Answer Set Solving with Lazy Nogood Generation

CHRISTIAN DRESCHER and TOBY WALSH

NICTA and University of New South Wales

submitted 25 March 2012; revised 25 March 2012; accepted N/A

Abstract

Although Answer Set Programming (ASP) systems are highly optimised, their performance is sensitive to the size of problem encodings. We address this deficiency by introducing a new extension to ASP solving. The idea is to integrate external propagators to represent parts of the encoding implicitly, rather than generating it a-priori. To match the state-of-the-art in conflict-driven solving, however, external propagators can generate an encoding of their inference on demand. We demonstrate the applicability of our approach in a novel Constraint Answer Set Programming system that can seamlessly integrate Constraint Programming techniques without sacrificing the advantages of conflict-driven techniques. Experiments provide evidence for computational impact.

KEYWORDS: Answer Set Programming, Conflict-Driven Nogood Learning, Constraint Propagation, Lazy Nogood Generation

1 Introduction

Developing a powerful paradigm for declarative problem solving is one of the key challenges in the area of knowledge representation and reasoning. A promising candidate is answer set programming (ASP; Baral 2003) which builds on logic programming and non-monotonic reasoning. Its success depends on two factors: efficiency of the solving capacities, and modelling convenience. Efficient ASP solvers (cf. Lin and Zhao 2002, Giunchiglia et al. 2006, Gebser et al. 2007) match the state-of-the-art in conflict-driven solving (Mitchell 2005), including conflict-driven learning, lookback-based heuristics, and backjumping. However, their performance is sensitive to the size of problem encodings which can quickly become infeasible, for instance, through the worst-case exponential number of loops in a logic program (Lifschitz and Razborov 2006), or constructs that are naturally non-propositional, like constraints over finite domains. A variety of extensions to ASP have been proposed that deal with some of these issues via other declarative problem solving paradigms, similar to the idea of Satisfiability Modulo Theories (SMT, Nieuwenhuis et al. 2006). Recently, for example, we have witnessed the development of Constraint Answer Set Programming (CASP) that integrates Constraint Programming (CP) with ASP, supporting constraints over finite domains, and most importantly, global constraints. While this approach certainly increases modelling convenience and can drastically decrease the size of an encoding, it does not fully carry over to conflict-driven solving technology (cf. Drescher and Walsh 2010).

We address this problem and present a new computational extension to ASP solving, called Lazy Nogood Generation. Motivated by the success of Lazy Clause Generation (Ohrimenko et al. 2009) in Constraint Satisfaction Problem (CSP) solving, the key idea is to generate (parts of) the problem encoding on demand, only when new information can be propagated. We make several contributions to the study of Lazy Nogood Generation in ASP. First, we lay the foundations of external propagation based on a uniform characterisation of answer sets in terms of nogoods. This provides the underpinnings to represent conditions on the answer sets of a logic program without encoding the entire problem a-priori. However, external propagators can make parts of the encoding explicit, in particular, when they can trigger inference. As we shall see, our techniques generalise existing ones, e.g., loop formula propagation (Gebser et al. 2007), and weight constraint rule propagation (Gebser et al. 2009). Second, we specify a decision procedure for ASP solving with Lazy Nogood Generation. It is centred around conflict-driven solving and integrates external propagation. Third, we demonstrate applicability. We show how to seamlessly integrate constraint propagation with our framework, resulting in a novel approach to CASP solving. Finally, we empirically evaluate a prototypical implementation and compare to the state-of-the-art in ASP and CASP solving.

2 Background

Many tasks from the declarative problem solving domain can be defined as CSP, that is a tuple (V, D, C) where V is a finite set of *constraint variables*, each $v \in V$ has an associated finite *domain* $dom(v) \in D$, and C is a set of constraints. A *constraint* c is a k-ary relation, denoted $range(c)$, on the domains of the variables in $scope(c) \in V^k$. A *(constraint variable) assignment* is a function A that assigns to each variable $v \in V$ a value from $dom(v)$. For a constraint c with $scope(c) = (v_1, \dots, v_k)$ define $A(scope(c)) = (A(v_1), \dots, A(v_k))$. The constraint c *satisfied* if $A(scope(c)) \in range(c)$. Otherwise, we say that c is *violated*. Let $C^A = \{c \in C \mid A(scope(c)) \in range(c)\}$. An assignment A is a *solution* iff $C = C^A$. CP systems are oriented towards solving CSP and typically interleave backtracking search to explore assignments with *constraint propagation* to prune the set of values a variable can take. The effect of constraint propagation is studied in terms of *local consistency*. E.g., a binary constraint c is called *arc consistent* iff a variable in $scope(c)$ is assigned any value, there exists a value in the domain for the other variable in $scope(c) \setminus \{v\}$ such that c is not violated. An n-ary constraint c is called *domain consistent* iff $v \in scope(c)$ is assigned any value, there exist values in the domains of all other variables in $scope(c) \setminus \{v\}$ such that c is not violated. Observe that, in general, a constraint propagator that enforces domain consistency prunes more values than one that enforces arc consistency on a binary decomposition of the original constraint.

CSPs can be encoded with ASP (Niemelä 1999), which is founded on logic programming. A *(normal) logic program* P over an alphabet $\mathcal{A}$ is a finite set of *rules* r of the form $a_0 \leftarrow a_1, \dots, a_m, not\ a_{m+1}, \dots, not\ a_n$ where $a_i \in \mathcal{A}$ are *atoms* for

$0 \leq i \leq n$. A *default literal* is an atom a or its *default negation not* a. The atom $\mathit{head}(r) = a_0$ is called the *head* of r and the set of default literals $\mathit{body}(r) = \{a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n\}$ is called the *body* of r. A *fact* is a rule r such that $\mathit{body}(r) = \emptyset$. We also use $\leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n$ as a shorthand for $a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n, \mathit{not}\ a_0$, referred to as *integrity constraint*. For a set of default literals S, define $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid \mathit{not}\ a \in S\}$. For restricting S to atoms $\mathcal{E}$, define $S|_{\mathcal{E}} = \{a \mid a \in S^+ \cap \mathcal{E}\} \cup \{\mathit{not}\ a \mid a \in S^- \cap \mathcal{E}\}$. For $X \subseteq \mathcal{A}$ define *external support* for X as $\mathit{ES}_P(X) = \{\mathit{body}(r) \mid r \in P,\ \mathit{head}(r) \in X,\ \mathit{body}(r)^+ \cap X = \emptyset\}$. The set of atoms occurring in P is denoted by $\mathit{atom}(P)$, and the set of bodies in P is $\mathit{body}(P) = \{\mathit{body}(r) \mid r \in P\}$. For regrouping rules sharing the heads in $X \subseteq \mathcal{A}$, define $P_X = \{r \in P \mid \mathit{head}(r) \in X\}$, and for bodies sharing the same head a, define $\mathit{body}(a) = \{\mathit{body}(r) \mid r \in P,\ \mathit{head}(r) = a\}$. A *logic program with externals over* $\mathcal{E}$ is a logic program P over an alphabet distinguishing regular atoms $\mathcal{A}$ and external atoms $\mathcal{E}$, such that $\mathit{head}(r) \in \mathcal{A}$ for each $r \in P$. Let $Y \subseteq \mathcal{E}$. For a logic program P over externals from $\mathcal{E}$ define the *pre-reduct* $P(Y) = \{\mathit{head}(r) \leftarrow \mathit{body}(r)|_{\mathcal{A} \setminus \mathcal{E}} \mid r \in P,\ \mathit{body}(r)^+|_{\mathcal{E}} \subseteq Y,\ \mathit{body}(r)^-|_{\mathcal{E}} \cap Y = \emptyset\}$. A *splitting set* for a logic program P (Lifschitz and Turner 1994) is a set $\mathcal{E} \subseteq \mathcal{A}$ if $\mathit{head}(r) \in \mathcal{E}$ then $\mathit{body}(r)^+ \cup \mathit{body}(r)^- \subseteq \mathcal{E}$ for each $r \in P$. Observe that, if $\mathcal{E}$ is a splitting set of P, it *splits* P into a logic program $P_{\mathcal{E}}$ over $\mathcal{E}$ and a logic program $P_{\mathcal{A} \setminus \mathcal{E}}$ with externals over $\mathcal{E}$. The semantics of a logic program P is given by its answer sets. A set $X \subseteq \mathcal{A}$ is an *answer set* of P, if X is a minimal model of the *reduct* $P^X = \{\mathit{head}(r) \leftarrow \mathit{body}(r)^+ \mid r \in P,\ \mathit{body}(r)^- \cap X = \emptyset\}$ (Gelfond and Lifschitz 1988). If $\mathcal{E}$ is a splitting set of P, the set $Z \subseteq \mathcal{A}$ is an answer set of P iff $Z = X \cup Y$ with X is an answer set of $P_{\mathcal{E}}$ and Y is an answer set of $P_{\mathcal{A} \setminus \mathcal{E}}(Y)$ (Splitting Set Theorem, Lifschitz and Turner 1994). Although the language of ASP is propositional, atoms in $\mathcal{A}$ and can be constructed from a first-order signature via a *grounding* process, systematically substituting all occurrences of first-order variables by terms, resulting in a *(ground) instantiation*.

Following Gebser et al. (2007), the answer sets of a logic program P can be characterised as Boolean assignments over $\mathit{atom}(P) \cup \mathit{body}(P)$ that do not conflict with the conditions induced by the Clark (1978) *completion* and all *loop formulas* of P (Lee 2005). These conditions are expressed in terms of nogoods (Dechter 2003). Formally, a *(Boolean) assignment* $\mathbf{A}$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of *(signed) literals* σ_i of the form $\mathbf{T}a$ or $\mathbf{F}a$ where a is in the scope of $\mathbf{A}$, e.g., $\mathit{scope}(\mathbf{A}) = \mathit{atom}(P) \cup \mathit{body}(P)$. The complement of a literal σ is denoted $\overline{\sigma}$. True and false variables in $\mathbf{A}$ are accessed via $\mathbf{A^T}$ and $\mathbf{A^F}$. A *nogood* represents a set $\delta = \{\sigma_1, \ldots, \sigma_n\}$ of signed literals, expressing a condition *conflicting* with any assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$. If $\delta \setminus \mathbf{A} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{A}$, we say that δ is *unit* and *asserts* the *unit-resulting* literal $\overline{\sigma}$. A *total* assignment, that is $\mathbf{A^T} \cup \mathbf{A^F} = \mathit{scope}(\mathbf{A})$ and $\mathbf{A^T} \cap \mathbf{A^F} = \emptyset$, is a *solution* for a set of nogoods Γ if $\delta \not\subseteq \mathbf{A}$ for each $\delta \in \Gamma$.

3 Nogoods of Logic Programs with Externals

We generalise the approach of Gebser et al. (2007) and describe nogoods capturing inferences from Clark completion and loop formulas for logic programs P with externals over $\mathcal{E}$. For $\beta = \{a_1, \dots, a_m, \mathit{not}\ a_{m+1}, \dots, \mathit{not}\ a_n\} \in \mathit{body}(P)$, define

$$\Delta_\beta = \left\{ \begin{array}{l} \{\mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots \mathbf{F}a_n, \mathbf{F}\beta\}, \\ \{\mathbf{F}a_1, \mathbf{T}\beta\}, \dots, \{\mathbf{F}a_m, \mathbf{T}\beta\}, \{\mathbf{T}a_{m+1}, \mathbf{T}\beta\}, \dots, \{\mathbf{T}a_n, \mathbf{T}\beta\} \end{array} \right\}.$$

Intuitively, the nogoods in Δ_β enforce the truth of body β iff all its elements are satisfied. For an atom $a \in \mathit{atom}(P)$ with $\mathit{body}(a) = \{\beta_1, \dots, \beta_k\}$, define

$$\Delta_a = \left\{ \ \{\mathbf{T}\beta_1, \mathbf{F}a\}, \dots, \{\mathbf{T}\beta_k, \mathbf{F}a\}, \{\mathbf{F}\beta_1, \dots, \mathbf{F}\beta_k, \mathbf{T}a\} \ \right\}.$$

Let $\Delta_P^{\mathcal{E}} = \bigcup_{\beta \in \mathit{body}(P)} \Delta_\beta \cup \bigcup_{a \in \mathit{atom}(P) \setminus \mathcal{E}} \Delta_a$. The solutions for $\Delta_P^{\emptyset}$ correspond to the models of the completion of P (Gebser et al. 2007). To capture the effect of loop formulas induced by a set $L \subseteq \mathit{atom}(P) \setminus \mathcal{E}$, for $a \in L$ define $\lambda(a, L) = \{\{\mathbf{T}a\} \cup \{\mathbf{F}\beta \mid \beta \in \mathit{ES}_P(L)\}\}$. The set of loop nogoods is $\Lambda_P^{\mathcal{E}} = \bigcup_{L \subseteq \mathit{atom}(P) \setminus \mathcal{E}, L \neq \emptyset} \{\lambda(a, L) \mid a \in L\}$. Let P be a logic program and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution for $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ such that $\mathbf{A}^{\mathbf{T}} \cap \mathit{atom}(P) = X$ (Gebser et al. 2007). We combine this result with the Splitting Set Theorem (Lifschitz and Turner 1994).

Proposition 1
Let P be a logic program, $\mathcal{E}$ a splitting set for P, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta_{P_{\mathcal{E}}}^{\emptyset} \cup \Lambda_{P_{\mathcal{E}}}^{\emptyset} \cup \Delta_{P_{\mathcal{A} \setminus \mathcal{E}}}^{\mathcal{E}} \cup \Lambda_{P_{\mathcal{A} \setminus \mathcal{E}}}^{\mathcal{E}}$ such that $\mathbf{A}^{\mathbf{T}} \cap (\mathit{atom}(P_{\mathcal{E}}) \cup \mathit{atom}(P_{\mathcal{A} \setminus \mathcal{E}})) = X$.

An efficient algorithm for computing solutions to $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ is *Conflict-Driven Nogood Learning* (CDNL, Gebser et al. 2007). It combines search and propagation by recursively assigning the value of a proposition and performing unit-propagation to determine consequences of an assignment (Mitchell 2005).

4 Lazy Nogood Generation

Instead of generating all nogoods $\Delta_P^{\emptyset} \cup \Lambda_P^{\emptyset}$ a-priori, referred to as *eager* encoding, we introduce external propagators to generate nogoods on demand, i.e., only when they are able to propagate new information. We call this technique *Lazy Nogood Generation*, generalising an approach to encoding constraints over finite domains into sets of clauses by executing constraint propagation during SAT search and recording the propagation in terms of clauses (Lazy Clause Generation, Ohrimenko et al. 2009). Formally, an *external propagator* for a set of nogoods Γ is a function π that maps a Boolean assignment $\mathbf{A}$ to a subset of Γ such that for each total assignment $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \subseteq \mathbf{A}$ for some $\delta' \in \pi(\mathbf{A})$. In other words, an external propagator generates a conflicting nogood from Γ iff some nogood in Γ is conflicting with a total assignment. We call an external propagator *conflict-optimal*, if this condition holds for each (partial) assignment. Notice that, even for a conflict-optimal external propagator, unit-propagation on Γ might be stronger than unit-propagation on lazily generated nogoods, i.e., infer more unit-resulting literals. To close this gap, we define inference-optimal external propagators. An external propagator π for a set of nogoods Γ is *inference-optimal* if π is conflict-optimal and for each non-conflicting assignment $\mathbf{A}$ if $\delta \setminus \mathbf{A} = \{\sigma\}$ such that $\overline{\sigma} \notin \mathbf{A}$ for some $\delta \in \Gamma$ then $\delta' \setminus \mathbf{A} = \{\sigma\}$ for some $\delta' \in \pi(\mathbf{A})$. The correspondence between external propagation and the set of nogoods it represents can be formalised as follows.

Proposition 2
Let Δ be a set of nogoods, and π be an external propagator for $\Gamma \subseteq \Delta$. Then, the assignment $\mathbf{A}$ is a solution of Δ iff $\mathbf{A}$ is a solution of $(\Delta \setminus \Gamma) \cup \pi(\mathbf{A})$.

One of the advantages of Lazy Nogood Generation over eager encodings is space efficiency. For instance, the worst-case exponential number of loops in a logic program P makes an eager encoding of the conditions induced by $\Lambda^{\emptyset}_{P}$ infeasible (Lifschitz and Razborov 2006). External propagation, however, can check all loop formulas in linear time (Calimeri et al. 2002). The approach of Gebser et al. (2007) applies non-optimal external propagation that determines the nogoods in $\Lambda^{\emptyset}_{P}$ on demand via directed unfounded set inference.

To reflect Lazy Nogood Generation also on the language level of ASP, we make use of splitting (Lifschitz and Turner 1994) for outsourcing conditions over $\mathcal{E} \subseteq \mathcal{A}$ into $P_{\mathcal{E}}$. Instead of making $P_{\mathcal{E}}$ explicit, however, a set of external propagators Π can be provided that precisely represent the conditions induced by $P_{\mathcal{E}}$. We will write $atom(\Pi)$ to access $\mathcal{E}$. The previous propositions yield the following result.

Theorem 1
Let P be a logic program, $\mathcal{E}$ a splitting set for P, Π a set of external propagators for $\Delta^{\emptyset}_{P_{\mathcal{E}}} \cup \Lambda^{\emptyset}_{P_{\mathcal{E}}}$, and $X \subseteq \mathcal{A}$. Then, X is an answer set of P iff there is a (unique) solution $\mathbf{A}$ for $\Delta^{\mathcal{E}}_{P_{\mathcal{A}\setminus\mathcal{E}}} \cup \Lambda^{\mathcal{E}}_{P_{\mathcal{A}\setminus\mathcal{E}}} \cup \bigcup_{\pi\in\Pi} \pi(\mathbf{A})$ s.t. $\mathbf{A}^{\mathbf{T}} \cap (atom(P_{\mathcal{E}}) \cup atom(\Pi)) = X$.

External propagation provides a form of modularity that allows programmers to select encodings which propagate better, but were previously avoided for space-related reasons. For instance, in (Drescher and Walsh 2010) we describe eager encodings that simulate constraint propagators for the ALL-DIFFERENT constraint which achieve arc, bound, or range consistency. A constraint propagator that can achieve domain consistency exists (Régin 1994) but it cannot be simulated efficiently (Bessière et al. 2009). Because of the fact that external propagators generate nogoods only on demand, however, we can implicitly represent encodings via Lazy Nogood Generation that are otherwise infeasible.

5 Conflict-Driven Nogood Learning with Lazy Nogood Generation

We develop a decision procedure for answer set solving with Lazy Nogood Generation based on CDNL (Gebser et al. 2007). It is centred around *conflict analysis* according to the *First-UIP* scheme (Mitchell 2005). That is, a conflicting nogood is iteratively resolved against other nogoods until a conflicting nogood that contains a *unique implication point* is obtained. This guides backjumping. Recording the resolved nogood enables conflict-driven learning, which can further prune the search space. For controlling the set of recorded nogoods, deletion strategies can be applied (cf. Moskewicz et al. 2001). In contrast to CDNL we will integrate external propagators that perform Lazy Nogood Generation in order to represent conditions on the answer sets of a logic program that are not encoded eagerly. Like their eager encoded counterpart, lazily generated nogoods can contribute to conflict analysis. This can improve propagation and contribute to lookback-based search heuristics.

```
Input  : A logic program $P$ with external propagators $\Pi$.
Output: An answer set of $P$ if one exists.
1  $\mathbf{A} \leftarrow \emptyset$                                   // Boolean assignment
2  $\nabla \leftarrow \emptyset$                                   // set of recorded nogoods
3  $dl \leftarrow 0$                                   // decision level
4  loop
5      $(\mathbf{A}, \nabla) \leftarrow$ PROPAGATION$(P, \Pi, \nabla, \mathbf{A})$
6      if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_P^{atom(\Pi)} \cup \nabla$ then
7          if $dl = 0$ then return no answer set
8          $(\varepsilon, k) \leftarrow$ CONFLICTANALYSIS$(\delta, P, \nabla, \mathbf{A})$
9          $\nabla \leftarrow \nabla \cup \{\varepsilon\}$
10         $\mathbf{A} \leftarrow \mathbf{A} \backslash \{\sigma \in \mathbf{A} \mid k < dl(\sigma)\}$
11         $dl \leftarrow k$
12     else if $\mathbf{A^T} \cup \mathbf{A^F} = atom(P) \cup body(P) \cup atom(\Pi)$ then
13         return $\mathbf{A^T} \cap (atom(P) \cup atom(\Pi))$
14     else
15         $\sigma_d \leftarrow$ SELECT$(P, \Pi, \nabla, \mathbf{A})$
16         $\mathbf{A} \leftarrow \mathbf{A} \circ (\sigma_d)$
17         $dl \leftarrow dl + 1$
```

Fig. 1. CDNL-LNG

Different from eager encoded nogoods, the amount of recorded nogoods can be controlled via deletion. If needed, however, deleted nogoods will be rediscovered.

5.1 Main Algorithm

The main algorithm, CDNL-LNG, is shown in Fig. 1. It takes a logic program P with external propagators Π, and starts with an empty assignment $\mathbf{A}$ and an empty set ∇ that will store recorded nogoods, including lazily generated nogoods. The *decision level* dl is initialised with 0. Its purpose is counting *decision literals* in the assignment. We use $dl(\sigma)$ to access the decision level of literal σ. The following loop is very similar to CDNL. First, PROPAGATION (Line 5) extends $\mathbf{A}$ and ∇, as described in the next section. If this encounters a conflict (Line 6), the CONFLICT-ANALYSIS procedure generates a conflicting nogood ε by exploiting interdependencies between nogoods in $\Delta_P^{atom(\Pi)} \cup \nabla$ through conflict resolution, and determines a decision level k to continue search at. Then, ε is added to the set of recorded nogoods ∇ in Line 9. This can prune the search space and lead to faster propagation. Lines 10–11 account for backjumping to level k. Thereafter ε is unit and triggers inference in the next round of propagation. If CONFLICTANALYSIS, however, yields a conflict at level 0, no answer set exists (Line 7). Furthermore, we distinguish the cases of a complete assignment (Lines 12–13) and a partial one (Lines 14–17). In case of a complete assignment, the atoms in $\mathbf{A^T}$ obtain an answer set of P. In the other case, $\mathbf{A}$ is partial and no nogood is conflicting. Then, a decision literal σ_d is selected by some heuristic, added to $\mathbf{A}$, and the decision level is incremented. While the CONFLICTANALYSIS and SELECT procedures are similar to the ones in CDNL, we extend PROPAGATION to accommodate Lazy Nogood Generation.

```
Input  : A logic program $P$ with external propagators $\Pi$, recorded nogoods $\nabla$,
         Boolean assignment $\mathbf{A}$.
Output: An extended assignment and set of recorded nogoods.
loop
    repeat                                                          // unit-propagation
        if $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_P^{atom(\Pi)} \cup \nabla$ then return $(\mathbf{A}, \nabla)$
        $\Sigma \leftarrow \{\delta \in \Delta_P^{atom(\Pi)} \cup \nabla \mid \delta \backslash \mathbf{A} = \{\sigma\}, \overline{\sigma} \notin \mathbf{A}\}$
        if $\Sigma \neq \emptyset$ then let $\sigma \in \delta \backslash \mathbf{A}$ for some $\delta \in \Sigma$ in
            $\mathbf{A} \leftarrow \mathbf{A} \circ (\overline{\sigma})$
    until $\Sigma = \emptyset$
    foreach $\pi \in \Pi$ do
        $\Sigma \leftarrow \pi(\mathbf{A})$                                    // external propagation
        if $\Sigma \neq \emptyset$ then break
    if $\Sigma = \emptyset$ then
        $\Sigma \leftarrow$ LoopFormulaPropagation$(P, \mathbf{A})$        // loop formula propagation
    if $\Sigma = \emptyset$ then return $(\mathbf{A}, \nabla)$
    $\nabla \leftarrow \nabla \cup \Sigma$
```

Fig. 2. PROPAGATION

5.2 Propagation

A specification of our PROPAGATION procedure is shown in Fig. 2. It works on a logic program P with external propagators Π, a set of recorded nogoods ∇, and an assignment $\mathbf{A}$. PROPAGATION interleaves unit-propagation on nogoods $\Delta_P^{atom(\Pi)}$ and recorded nogoods ∇ including lazily generated nogoods from external propagators. We start with unit-propagation (Lines 2–7), resulting either in a conflict, i.e., some nogood is conflicting (Line 3), or in a fixpoint possibly extending $\mathbf{A}$ with unit-resulting literals. If there is no conflict, PROPAGATION performs external propagation following some priority (Lines 8–10). Based on $\mathbf{A}$, each propagator may encode inference in a set of lazily generated nogoods Σ which is added to the set of recorded nogoods ∇ at the end of the loop in Line 14. The LOOPFORMULAPROPAGATION procedure (Line 12, c.f. Gebser et al. 2007) works similarly to ensure that no loop formula is violated, i.e., no loop nogood in $\Lambda_P^{atom(\Pi)}$ is conflicting. This only has an effect if the logic program is non-tight (Erdem and Lifschitz 2003).

Note that external propagation is interleaved by unit-propagation in order to assign unit-resulting literals immediately and detect conflicts early. Our algorithm also favours external propagation over loop formula propagation, motivated by the fact that external propagators can affect the assignment to atoms in $atom(\Pi)$, possibly falsifying external support for a loop in P.

6 Constraint Answer Set Solving via Lazy Nogood Generation

One difficult task for answer set solving with Lazy Nogood Generation remains, i.e., the one of creating efficient external propagators. A research area that is largely concerned with efficient propagation in declarative problem solving is CP. We here follow the idea of Ohrimenko et al. (2009) and apply CP techniques to generate

$$
\begin{array}{lrl}
(1) & \multicolumn{2}{l}{\#\text{var } \$\mathit{value}(x) = 1..|S|} \\
(2) & \multicolumn{2}{l}{\$\mathit{value}(x)\ \#{==}\ i \leftarrow \mathit{hint}(x,i)} \\
(3) & \multicolumn{2}{l}{\#\textsc{all-different}\ \{\$\mathit{value}(x) \mid \forall x \in S\} \leftarrow} \\
(4) & \mathit{cns}(x,y) & \leftarrow \mathit{conn}(x,y),\ \#\textsc{linear}\ [\$\mathit{value}(y), -\$\mathit{value}(x)] == 1 \\
(5) & \mathit{in\_path}(x,y) & \leftarrow \mathit{conn}(x,y),\ \mathit{not\ in\_path}(x,y),\ \mathit{not\ hint}(y,1) \\
(6) & \mathit{out\_path}(x,y) & \leftarrow \mathit{conn}(x,y),\ \mathit{not\ out\_path}(x,y) \\
(7) & & \leftarrow \mathit{in\_path}(x,y),\ \mathit{in\_path}(x,z) \\
(8) & \mathit{reached}(1) & \leftarrow \\
(9) & \mathit{reached}(y) & \leftarrow \mathit{reached}(x),\ \mathit{in\_path}(x,y) \\
(10) & & \leftarrow \mathit{not\ reached}(x) \\
(11) & & \leftarrow \mathit{in\_path}(x,y),\ \mathit{not\ cns}(x,y)
\end{array}
$$

where $x, y, z \in S, x \neq y, y \neq z$, and $i \in \{1, \ldots, |S|\}$

Fig. 3. CASP encoding of Hidato.

lazy nogoods representing constraints over finite domains. To reflect this on the language level, we use of CASP, a paradigm that naturally merges CP and ASP.

6.1 A Language for Lazy Nogood Generation

CASP abstracts from non-propositional constraints by incorporating *constraint atoms* into logic programs. We access the constraint atom associated to a constraint c via the function $\mathit{atom}(c)$. A *constraint logic program* is a tuple $\mathbb{P} = (V, D, C, P)$, where V, D, C are the same as in the definition of a CSP, and P is a logic program with externals over *constraint atoms* $\mathcal{C} = \{\mathit{atom}(c) \mid c \in C\}$. To improve the modelling convenience, however, we follow Gebser et al. (2009) and view a rule r with $\mathit{head}(r) \in \mathcal{C}$ as an integrity constraint $\leftarrow \mathit{body}(r), \mathit{not\ head}(r)$.

Example 1
Hidato is a number-placement puzzle game. Its goal is to fill a grid with consecutive integers connecting horizontally, vertically, or diagonally. Some numbers, including the smallest and the highest number on the grid, are preassigned in order to guarantee a unique solution. Logic programming based approaches are particularly well suited for modelling Hidato because REACHABILITY is encoded straightforwardly using recursive transitive closure. This is difficult to represent, e.g., in CP. Let S be the set of cells on the grid. Fig. 3 provides a CASP encoding of Hidato. It works on a set of facts about connections between cells $x, y \in S$ of the form $\mathit{conn}(x, y)$, and preassigned values of the form $\mathit{hint}(x, i)$, representing that cell $x \in S$ has the value $i \in \{1, \ldots, |S|\}$ preassigned. We distinguish variable definitions and constraint atoms, indicated by the # symbol, and constraint variables, indicated by the $ symbol, from regular atoms. Line 1 defines a constraint variable for each cell in the grid with values between 1 and $|S|$. Line 2 makes use of primitive constraints. It insures that preassignments are respected by the corresponding variables. Line 3 encodes the condition that the values taken by the cells in the grid are ALL-DIFFERENT. Line 4 defines an auxiliary atom $\mathit{cns}(x, y)$ which indicates whether two connected cells x and y take consecutive values. Lines 6–10 encode a Hamiltonian path, represented by atoms of the form $\mathit{in\_path}(x, y)$, and the integrity constraint in Line 11

insures that the values taken by two connected cells in the Hamiltonian path are consecutive. Note that the global ALL-DIFFERENT constraint is redundant in this encoding, but it improves run time.

A fundamental difference to traditional CP is that, in CASP, each constraint c is reified via $atom(c)$. Its truth value is determined by the conditions induced by P and an assignment A to the variables in $scope(c)$. The set of constraint atoms $\mathcal{C}^A = \{atom(c) \mid c \in C^A\}$ correspond to the constraints satisfied by A. Let $\mathbb{P}$ be a constraint logic program and A an assignment. The pair (X, A) is a *constraint answer set* of $\mathbb{P}$ iff X is an answer set of $P(\mathcal{C}^A)$ (cf. Gebser et al. 2009). Given that assignments A and their effect on each constraint can be represented in a logic program (Niemelä 1999), the task of computing constraint answer sets can be reduced to the one of computing answer sets of P with external propagators for generating assignments A and capturing the inference of constraint propagation.

6.2 Conflict-Driven Constraint Answer Set Solving

To begin with, CASP solving via Lazy Nogood Generation requires a propositional representation of assignments to constraint variables. A popular choice is called the *value encoding*. In the value encoding, an atom $value(v, i)$, representing $v = i$, is introduced for each variable $v \in V$ and value $i \in dom(v)$. Intuitively, the atom $value(v, i)$ is true if v takes the value i, and false if v takes a value different from i (cf. Walsh 2000). To insure that an assignment $\mathbf{A}$ represents a consistent set of possible values for v, for instance, the atoms $value(v, 1)$ and $value(v, 3)$ cannot both be true, we encode the conditions that v must not take two values, i.e., $\{\mathbf{T}value(v, i), \mathbf{T}value(v, j)\} \not\subseteq \mathbf{A}$ for all $i, j \in dom(v),\ i \neq j$, and that v must take at least one value, i.e., $\mathbf{F}value(v, i) \notin \mathbf{A}$ for some $i \in dom(v)$, in the set of nogoods $\Gamma_v = \{\{\mathbf{T}value(v, i), \mathbf{T}value(v, j)\} \mid i, j \in dom(v),\ i \neq j\} \cup \{\{\mathbf{F}value(v, i) \mid i \in dom(v)\}\}$ (Drescher and Walsh 2010). We employ external propagators to represent the nogoods in Γ_v. The algorithm in Fig. 4 provides a specification of an inference-optimal external propagator for this task. It takes a Boolean assignment $\mathbf{A}$ and returns a set of lazily generated nogoods, initialised in Line 1, that are unit or conflicting. Lines 2–3 insure that if v is assigned a value i then all other values are removed from its domain, while Lines 4–5 deal with the condition that there is at least one value that can be assigned to v. This procedure can be made very efficient, for instance, by using *watched literals* (Moskewicz et al. 2001). Another popular choice for representing constraint variables is the *bound encoding*, where an atom is introduced for each variable $v \in V$ and value $i \in dom(v)$ to represent that v is bounded by i, i.e., $v \leq i$ (cf. Tamura et al. 2006). Similar to the value encoding, we can define nogoods that insure a consistent Boolean assignment (Drescher and Walsh 2010). A combination of value and bound encoding is also possible.

We see atoms from the value and bound encoding as *primitive constraints*, as all constraints can be decomposed into nogoods over them, e.g., by describing changes in the variables' domains inferred by constraint propagation. This way, constraint propagators can be encoded eagerly or lazily. Transforming a constraint propagator into an external propagator is straightforward: Rather than applying domain

```
Input  : A Boolean assignment $\mathbf{A}$.
Output: A set of lazily generated nogoods.
1 $\nabla \leftarrow \emptyset$                                   // set of lazily generated nogoods
2 if $\mathbf{T}value(v,i) \in \mathbf{A}$ for some $i \in dom(v)$ then
3     $\nabla \leftarrow \{\{\mathbf{T}value(v,i),\ \mathbf{T}value(v,j)\} \mid j \in dom(v)\backslash\{i\},\ \mathbf{F}value(v,j) \notin \mathbf{A}\}$
4 if $\mathbf{T}value(v,i) \notin \mathbf{A}$ for some $i \in dom(v) \wedge \forall j \in dom(v)\backslash\{i\}\ \mathbf{F}value(v,j) \in \mathbf{A}$ then
5     $\nabla \leftarrow \{\{\mathbf{F}value(v,i) \mid i \in dom(v)\}\}$
6 return $\nabla$
```

Fig. 4. An external propagator for the value encoding Γ_v.

changes directly, the constraint propagator has to be made encoding its inferences in form of nogoods over primitive constraints (Ohrimenko et al. 2009).

Example 2
An external propagator for encoding the reified ALL-DIFFERENT constraint c is specified in the algorithm given by Fig. 5. Provided with a Boolean assignment $\mathbf{A}$, it starts with an empty set of lazily generated nogoods, followed by a distinction into two cases. First, if the constraint is to be satisfied, i.e., $\mathbf{T}atom(c) \in \mathbf{A}$, then for each variable in the scope of the constraint that has a value assigned, a nogood is generated that asserts the removal of this value from the domain of all other variables in the scope of the constraint (Lines 3–4). On the other hand, if the constraint is not set to be satisfied, the algorithm checks whether two variables in the scope of the constraint have the same value assigned (Lines 6–10). If so, the ALL-DIFFERENT constraint is violated and a nogood asserting that the constraint atom is set to false will be returned (unless $\mathbf{F}atom(c) \in \mathbf{A}$, in which case the constraint atom is already false). If, however, no such two variables can be found and all variables in the scope of the constraint have a value assigned, then the ALL-DIFFERENT condition is satisfied and a nogood is generated that asserts the truth of the constraint atom (Lines 11–12).

Observe that this propagator enforces arc consistency on the binary decomposition of the reified ALL-DIFFERENT constraint if $atom(c)$ is true, but propagates weakly if $atom(c)$ is false. However, propagators that achieve higher levels of local consistency are also possible (Régin 1994).

Constraint propagators encode new information into unit or conflicting nogoods, while unit-propagation performs the encoded inference within the next iteration. This can change (the representation of) some variable's domain. Unit-propagation, constraint propagation, and loop formula propagation are repeated until a fixpoint is reached or a conflict is encountered. By generating a conflicting nogood, for instance, a constraint propagator can yield that the underlying constraint is violated.

Notice that constraint answer set solving via Lazy Nogood Generation is fundamentally more general then the SAT-based approach to CP solving presented in (Ohrimenko et al. 2009), which can be seen as a special case of our techniques, that is, if the truth value of each constraint atom is known a-priori and every nogood is represented by a clause.

```
Input  : A Boolean assignment $\mathbf{A}$.
Output: A set of lazily generated nogoods.
1  $\nabla \leftarrow \emptyset$                                  // set of lazily generated nogoods
2  if $\mathbf{T}atom(c) \in \mathbf{A}$ then
3      foreach $v \in scope(c)$ s.t. $\mathbf{T}value(v,i) \in \mathbf{A}$ for some $i \in dom(v)$ do
4          $\nabla \leftarrow \nabla \cup \{\{\mathbf{T}atom(c),\ \mathbf{T}value(v,i),\ \mathbf{T}value(w,i)\} \mid w \in scope(c)\backslash\{v\},$
                                   $i \in dom(w),\ \mathbf{F}value(w,i) \notin \mathbf{A}\}$
5  else
6      foreach $v \in scope(c)$ s.t. $\mathbf{T}value(v,i) \in \mathbf{A}$ for some $i \in dom(v)$ do
7          if $w \in scope(c)\backslash\{v\}$ s.t. $\mathbf{T}value(w,i) \in \mathbf{A}$ then
8              if $\mathbf{F}atom(c) \notin \mathbf{A}$ then
9                  $\nabla \leftarrow \{\{\mathbf{T}atom(c),\ \mathbf{T}value(v,i),\ \mathbf{T}value(w,i)\}\}$
10             return $\nabla$
11     if $\forall v \in scope(c)\ \exists i \in dom(v)$ s.t. $\mathbf{T}value(v,i) \in \mathbf{A}$ then
12         $\nabla \leftarrow \{\{\mathbf{F}atom(c)\} \cup \{\mathbf{T}value(v,i) \mid v \in scope(c),$
                                   $i \in dom(v),\ \mathbf{T}value(v,i) \in \mathbf{A}\}\}$
13 return $\nabla$
```

Fig. 5. An ext. propagator for encoding the reified ALL-DIFFERENT constraint c.

7 Experiments

We have implemented Lazy Nogood Generation for constraint variables, the ALL-DIFFERENT and integer LINEAR constraints within a new version of our prototypical CASP system *inca*[1]. It is based on the latest development version of the ASP system *clingo*[2] (3.0.92), and uses CDNL-LNG as its core reasoning engine. We also include the CASP system *clingcon*[2] (2.0.0-beta) in our analysis. It too extends *clingo* (3.0.92), but integrates the CP solver *gecode*[3] (3.7.1). Similar to our approach, *clingcon* is based on CDNL and abstracts from the constraints via constraint atoms. Following the idea of SMT, however, *clingcon* employs its CP solver to check the existence of a constraint variable assignment that does not violate any constraint according to the assignment to constraint atoms. In turn, the CP solver can yield a conflict or propagate constraint atoms by generating nogoods over only constraint atoms that occur in the constraint logic program. Hence, this process constitutes a very limited form of Lazy Nogood Generation. We have set *clingcon* to generate nogoods by looking at dependency between constraints according to the *irreducibly inconsistent set construction* method in "forward" mode, when we noticed that this option significantly improves the performance of *clingcon* on our benchmarks. For a comparison with the state-of-the-art in answer set solving, our experiments also consider eager encodings. We often use *inca* to generate eager encodings, but it only relies on its ASP subsystem *clingo* (3.0.92) for the solving process. Hence, we denote this option as *clingo*. Note that loop formulas are en-

[1] to be released
[2] `http://potassco.sourceforge.net`
[3] `http://www.gecode.org`

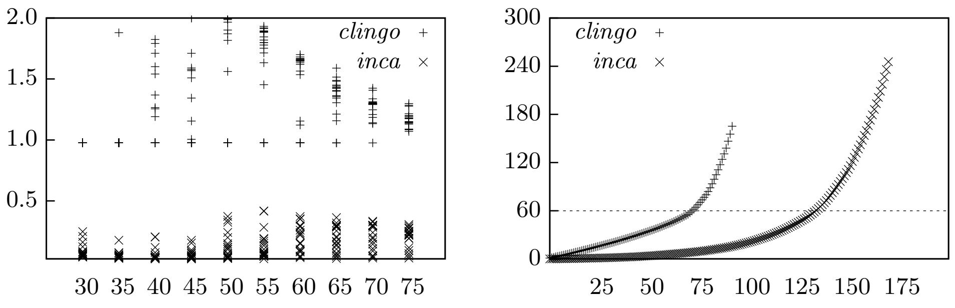

Fig. 6. Results on Latin square problems of size 40. Left: Maximum memory usage in GB by preassigned values in percent. Right: Runtime in minutes by number of solved instances.

coded lazily in all settings. Experiments were run on a Linux PC, where each run was limited to 600 sec CPU time on a 2.00 GHz core and 2 GB RAM.

7.1 Latin Squares

We consider Latin squares to test Lazy Nogood Generation for the ALL-DIFFERENT constraint. A *Latin square* is an $n \times n$-table filled with n different elements such that each element occurs exactly once in each row and each column of the table. The *Latin square problem* is to determine whether a partially filled table can be completed in such a way that a Latin square is obtained. We consider randomly generated tables because they combine the features of purely random problems and highly structured ones (Gomes and Selman 1997). Previous experiments on this benchmark domain (Drescher and Walsh 2010) with $n = 20$ have shown eager encodings outperforming systems like *clingcon* (0.1.2), a previous version of *clingcon* with only very limited capacities for conflict-driven learning and no support for global constraints. In order to study the computational behaviour on large-scale tables we have increased size to $n = 40$. We have generated 200 instances near to the phase transition. Phase transition can be observed, e.g., at the maximum memory consumption of *clingo* in relation to the preassignment rate (Fig. 6). In fact, we observe most cases of memory exhaustion with a preassignment rate of 50 percent. Since all atoms in the CASP model for Latin squares are constraint atoms whose truth value is known a-priori, *clingcon* cannot make use of its conflict-driven capacities. Instead, search and propagation in *clingcon* is carried out by its backtracking-based CP solver, i.e., following a smallest domain and smallest value first selection heuristic, whilst search in *clingo* and *inca* is based on activity in conflict analysis according. This is likely be the reason that renders *clingcon* ineffective on this benchmark domain, solving only 9 of 200 instances. To our surprise, enabling learning capacities in *clingcon* by decomposing the ALL-DIFFERENT constraints into primitive constraints, like in the eager encoding used by *clingo*, exhausts CPU time on all instances. Within the allowed execution time, *clingo*, however, solves almost half of all instances. Though, *clingo* performs drastically worse on Latin square problems with the increased size, compared to previous experiments (Drescher and Walsh 2010). Lazy Nogood Generation in *inca*, on the other hand, avoids huge

Table 1. *Average time in seconds over completed runs on Latin square, Packing, and Numbrix benchmarks. Number of completed runs are given in parenthesis.*

benchmark class (#instances)	*clingo*	*clingcon*	*inca*
Latin squares (200)	106.61 (93)	34.38 (9)	86.19 (171)
Packing (50)	104.06 (1)	33.10 (50)	24.61 (50)
Numbrix (12)	10.42 (12)	17.36 (12)	1.29 (12)
weighted, penalised avg. time	323.69	208.34	62.20

encodings while preserving conflict-driven learning capacities. In fact, *inca* solves twice as many instances as *clingo* within one hour of execution time, and nearly twice as many instances overall (Fig. 6).

7.2 Packing

To test Lazy Nogood Generation for integer LINEAR constraints, we consider instances of the *Packing* problem, that is the problem of packing objects together inside a container. A special case of the packing problem formed a benchmark class in the third ASP competition (Calimeri et al. 2011). There, a set of squares of known dimensions had to be packed into a rectangular area such that no two squares overlap each other and all the squares are packed, possibly leaving space in the rectangular area unoccupied. Problems like Packing are particularly hard to solve with ASP systems because they typically involve large domains that affect the size of their encoding. In fact, the encoding that was given in the system track of the competition quickly reaches the memory limit of 2 GB in 49 over 50 instances, while the CASP systems *clingcon* and *inca* solve every instance within a reasonable amount of space and time. On the Packing domain, the advantage of *inca* over *clingcon* is only marginal.

7.3 Numbrix

Numbrix is a variant of Hidato. It is played on an $n \times n$ grid with the restriction that only horizontal and vertical connections are allowed. The other difference is that the positions of the smallest and the highest number are not always given. The objective of Numbrix is the same as Hidato. Our experiments determine the existence of a unique solution for $n = 9$. While *clingo*, *clingcon*, and *inca*, all are able to solve all puzzles considered in our analysis, we observe that execution time improves when CP constructs are encoded into nogoods, as in the options *clingo* and *inca*, and drastically improves when this encoding is lazy, as in the option *inca*.

7.4 Discussion

A summary of our experiments is provided in Table 1. Although more benchmark classes are needed for a meaningful comparison, we can draw a few interesting conclusions. First, execution time can improve when CP constructs are treated by external propagation rather than encoding them eagerly. The latter can lead to huge encodings, in particular, when large domains are involved. Second, the advantage of generating nogoods to describe the inferences of constraint propagators is that CDNL can exploit constraint interdependencies for directing search, and most importantly conflict analysis. The fact that *clingcon* does not encode CP constructs into nogoods, by design, is likely to be the reason for its limited success in our experiments. Third, experiments show that our approach, represented through *inca*, combines the best of both worlds: It can avoid huge encodings via abstraction to external propagation while retaining the ability to make the encoding explicit. It outperforms the state-of-the-art in constraint answer set solving on individual benchmark classes, and is more robust over all benchmark instances.

8 Related Work

Related work on the integration of ASP with other declarative problem solving paradigms is plentiful, and roughly falls into one of three categories: translation-based approaches, modular approaches, and integrated approaches.

In translation-based approaches, all parts of an (extended) ASP model are eagerly encoded into a single language for which highly efficient off-the-shelf solvers are available. Niemelä (1999) provides a simple mapping of constraints into ASP given by allowed or forbidden combinations of values in a very straightforward and easily maintainable way. We have demonstrated efficiency in (Drescher and Walsh 2010), describing what type of local consistency the unit-propagation of an ASP solver achieves on value, bounds, and range encodings. Specialised encodings for GRAMMAR and related constraints are presented in (Drescher and Walsh 2011b).

There is also a substantial body of work on encoding constraints into SAT which can be translated into ASP (Niemelä 1999). For instance, Walsh (2000) analyses two different mappings of binary CSPs into SAT, i.e., the *direct encoding* and the *log encoding*. While the direct encoding is reflected by our value encoding, the log encoding represents the bit-vector of each variable. Although the log encoding is more space-efficient, unit-propagation on the direct encoding prunes more possible values than unit-propagation (Walsh 2000). Gent (2002) proposes to encode *support* rather than encoding conflicts, and proves that unit-propagation on the support encoding achieves arc consistency. Bessière et al. (2003) generalises this technique to n-ary CSP. Specialised SAT encodings that can be propagated more efficiently have been proposed, e.g., for pseudo-Boolean constraints (Eén and Sörensson 2006), and integer LINEAR constraints (Tamura et al. 2006).

In a modular approach, theory-specific solvers interact in order to compute solutions, similar to the idea of SMT (Nieuwenhuis et al. 2006). Baselice et al. (2005) and Mellarkod and Gelfond (2008) combine systems for solving ASP and CP such that no full grounding of first-order ASP is required. Instead, they are handled in a CP solver. Dal Palù et al. (2009) employs the CP solver to also compute the answer sets. The approach taken by Balduccini (2009) consists of writing logic programs whose answer sets encode a desired CSP, which is, in turn, solved by a CP system. Järvisalo et al. (2009) obtain the overall semantics from the ones of individual modules, including CP modules. While the mentioned modular approaches see ASP and CP solvers as blackboxes, Mellarkod et al. (2008) integrate a CP solver into the decision engine of a backtracking-based ASP solver. To match the performance of SMT solvers, i.e., by employing conflict-driven techniques, Gebser et al. (2009) extend the conflict analysis of an ASP solver to include constraint atoms. The abstraction from the inference performed by constraint propagation, however, limits the exploitation of constraint interdependencies.

Answer set solving via Lazy Nogood Generation was first outlined in (Drescher and Walsh 2011a), and falls into the category of integrated approaches. The related work closest to this paper is Lazy Clause Generation (Ohrimenko et al. 2009), a SAT-based approach to CSP solving where lazy clause generators encode the inference of *propagation rules* into clauses. However, our approach generalises Lazy Clause Generation: Every nogood can be syntactically represented by a clause, but other ASP constructs are also possible, such as cardinality rules, their generalisation to weight constraint rules (Niemelä et al. 1999), and aggregations and other forms of set constructions that have been shown to be useful (Dell'Armi et al. 2003). E.g., Gebser et al. (2009) show that constraint variables can be conveniently expressed by means of cardinality rules. Elkabani et al. (2004) provide a generic framework which provides an elegant treatment of such extensions to ASP, employing constraint propagators for their handling, though, without support for conflict-driven techniques. A thorough approach to integrating propagators for weight constraint rules within a conflict-driven framework is presented in (Gebser et al. 2009).

9 Conclusion

We presented a comprehensive extension for answer set solving to address the scalability and efficiency of ASP, called Lazy Nogood Generation. Founded on a nogood-based characterisation of external propagation, our techniques allow for representing encodings that are otherwise infeasible. However, external propagators can make parts of the encoding explicit whenever it triggers inference. Deletion strategies can be applied for controlling its size. We presented key algorithms that are centred around conflict-driven learning, and seamlessly applied our techniques to CASP solving by employing constraint propagation. Experiments show that our prototypical implementation is competitive with the state-of-the-art in CASP solving, and also advocate a dedicated treatment of feasible encodings via external propagation. We expect further significant computational impact given the empirical evidence provided by Lazy Clause Generation (Ohrimenko et al. 2009). However, Lazy Nogood Generation generalises Lazy Clause Generation, as every nogood can be syntactically represented by a clause, but other ASP constructs are also possible. Future work considers the exploitation of ASP constructs like aggregation and loops. This can lead to further impact. Many questions on modelling and solving CASP also remain open, concerning encoding optimisations and further language extensions. For instance, we plan to explore the different choices that arise from the combination of translation-based and integrated constraint answer set solving.

References

Balduccini, M. 2009. Representing constraint satisfaction problems in answer set programming. In *Proceedings of ICLP'09, ASPOCP'09 Workshop*.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Baselice, S., Bonatti, P., and Gelfond, M. 2005. Towards an integration of answer set and constraint solving. In *Proceedings of ICLP'05*. Springer, 52–66.

Bessière, C., Hebrard, E., and Walsh, T. 2003. Local consistencies in SAT. In *Proceedings of SAT'03*. Springer, 299–314.

Bessière, C., Katsirelos, G., Narodytska, N., and Walsh, T. 2009. Circuit complexity and decompositions of global constraints. In *Proceedings of IJCAI'09*. 412–418.

Calimeri, F., Faber, W., Leone, N., and Pfeifer, G. 2002. Pruning operators for answer set programming systems. In *Proceedings of NMR'02*. 200–209.

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M. C., Sirianni, M., Terracina, G., and Veltri, P. 2011. The third answer set programming competition: Preliminary report of the system competition track. In *Proceedings of LPNMR'11*. Springer, 388–403.

Clark, K. 1978. Negation as failure. In *Logic and Data Bases*. Plenum Press, 293–322.

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. 2009. Answer set programming with constraints using lazy grounding. In *Proceedings of ICLP'09*. Springer, 115–129.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., and Pfeifer, G. 2003. Aggregate functions in disjunctive logic programming: Semantics, complexity, and implementation in DLV. In *Proceedings of IJCAI'03*. Morgan Kaufmann, 847–852.

Drescher, C. and Walsh, T. 2010. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming 10,* 4-6, 465–480.

Drescher, C. and Walsh, T. 2011a. Conflict-driven constraint answer set solving with lazy nogood generation. In *Proceedings of AAAI'11*. AAAI Press, 1772–1773.

Drescher, C. and Walsh, T. 2011b. Modelling grammar constraints with answer set programming. In *ICLP'11 Technical Communications*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 28–39.

Eén, N. and Sörensson, N. 2006. Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation 2*, 1–26.

Elkabani, I., Pontelli, E., and Son, T. 2004. Smodels with CLP and its applications: A simple and effective approach to aggregates in ASP. In *Proceedings of ICLP'04*. Springer, 73–89.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming 3,* 4-5, 499–518.

Gebser, M., Hinrichs, H., Schaub, T., and Thiele, S. 2009. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In *Proceedings of WLP'09*.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. 2009. On the implementation of weight constraint rules in conflict-driven ASP solvers. In *Proceedings of ICLP'09*. Springer, 250–264.

Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T. 2007. Conflict-driven answer set solving. In *Proceedings of IJCAI'07*. AAAI Press/MIT Press, 386–392.

Gebser, M., Ostrowski, M., and Schaub, T. 2009. Constraint answer set solving. In *Proceedings of ICLP'09*. Springer, 235–249.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proceedings of ICLP'88*. The MIT Press, 1070–1080.

Gent, I. P. 2002. Arc consistency in SAT. In *Proceedings of ECAI'02*. IOS Press, 121–125.

Giunchiglia, E., Lierler, Y., and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

Gomes, C. P. and Selman, B. 1997. Problem structure in the presence of perturbations. In *Proceedings of AAAI'97*. AAAI Press/The MIT Press, 221–226.

Järvisalo, M., Oikarinen, E., Janhunen, T., and Niemelä, I. 2009. A module-based framework for multi-language constraint modeling. In *Proceedings of LPNMR'09*. Springer, 155–169.

Lee, J. 2005. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI'05*. Professional Book Center, 503–508.

Lifschitz, V. and Razborov, A. 2006. Why are there so many loop formulas? *ACM Transactions on Computational Logic 7,* 2, 261–268.

Lifschitz, V. and Turner, H. 1994. Splitting a logic program. In *Proceedings of ICLP'94*. 23–37.

Lin, F. and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proceedings of AAAI'02*. AAAI Press/MIT Press, 112–118.

Mellarkod, V. and Gelfond, M. 2008. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of FLOPS'08*. Springer, 15–31.

Mellarkod, V., Gelfond, M., and Zhang, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

Mitchell, D. 2005. A SAT solver primer. *Bulletin of the European Association for Theoretical Computer Science 85*, 112–133.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC01*. ACM, 530–535.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

Niemelä, I., Simons, P., and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *Proceedings of NMR'99*. Springer, 317–333.

Nieuwenhuis, R., Oliveras, A., and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

Ohrimenko, O., Stuckey, P. J., and Codish, M. 2009. Propagation via lazy clause generation. *Constraints 14,* 3, 357–391.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*. 362–367.

Tamura, N., Taga, A., Kitagawa, S., and Banbara, M. 2006. Compiling finite linear CSP into SAT. In *Proceedings of CP'06*. Springer, 590–603.

Walsh, T. 2000. SAT v CSP. In *Proceedings of CP'00*. Springer, 441–456.