mathcal{CAS}(\Pi)$ for $\mathbf{A} = \mathbf{A}' \cap \{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi)\}$.*
2. *If $\mathbf{A} \in \mathcal{CAS}(\Pi)$, then $\mathbf{A} = \mathbf{A}' \cap \{\mathbf{T}a, \mathbf{F}a \mid a \in A(\Pi)\}$ for some $\mathbf{A}' \in \mathcal{AS}(\Pi')$.*

4 Conflict-Driven Learning

Modern SAT and ASP solvers store the instance as nogoods (cf. Section 2) and construct an assignment which satisfies them all. The basic technique is *unit propagation*: whenever all but one literals of a nogood are true, the last one is set to false. If this does not derive further values, a guess is made. However, many guesses lead to inconsistency because of similar reasons. Therefore, *conflict-driven learning* was introduced to derive additional nogoods from conflicts [16, 10] which describe its *initial* reason, avoid this reason in the further search, and potentially allow for backtracking several decisions.

The technique was extended to external sources, where external atoms might add nogoods which describe (parts of) their behavior [6]. For example, let $\&\mathit{diff}\,[p,q]\,(X)$ compute the difference of the extensions of p and q and $\mathbf{A} = \{\mathbf{T}p(a), \mathbf{T}p(b), \mathbf{T}q(a)\mathbf{F}q(b)\}$.

Then $f_{\&diff}(\mathbf{A}, p, q, b) = 1$ and $f_{\&diff}(\mathbf{A}, p, q, x) = 0$ for $x \neq b$. The nogood $\{\mathbf{T}p(b), \mathbf{F}q(b), \mathbf{F}e_{diff[p,q]}(b)\}$ might be added to express that the output contains b whenever $p(b)$ is true and $q(b)$ is false. It eliminates guesses in the further search which violate known behavior. We adopt the technique to constraint HEX-programs by learning techniques for the external atom *&check*, which encode reasons for inconsistencies wrt. constraints.

4.1 Irreducibility

Whenever the external atom *&check* $[con, sum]$ () is unsatisfied, the set of all constraint expressions in the assignment $\mathbf{A}$ is inconsistent. This set is a valid explanation for the conflict, i.e., $\{\mathbf{T}e_{\&check[con,sum]}()\} \cup \{\mathbf{T}con(\mathbf{x}) \in \mathbf{A}\} \cup \{\sigma sum(p, i) \in \mathbf{A} \mid \sigma \in \{\mathbf{T}, \mathbf{F}\}\}$ could serve as a nogood. However, in many cases not all constraint expressions are relevant. In order to describes the reason effectively, we rather compute a *subset-minimal* set of atoms to represent conflicting constraint expressions.

Definition 14 (Irreducible Inconsistent Set (IIS)). *An inconsistent set C of constraints is an* irreducible inconsistent set (IIS) *[17], if every subset $C' \subsetneq C$ is consistent.*

Example 6. Consider the inconsistent set of constraints C in Example 2. One can show that after removing c_1, set C is still inconsistent and therefore it is not an IIS. In contrast, $C' = \{c_3, c_4\}$ is an IIS as the removal of at least one constraint makes it consistent. $\square$

The definition carries over to constraint expressions. For an interpretation $\mathbf{A}$ we call $E \subseteq \mathcal{CO}$ an IIS if $\Gamma(E, \mathbf{A})$ is inconsistent but $\Gamma(E', \mathbf{A})$ is consistent for any $E' \subsetneq E$.

Example 7. The set $E = \{x < y, y < x\}$ of constraint expressions is an IIS (for any domain) as it is inconsistent but removal of at least one element makes it consistent. $\square$

Forward filtering. The algorithm constructs the IIS iteratively beginning from the empty set, cf. Algorithm 1. In each iteration, a constraint expression which is involved in the inconsistency is identified and added to the constructed IIS. The process is repeated until the current set of constraints becomes inconsistent, i.e., an IIS has been found.

Proposition 2. *For an inconsistent set $E \subseteq \mathcal{CO}$, the ForwardFiltering(E) is an IIS.*

4.2 Nogood Learning

We now present nogoods which can be learned from IISs. The input to the nogood learning function Λ shown in the following definition needs to be an IIS of constraint expressions, but no specific construction method for an IIS is assumed. Thus, the IIS might be computed by the algorithm shown above or in the literature, e.g., [17].

Definition 15. *Let $\mathbf{A}$ be an interpretation over $translate(\Pi)$ where Π is a constraint* HEX-*program. We define* the nogood learned from an IIS *$E \subseteq \mathcal{CO}$ as follows:*

$$\begin{aligned}\Lambda(\mathbf{A}, E) = &\{\mathbf{T}e_{\&check[con,sum]}()\} \cup \{\mathbf{T}toAtom(e) \mid e \in E\} \cup \\ &\{\sigma sum(p, i, v) \in \mathbf{A} \mid sum(p, i) \textit{ occurs in } E, \sigma \in \{\mathbf{T}, \mathbf{F}\}\}\end{aligned}$$

Algorithm 1: ForwardFiltering

```
Data: Inconsistent set $E \subseteq \mathcal{CO}$
Result: IIS $E' \subseteq E$
$E' \leftarrow \emptyset$
while $E'$ is consistent do
    $T \leftarrow E'$
    for $c \in E$ do
        $T \leftarrow T \cup \{c\}$
        if $T$ is inconsistent then
            $E' \leftarrow E' \cup \{c\}$
            break
return $E'$
```

Algorithm 2: TheoryPropagation

```
Data: Constraint expressions $E$ and assigned ones $E_A \subseteq E$
Result: IIS $E'$ s.t. $E_A \subseteq E' \subseteq E$ or $\perp$ if none exists
$E_U \leftarrow E \setminus E_A$
for $u \in E_U$ do
    $E' \leftarrow E_A$
    if $E' \cup \{u\}$ is inconsistent then
        return IIS over $E' \cup \{u\}$ (cf. Section 4)
return $\perp$
```

Proposition 3. *For a constraint* HEX*-program Π, any answer set of $translation(\Pi)$ is a solution to $\Lambda(\mathbf{A}, E)$ for all assignments $\mathbf{A}$ and IISs $E \subseteq \mathcal{CO}$.*

Intuitively, $\Lambda(\mathbf{A}, E)$ expresses that the IIS forms a reason why $e_{\&check[con,sum]}()$ is false for the current values of all $sum(p, i)$. Such nogoods can be added as any guess eliminated by such nogoods would be rejected when $\&check$ is evaluated anyway.

5 Theory Propagation

We now exploit the constraint theory for the search before the interpretation is inconsistent by presenting the current partial assignment to the external source whenever internal propagation cannot infer further truth values; the source might add nogoods [6]. For a constraint HEX-program Π and a (possibly partial) assignment $\mathbf{A}$ over $translation(\Pi)$, one can extract the set $E_A = toAtom^{-1}(\mathbf{A})$ of constraint expressions assumed to be true. For the set of all constraint expressions E appearing in Π, Algorithm 2 computes an IIS $E' \subseteq E$. Since E_A is not necessarily inconsistent, the algorithm might include yet unassigned constraint expressions, i.e., we might have $E' \nsubseteq E_A$. Adding nogoods for such IISs possibly implies yet unassigned atoms and, thus, avoid wrong guesses.

Proposition 4. *Whenever Algorithm 2 returns $E \neq \perp$, then E is an IIS.*

Example 8. Consider the set of constraint expression $E = \{e_1\colon x + y < 3, e_2\colon x \leqslant 1, e_3\colon y > 3, e_4\colon z > 2\}$, and suppose the algorithm starts with the set $E_A = \{e_1\}$ of already assigned constraint expressions. As it is consistent, it adds e_2, which is still consistent. It then adds e_3, resulting in the inconsistent set $\{e_1, e_3\}$, which is also an IIS. (In general one can reduce the set resulting from this process to an IIS, cf. Section 4.) □

Although the IIS might contain constraint expressions whose truth values are yet unknown, it can still be fed to function Λ to learn nogoods in advance. Formally:

Proposition 5. *For a constraint* HEX*-program Π with constraint expressions E, $\Pi' = translation(\Pi)$ and a possibly partial assignment $\mathbf{A}$ over Π', any answer set of Π' is a solution to $\Lambda(\mathbf{A}, E')$ for $E' = TheoryPropagation(E, toAtom^{-1}(\mathbf{A}))$ if $E' \neq \perp$.*

size	Const. HEX w/o Th. Prop.			Const. HEX with Th. Prop.			CLINGCON		
	back	for	range	back	for	range	back	for	range
1	0.08	0.08	0.07	0.08	0.07	0.08	0.02	0.02	0.02
3	0.13	0.12	0.35	0.22	0.12	0.32	0.03	0.03	0.03
5	—	—	—	15.79	15.63	15.57	2.51	2.63	2.64
7	—	—	—	43.89	43.96	43.99	9.27	9.56	9.29
9	—	222.99	—	40.74	40.62	40.50	7.78	7.85	7.41
11	—	—	—	49.44	49.59	49.39	10.56	11.20	9.30
13	—	—	—	32.22	31.87	31.84	5.69	5.55	5.71

Table 1: Worker Skill Benchmark Results

6 Implementation and Evaluation

For implementing our technique, we integrated GRINGO and CLASP as backends into our prototype system DLVHEX. We then exploited the external source interface for implementing the external atom *&check* for consistency checking wrt. constraint theories. The implementation is provided as a plugin to the reasoner and uses GECODE as constraint solver backend (the implementation is flexible such that interfaces to other constraint solvers could be easily added). The plugin further supports theory propagation as discussed in the previous section. Moreover, thanks to the possibility to extend the input language, the constraint plugin supports constraint HEX-programs as in Definition 7 and transparently rewrites it to a standard HEX-program prior to evaluation.

Benchmarks. We evaluated the implementation on a Linux server with two 12-core AMD 6176 SE CPUs with 128GB RAM using a couple of benchmarks (see https://github.com/hexhex/caspplugin), which have been considered in the preliminary thesis work [19]. We compare our implementation without and with theory propagation using different nogood learning techniques. We further compare our implementation to the CLINGCON system. The timeout was set to 300 seconds per instance (denoted —).

Worker skill. In the following scheduling problem, we consider tasks to be assigned to workers, such that the skills of the worker are not smaller than the difficulty of the task (difficulty and the skills are expressed by integers).

The results are shown in Table 1. While the CLINGCON system is faster than our system, theory propagation reduced significantly the performance gap. Without theory propagation, most instances are not solvable within the timeout. This is because learned nogoods are added much less frequently since the assignment needs to be completed before the external source is triggered, so the reasoner spends a significant amount of time completing assignments that might be actually already conflicting.

Packing. This problem consists of a set of squares of given sizes which are to be placed in a rectangular area of a given size such that no squares intersect. It was taken from the ASP Competition 2011 but restated as a constraint HEX-program.

The results are shown in Table 2. Also here, CLINGCON is faster. A closer analysis revealed that theory propagation is applied only very few times at the beginning of the search with no visible effect. This is because the problem features only constraint atoms in disjunctive rule heads and the instances are under constrained, thus constraint atoms are mostly subject to choices upon either a solution is found or some nogood is learned.

size	Const. HEX w/o Th. Prop.		Const. HEX with Th. Prop.		CLINGCON	
	for	back	for	back	for	back
1	4.44	4.26	4.40	4.24	0.03	0.03
3	5.40	4.94	5.47	4.65	0.02	0.02
5	6.50	6.09	6.73	5.80	0.02	0.03
7	7.00	5.41	7.48	6.14	0.03	0.02
9	6.69	5.14	6.36	4.94	0.03	0.02
11	7.26	5.33	5.65	5.44	0.02	0.02
13	6.15	4.99	6.73	5.84	0.02	0.02

Table 2: Packing Benchmark Results

size	Const. HEX w/o Th. Prop.			Const. HEX with Th. Prop.			CLINGCON		
	back	for	range	back	for	range	back	for	range
1	0.08	0.08	0.07	0.07	0.07	0.07	0.02	0.02	0.01
3	0.20	0.13	1.00	0.13	0.14	0.36	0.03	0.02	0.02
5	0.29	0.32	0.29	0.34	0.41	0.33	0.04	0.04	0.04
7	—	—	—	1.38	1.38	1.39	0.04	0.04	0.03
9	—	—	—	1.73	1.74	1.63	0.05	0.06	0.05
11	—	—	—	2.91	2.90	2.57	0.06	0.07	0.07
13	—	—	—	4.10	4.29	4.07	0.08	0.09	0.08

Table 3: Reachability Benchmark Results

Reachability. In this problem a set of cars with limited reachability is to be assigned to destinations in a graph, such that no car is assigned to a node which exceeds its limit.

The results are shown in Table 3. Without the theory propagation only the simplest instances can be solved, while theory propagation allows for scaling up to a larger size. We observed that theory propagation, in this problem, is never applied on the easiest instances (as they are solved after a few choices). However, starting from instances of size 7, the first calls to theory propagation, which occur at the beginning of the search, are very effective in cutting the search space. The strategy without theory propagation cannot benefit from this initial pruning, and since also nogood learning from the constraints becomes more rare, no solution is found within the timeout.

Summary. While all benchmarks show that our implementation cannot yet compete with CLINGCON, we already achieved a significant improvement compared to the implementation without theory propagation. The remaining gap is explained by a tighter coupling of the ASP solver and the theory solver. In fact, although thanks to theory propagation, the constraint solver is now queried for partial interpretations, each call still needs to investigate the updated partial interpretation from scratch to determine an IIS. In contrast, CLINGCON also exploits information about atoms which have changed or not since the previous call, which allows for more efficient computation of implied atoms. However, the theory propagation methods already show significant improvements, and it is expected that advanced technique such as in CLINGCON will further shrink the gap.

Integration with other External Sources – Application Scenario. In contrast to pure CASP, constraint HEX-programs allow for combining constraints and other external sources. We consider a company which hires employees and assigns them to departments based on their skills. Constraints limit the absolute salaries and salary gaps. The total set of employed applicants may influence the required skills due to administrative issues.

$$r_1\colon \mathit{worker}(W, \mathit{Sk}, \mathit{Sl}) \leftarrow \&\mathit{sql}\left[\begin{array}{l}\text{“Select id, skill, salary}\\ \quad\text{from Applicant where skill in }(\%1)\text{”}\end{array}, \mathit{req}\right](W, \mathit{Sk}, \mathit{Sl})$$

$$r_2\colon \mathit{dept}(D, U) \leftarrow \&\mathit{sql}\,[\text{“Select id, upperbound\_sal from Department”}]\,(D, U)$$

$$r_3\colon \mathit{worker\_dept}(W, D) \leftarrow \&\mathit{ontology}\,[\mathit{worker}, D]\,(W), \mathit{dept}(D, M)$$

$$r_4\colon \mathit{hire}(W, D, \mathit{Sk}, \mathit{Sl}) \vee \neg\mathit{hire}(W, D, \mathit{Sk}, \mathit{Sl}) \leftarrow \mathit{worker\_dept}(W, D), \mathit{worker}(W, \mathit{Sk}, \mathit{Sl})$$

$$r_5\colon \ \leftarrow \mathit{hire}(W, D_1, \mathit{Sk}, \mathit{Sl}), \mathit{hire}(W, D_2, \mathit{Sk}, \mathit{Sl}), D_1 \not\equiv D_2$$

$$r_6\colon \ \leftarrow \mathit{req}(\mathit{Sk}, N), \&\mathit{count}\,[\mathit{hire}, 3, \mathit{Sk}]\,(C), C \not\equiv N$$

$$r_7\colon U \geq \mathit{Sl} \leftarrow \mathit{hire}(W, D, \mathit{Sk}, \mathit{Sl}), \mathit{dept}(D, U)$$

$$r_8\colon \ \leftarrow \mathit{hire}(W_1, D, \mathit{Sk}_1, \mathit{Sl}_1), \mathit{hire}(W_2, D, \mathit{Sk}_2, \mathit{Sl}_2), \mathit{Sl}_1 > 15 * \mathit{Sl}_2$$

$$r_9\colon \mathit{req}(\mathit{hr\_accounting}, 1) \leftarrow \mathit{sum}(\mathit{hire}, 3) > 100000$$

We assume that facts of kind $\mathit{req}(\mathit{sk}, n)$ specify that n employees with skill sk are needed. Rule r_1 performs a pre-selection due to the possibly large number of applicants, and imports only those whose skills are required; here, $(\%1)$ refers to the skills represented by req. Rule r_2 imports departments and their upper salary bounds. Rule r_3 uses an ontology to infer possible departments at which a worker can be employed, based on his/her skills; for each atom $\mathit{worker}(w, \mathit{sk}, \mathit{sl})$ (e.g. $w = \mathit{joe}$, $\mathit{sk} = \mathit{programming}$, $\mathit{sl} = \mathit{2000}$), sk is interpreted as a concept in the ontology which is extended by individual w; this infers for each department D all workers W who might work there. Rules r_4 and r_5 guess all possible hirings of workers in at most one department; rule r_6 ensures that the requirements are satisfied, where $\&\mathit{count}\,[\mathit{hire}, 3, \mathit{Sk}]\,(C)$ yields the number of distinct atoms $\mathit{hire}(\cdot, \cdot, \mathit{Sk}, \cdot)$, i.e., of hired applicants with skill Sk. Rule r_7 enforces salaries to be within the department limit. Rule r_8 ensures that no employee can earn more than 15 times as much as any other employee in the department. Rule r_9 derives that a dedicated human resources accountant is needed, if the sum of all salaries exceeds a given limit. As external atoms are used cyclically rather than in sequence, the application cannot be simulated by manually calling dedicated systems for the various external sources.

7 Conclusion

We have presented *constraint* HEX*-programs*, which facilitate Constraint Answer Set Programming (CASP) on top of HEX-programs. As the latter extend ASP with access to arbitrary external sources, applications in constraint HEX-programs may exploit constraints and other background theories and can not be realized with CASP systems. Our approach uses generic algorithms developed for HEX-programs. The current implementation is not yet competitive with dedicated CASP systems wrt. efficiency, but the optimizations can improve performance significantly compared to the basic encoding.

CASP solvers realize more techniques than our solver [17], e.g. a tighter integration of theory propagation methods which exploit information about truth value updates, which we plan to integrate in future work. Also the integration of additional features, such as global constraints, are up to future work. Moreover, guesses of constraint atoms are not always necessary, e.g., when constraint expressions occur in heads of rules with satisfied bodies, then the value is enforced rather than queried. Finally, while the CASP resp. HEX-solver learns from the constraint solver, there is currently no information flow in the other direction, which would be interesting to investigate.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press, NY, USA (2003)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories, Frontiers in Artificial Intelligence and Applications, vol. 185, chap. 26, pp. 825–885. IOS Press (2009)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Comm. ACM 54(12), 92–103 (2011)
4. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. CoRR abs/1007.4114 (2010)
5. Eiter, T., Fink, M., Ianni, G., Krennwallner, T., Schüller, P.: Pushing efficient evaluation of HEX programs by modular decomposition. In: LPNMR. LNCS, vol. 6645, pp. 93–106 (2011)
6. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Conflict-driven ASP solving with external sources. TPLP 12, 659–679 (2012)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer-Set Programming. In: IJCAI. pp. 90–96. Professional Book Center (2005)
8. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning. In: ESWC. LNCS, vol. 4011, pp. 273–287. Springer (2006)
9. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
10. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187–188, 52–89 (2012)
11. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9(3–4), 365–386 (1991)
12. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning 36(4), 345–377 (2006)
13. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL 7(3), 499–562 (2006)
14. Lierler, Y.: Relating constraint answer set programming languages and algorithms. Artificial Intelligence 207, 1–22 (2014)
15. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Ann. Math. Artif. Intell. 53(1-4), 251–287 (2008)
16. Mitchell, D.G.: A SAT solver primer. EATCS Bulletin 85, 112–133 (2005)
17. Ostrowski, M., Schaub, T.: ASP modulo CSP: the clingcon system. TPLP 12, 485–503 (2012)
18. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence 138, 181–234 (2002)
19. Stashuk, O.: Integrating Constraint Programming into Answer Set Programming. Master's thesis, Vienna University of Technology, Knowledge-based Systems Group (Sept. 2013)