Modular Construction of Ground Logic Programs using LPCAT *

Tomi Janhunen

Aalto University School of Science and Technology
Department of Information and Computer Science
PO Box 15400, FI-00076 Aalto, Finland
Tomi.Janhunen@tkk.fi

Abstract. In this paper, we view declarative problem solving from the perspective of answer set programming (ASP). The idea is to solve any given problem by formalizing it as a logic program whose answer sets capture the solutions of the problem. In practice such sets are computed using a special-purpose search engine, viz. an answer set solver, and a ground program obtained by instantiating term variables appearing in the rules of the program. Due to a potential blow-up in the number of rules, the generation of the ground program can become a burden. Since modular program development is gaining more attention in ASP, the objective of this paper to apply modules in the construction of ground logic programs. Our specific goal is to demonstrate that a tool that links together ground program modules can be effective and highly useful when ground programs are generated. In this paper, we provide a formal account of ground program modules and present a link editor, LPCAT, which is designed to be used with SMODELS-compatible grounders and answer set solvers. We study the efficiency of our approach using a benchmark that yields millions of ground rules. Moreover, we illustrate the potential of ground program modules in program transformations.

1 Introduction

In this paper, we view declarative problem solving from the standpoint of answer set programming (ASP) [15, 16, 1] which emerged in the context of logic programming in the 90s. According to this paradigm, any problem of interest is formalized as a logic program whose *answer sets* capture the solutions of the problem. Answer sets are computed in practice using a dedicated search engine, viz. *an answer set solver*, and a ground logic program obtained by instantiating term variables in the rules of the program. Although modern *grounders* such as LPARSE [20] and GRINGO [7] try to reduce the number of resulting ground rules using partial evaluation techniques there is still a potential blow-up in the number of rules—slowing down the computation of answer sets. To address this downside, new reasoning techniques have been developed to circumvent grounding altogether [3]. Another strategy is to rearrange the computation of answer sets as a stepwise process where ground rules are produced for one program

* This research has been partially funded by the Academy of Finland under project "Methods for Constructing and Solving Large Constraint Models" (#122399).

slice at a time and the respective portion of an answer set is then computed *incrementally* [5]. Many application domains suit to this strategy as they involve parameters (such as plan length in AI planning [13]) that induce a natural slicing for problem instances.

On the other hand, modular program development is becoming increasingly important in ASP. Due to the global nature of answer sets, however, it is non-trivial to find an appropriate notion of program modules so that the semantics of a logic program can be directly based on the semantics of its component modules. The early approach based on *splitting sets* [14] is inherently asymmetric: the composition of logic programs out of modules is viewed as an ordered sequence of program unions in the most general setting. The same can be stated about the slicing technique [5] described above due to interleaving grounding and the computation of (partial) answer sets. By contrast, the aim of our work is a symmetric (order-independent) relationship of program modules based on the theory of modules presented in [17] in the case of SMODELS programs. These results provide a basis for a systematic (de)composition of ground programs produced by grounders. The respective operations on ground SMODELS programs have been implemented as tools called MODLIST and LPCAT.[1] The former can be used to split a ground logic program into smallest possible units as defined in [17]. The latter is a *link editor*, or *linker* for short, in the sense of traditional compilation. The tool can be used, among other things, to combine modules produced by MODLIST back into a single entity which is then ready to be processed by an answer set solver.

A specific objective of this paper is to study how a link editor like LPCAT could be exploited in the construction of ground programs in the first place. The idea is to produce the grounding of logic program in separate parts decided by the programmer and then linked together using LPCAT. The file format of the SMODELS system is used as the intermediate representation of modules.[2] Hence the approaches described in this paper are applicable to any SMODELS-compatible grounders and solvers—not just LPARSE and SMODELS. The other grounder mentioned above, GRINGO, meets this criterion. There are also other answer set solvers such as CMODELS [8] and CLASP [6] that are basically SMODELS-compatible but involve language extensions. Some of them can be translated back into primitives included in the SMODELS format.

The tools described in this paper are also applicable in far more versatile ways. They were originally designed for verification purposes, i.e., the problem of deciding whether two programs have exactly the same answer sets. One way to address such a problem is to modularize the task by splitting the two programs under consideration into small components and by checking the equivalence of components [18]. We foresee yet another general strategy for performing modular transformations to programs. The idea is (i) to split the program into its components, e.g., by invoking MODLIST, (ii) to transform each module in turn using transformation-specific tools, and (iii) to compose the result with LPCAT. It is likely that such a strategy will at least save memory since the input programs (modules) for the transforming program will be smaller. Our further

[1] Both tools are available in the ASPTOOLS collection at `http://www.tcs.hut.fi/Software/asptools/`.

[2] The expected benefits and requirements of intermediate representation languages are discussed in detail in [10]. A more recent and more general proposal, viz. *answer set programming intermediate language specification* (ASPILS) can be found in [4].

objective of this paper is to study the time efficiency of this strategy. To this end, a natural requirement is that modularization should not bring about substantial overhead.

The rest of this paper is organized as follows. In Section 2, we provide a brief account of SMODELS programs and, in particular, their modularity properties. The aim is to establish a theoretical background for the tools introduced in the sequel. This takes place in Section 3 where some technical aspects of program (de)composition are discussed. The goal of Section 4 is to present two encodings of the benchmark problem used in this paper, namely the n-queens problem. The first is a basic encoding as an SMODELS program $\mathbf{Q}_n^{\mathrm{sm}}$. The second is a modular one which is obtained by identifying a *slice* $\mathbf{Q}_{n,k}^{\mathrm{mod}}$ for each parameter value $1 \leq k \leq n$. The resulting ground program can be computed as a *join* $\mathbf{Q}_{n,1}^{\mathrm{mod}} \sqcup \ldots \sqcup \mathbf{Q}_{n,n}^{\mathrm{mod}}$ by linking the component programs $\mathbf{Q}_{n,1}^{\mathrm{mod}}$, $\ldots$, and $\mathbf{Q}_{n,n}^{\mathrm{mod}}$ together using LPCAT. The outcome is equivalent with $\mathbf{Q}_n^{\mathrm{sm}}$ but may differ syntactically due to new atoms introduced by grounders. Section 5 is devoted to the performance analysis of contemporary grounders when $\mathbf{Q}_n$ is produced directly or when it is grounded slice-by-slice and linked together using LPCAT. The results indicate systematic savings as regards time. In certain cases, substantial memory savings can be achieved using the modular approach. In Section 6, we present a strategy for performing modular program transformations using LPCAT. Section 7 concludes the paper.

2 Theoretical Background for Program Modules

In this section, we review the syntax and semantics of SMODELS programs [19] and present the notion of program modules from [17] which are both central for the design of LPCAT. Moreover, we recall the basic modularity properties of answer sets [17] which provide the foundation for modular (de)composition of SMODELS programs.

Any SMODELS-compatible grounder is supposed to produce a ground program in an intermediate representation, i.e., the file format originally introduced in the context the SMODELS system. The format is based on a numerical encoding of four rule types:

$$a \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \tag{1}$$

$$\{a_1, \ldots, a_h\} \leftarrow b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m \tag{2}$$

$$a \leftarrow l \leq \{b_1, \ldots, b_n, \sim c_1, \ldots, \sim c_m\} \tag{3}$$

$$a \leftarrow w \leq \{b_1 = w_{b_1}, \ldots, b_n = w_{b_n}, \sim c_1 = w_{c_1}, \ldots, \sim c_m = w_{c_m}\} \tag{4}$$

Rules of the forms above are assumed to be ground already. Hence a, a_i's, b_j's, and c_k's are ground atoms. *Basic rules* of the form (1) are *normal rules* involving default negation (denoted by "$\sim$"). The intuition is that the *head* a is supposed to be true whenever the *body* of the rule is satisfied, i.e., when all b_j's are true and none of c_k's is true by any other rules in the program. The head $\{a_1, \ldots, a_h\}$ of a *choice rule* (2) denotes a specific choice to be made when the body of the rule is satisfied: any of a_i's can be true. The body of a *cardinality rule* (3) is satisfied when the number of satisfied literals is at least l. More generally, the body of a *weight rule* (4) is satisfied if the sum of weights (denoted by w_{b_j}'s and w_{c_k}'s above) associated with satisfied literals is at least w. Using shorthands A, B, and C for the sets of atoms involved in (1)–(4) and $\sim C = \{\sim c \mid c \in C\}$ for any set C of atoms, we obtain abbreviations such as $\{A\} \leftarrow B, \sim C$ for a choice

rule (2). Likewise, a shorthand $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ denotes a weight rule (4) with the respective sets of weights W_B and W_C from (4).

An SMODELS program P is defined as a finite set of ground rules of the forms (1)–(4). Given such a program P, the set of atoms occurring in its rules, i.e., the signature of P, is denoted by $\mathrm{At}(P)$. We encapsulate SMODELS programs in the way proposed in [17]. An SMODELS program module Π is a quadruple $\langle P, I, O, H\rangle$ where

1. P is an SMODELS program,
2. I, O, and H are pairwise disjoint sets of input, output, and hidden atoms;
3. $\mathrm{At}(P) \subseteq \mathrm{At}(\Pi)$ where $\mathrm{At}(\Pi)$ is defined as $I \cup O \cup H$; and
4. $\mathrm{Hd}(P) \cap I = \emptyset$ where $\mathrm{Hd}(P)$ is the set of head atoms of P.

The *visible* part $\mathrm{At_v}(\Pi) = I \cup O$ of $\mathrm{At}(\Pi)$ can be accessed by other modules to supply input for Π or to utilize its output. The *input signature* I and the *output signature* O of Π is also denoted by $\mathrm{At_i}(\Pi)$ and $\mathrm{At_o}(\Pi)$, respectively. The *hidden* atoms in the difference $\mathrm{At_h}(\Pi) = \mathrm{At}(\Pi) \setminus \mathrm{At_v}(\Pi) = H$ can be used to formalize some auxiliary concepts of Π. The fourth requirement of a program module Π ensures that input atoms are only allowed to appear as positive or negative conditions in rule bodies.

Example 1. Consider an SMODELS program module Π having the following rules:

$$d \leftarrow 2 \leq \{a, b, c\}. \qquad e \leftarrow a, b, c. \qquad f \leftarrow {\sim}d, {\sim}f. \qquad f \leftarrow e, {\sim}f.$$

The I/O interface of $\Pi = \langle P, I, O, H\rangle$ is determined by $I = \{a, b, c\}$, $O = \emptyset$, and $H = \{d, e, f\}$. The purpose of Π is to check that exactly two among the input atoms a, b, and c are true. To achieve this, three auxiliary atoms are used. The meaning of d is that at least two input atoms are true as formalized by the first rule. The second rule makes e true only if all input atoms are true simultaneously. The last two rules are effectively *constraints* that deny ${\sim}d$ and e, i.e., d and e must be true and false, respectively. In this manner, we obtain the desired net effect: exactly two input atoms are true. ■

Let us now turn our attention to the semantics of program modules which cover also ordinary SMODELS programs as their special case, i.e., when $\mathrm{At_i}(\Pi) = \emptyset = \mathrm{At_h}(\Pi)$. The semantics of default negation goes back to [19] whereas the treatment of input atoms is based on [12, 17]. Both aspects are simultaneously covered by the following definition and the resulting program does not contain input atoms nor negative literals.

Definition 1 ([12, 17]). *Given a program module $\Pi = \langle P, I, O, H\rangle$, the* reduct *of P with respect to a set $S \subseteq \mathrm{At}(\Pi)$ and the input signature I, denoted by $P^{S,I}$, contains*

1. *a basic rule $a \leftarrow (B \setminus I)$ if and only if there is a basic rule $a \leftarrow B, {\sim}C$ in P such that $B \cap I \subseteq S$, and $S \cap C = \emptyset$;* or *there is a choice rule $\{A\} \leftarrow B, {\sim}C$ in P such that $a \in A \cap S$, $B \cap I \subseteq S$, and $S \cap C = \emptyset$;*
2. *a cardinality rule $a \leftarrow l' \leq \{B \setminus I\}$ if and only if there is a cardinality rule $a \leftarrow l \leq \{B, {\sim}C\}$ in P and $l' = \max(0, l - |B \cap I \cap S| - |C \setminus S|)$; and*
3. *a weight rule $a \leftarrow w' \leq \{B \setminus I = W_{B \setminus I}\}$ if and only if there is a weight rule $a \leftarrow w \leq \{B = W_B, {\sim}C = W_C\}$ in P and*

$$w' = \max(0, w - \textstyle\sum_{b \in B \cap I \cap S} w_b - \sum_{c \in C \setminus S} w_c).$$

Given any $\Pi = \langle P, I, O, H\rangle$ and $S \subseteq \mathrm{At}(\Pi)$, the reduced program $P^{S,I}$ is monotonic and thus it has a unique closure[3] $\mathrm{Cl}(P^{S,I}) \subseteq O \cup H$ by Knaster-Tarski lemma.

Definition 2. *A set $S \subseteq \mathrm{At}(\Pi)$ is an answer set of an* SMODELS *program module $\Pi = \langle P, I, O, H\rangle$, denoted by $S \in \mathrm{ASet}(\Pi)$, if and only if $S \setminus I = \mathrm{Cl}(P^{S,I})$.*

Example 2. The module Π from Example 1 has three answer sets in total, i.e., $\mathrm{ASet}(\Pi)$ equals to $\{\{a, b, d\}, \{a, c, d\}, \{b, c, d\}\}$. To verify that $S = \{a, b, d\}$ is indeed an answer set, we note that $P^{S,I} = \{d \leftarrow 0 \leq \{\}.\ \ f \leftarrow e.\ \}$ for which $\mathrm{Cl}(P^{S,I}) = \{d\} = S \setminus I$. On the other hand, the set $S' = \{a, b, c, d\}$ is not an answer set as $P^{S',I}$ consists of $d \leftarrow 0 \leq \{\}$, $e \leftarrow$, and $f \leftarrow e$. Then $\mathrm{Cl}(P^{S',I}) = \{d, e, f\} \neq S' \setminus I = \{d\}$. ■

Unfortunately, answer sets as defined above do not provide SMODELS program modules with a fully *compositional* semantics. For instance, taking straightforward unions of programs is not sufficient to guarantee that answer sets assigned to the union could be obtained by combining answer sets of its members. This is why we resort to Gaifman-Shapiro-style criteria for program composition as put forth in [17]. Two modules Π_1 and Π_2 are eligible for composition only if their output signatures are disjoint and they *respect each other's hidden atoms*, i.e. $\mathrm{At_h}(\Pi_1) \cap \mathrm{At}(\Pi_2) = \emptyset$ and $\mathrm{At_h}(\Pi_2) \cap \mathrm{At}(\Pi_1) = \emptyset$. The outcome of composing Π_1 and Π_2 is defined as follows.

Definition 3 ([17]). *The composition of program modules $\Pi_1 = \langle P_1, I_1, O_1, H_1\rangle$ and $\Pi_2 = \langle P_2, I_2, O_2, H_2\rangle$, denoted by $\Pi_1 \oplus \Pi_2$, is*

$$\langle P_1 \cup P_2, (I_1 \setminus O_2) \cup (I_2 \setminus O_1), O_1 \cup O_2, H_1 \cup H_2\rangle \tag{5}$$

if $\mathrm{At_o}(\Pi_1) \cap \mathrm{At_o}(\Pi_2) = \emptyset$ and Π_1 and Π_2 respect each other's hidden atoms.

Example 3. Consider another SMODELS program module $\Pi' = \langle P', I', O', H'\rangle$ where P' contains a single choice rule $\{a, b, c\} \leftarrow$ and $I' = \emptyset$, $O' = \{a, b, c\}$, and $H' = \emptyset$. The composition of Π from Example 1 with Π' is defined since $O \cap O' = \emptyset$, $H' = \emptyset$ and Π' does not mention atoms from H. The resulting composition $\Pi \oplus \Pi'$ has an I/O interface based on the sets of atoms $\emptyset$, $\{a, b, c\}$, and $\{d, e, f\}$, respectively. ■

As demonstrated in [17], the conditions of Definition 3 do not yet imply the desired relationship of answer sets in general. The conditions can be suitably tightened using the *positive dependency graph* of the composition $\Pi_1 \oplus \Pi_2$. Generally speaking, the positive dependency graph $\mathrm{DG^+}(\Pi)$ associated with an SMODELS program module $\Pi = \langle P, I, O, H\rangle$ is the pair $\langle O \cup H, \leq\rangle$ where $b \leq a$ holds for any atoms a and b of $O \cup H$ if and only if a appears in the head of a rule of P so that $b \in B$. A *strongly connected component* (SCC) S of $\mathrm{DG^+}(P)$ is a maximal set $S \subseteq \mathrm{At}(P)$ such that $b \leq^* a$ holds for every $a, b \in S$, i.e., all atoms of S depend positively on each other. If the composition $\Pi_1 \oplus \Pi_2$ is defined, the members of the composition are *mutually dependent* if and only if $\mathrm{DG^+}(\Pi_1 \oplus \Pi_2)$ has an SCC S such that $S \cap \mathrm{At_o}(\Pi_1) \neq \emptyset$ and $S \cap \mathrm{At_o}(\Pi_2) \neq \emptyset$, i.e., the SCC in question is effectively *shared* by Π_1 and Π_2.

[3] The least set S' of atoms such that $S' \subseteq O \cup H$ and S' is closed under the rules of $P^{S,I}$.

Definition 4 ([17]). *The* join $\Pi_1 \sqcup \Pi_2$ *of two* SMODELS *program modules* Π_1 *and* Π_2 *is* $\Pi_1 \oplus \Pi_2$*, provided* $\Pi_1 \oplus \Pi_2$ *is defined and* Π_1 *and* Π_2 *are mutually* independent.

The key observation from [17] is that positive recursion cannot be tolerated across module boundaries. The proviso of Definition 4 is sufficient to guarantee that, roughly speaking, a (global) answer set of an SMODELS program is also a (local) answer set of its modules and vice versa. The following theorem characterizes the exact relationship of answer sets for the join $\Pi_1 \sqcup \Pi_2$ and its component modules Π_1 and Π_2.

Theorem 1 (Module Theorem [17]). *If* Π_1 *and* Π_2 *are* SMODELS *program modules such that* $\Pi_1 \sqcup \Pi_2$ *is defined, then* $\mathrm{ASet}(\Pi_1 \sqcup \Pi_2) = \mathrm{ASet}(\Pi_1) \bowtie \mathrm{ASet}(\Pi_2)$.

In the above, the operation $\bowtie$ denotes a *natural join* of *compatible* answer sets, i.e., $S_1 \cup S_2$ belongs to $\mathrm{ASet}(\Pi_1) \bowtie \mathrm{ASet}(\Pi_2)$ if and only if $S_1 \in \mathrm{ASet}(\Pi_1)$, $S_2 \in \mathrm{ASet}(\Pi_1)$, and $S_1 \cap \mathrm{At_v}(\Pi_2) = S_2 \cap \mathrm{At_v}(\Pi_1)$. Theorem 1 is easily generalized for finite joins of modules: if $\Pi_1 \sqcup \cdots \sqcup \Pi_n$ is defined, then

$$\mathrm{ASet}(\Pi_1 \sqcup \cdots \sqcup \Pi_n) = \mathrm{ASet}(\Pi_1) \bowtie \cdots \bowtie \mathrm{ASet}(\Pi_n). \tag{6}$$

Equation (6) provides a semantical basis for the link editor LPCAT to be described in detail in the next section. Given SMODELS program modules $\Pi_1, \ldots, \Pi_n$ which (i) respect each other's hidden atoms, (ii) have distinct output signatures, and (iii) are mutually independent, the tool can be used to safely compute their composition. The answer sets of the resulting ground SMODELS program are then governed by (6).

Example 4. Recall SMODELS program modules Π and Π' from Examples 1 and 3. The set $\mathrm{ASet}(\Pi)$ is listed in Example 2 whereas it is clear that $\mathrm{ASet}(\Pi') = \mathbf{2}^{\{a,b,c\}}$. For instance, $T = \{a, b\}$ belongs to this set, because $(P')^{T,I'} = \{a \leftarrow .\ \ b \leftarrow .\ \}$ by Definition 1. Note that S from Example 2 and T are mutually compatible, as $S \cap \mathrm{At_v}(\Pi') = \{a, b\} = T \cap \mathrm{At_v}(\Pi)$, and $S \cup T = S$. It follows that $\mathrm{ASet}(\Pi \oplus \Pi') = \mathrm{ASet}(\Pi) \bowtie \mathrm{ASet}(\Pi') = \mathrm{ASet}(\Pi)$ by generalization. This reflects the fact that the join $\Pi \sqcup \Pi'$ is defined and Theorem 1 holds for the modules Π and Π' under consideration. ■

Finally, let us discuss how non-ground programs fit into this scenario. Given a set of non-ground rules P, we write $\mathrm{Gnd}(P)$ for the resulting ground SMODELS program module. Because non-ground rules typically involve language extensions in addition to term variables, we leave the exact definition of $\mathrm{Gnd}(\Pi)$ open and refer the reader to [20] for a comprehensive syntax. To exploit the theory of modules presented so far we assume that for each non-ground program P, the semantics of P is determined by the set $\mathrm{ASet}(\mathrm{Gnd}(P))$ where $\mathrm{Gnd}(P)$ contains only rules of the forms (1)–(4). In the sequel, our strategy to compute $\mathrm{Gnd}(P)$ is to split P into parts $P_1, \ldots, P_n$ so that $\mathrm{Gnd}(P_1), \ldots, \mathrm{Gnd}(P_n)$ can be computed in separation, the join $\mathrm{Gnd}(P_1) \sqcup \ldots \sqcup \mathrm{Gnd}(P_n)$ is defined, and it effectively[4] equals to $\mathrm{Gnd}(P)$ for the original program P. Hence $\mathrm{ASet}(\mathrm{Gnd}(P_1)) \bowtie \cdots \bowtie \mathrm{ASet}(\mathrm{Gnd}(P_n))$ essentially captures the set $\mathrm{ASet}(\mathrm{Gnd}(P))$ of answer sets associated with P. In contrast to the incremental approach of [5], the strategy just described is totally symmetric and, basically, the join $\mathrm{Gnd}(P_1) \sqcup \ldots \sqcup \mathrm{Gnd}(P_n)$ can be computed in any order.

[4] The new (hidden) atoms inserted by grounders may lead to slight syntactic differences, though.

3 Practical Issues of the Implementation

The objectives of this section are twofold. First, we describe how the notion of SMODELS program modules, as outlined in Section 2, can be realized in practice. Second, we discuss the main design decisions behind the link editor LPCAT.

It is natural to assume that the *source code* of an ASP module is written in the input language of the grounder to be used—exploiting term variables and language extensions as appropriate. However, for compatibility reasons, we expect that the resulting ground program is in the file format of the SMODELS system. Note that such a representation is analogous to an *object module* in conventional programming languages. Recalling the general structure $\langle P, I, O, H\rangle$ of SMODELS program modules from Section 2, this is how we obtain the set P of ground rules. In order to control the visibility of atoms, both LPARSE and GRINGO support *hide* and *show* declarations. The atoms which are not hidden will have a name in a symbol table which accompanies the internal representation of rules (in terms of natural numbers). In practice, the table provides a partial mapping from natural numbers to symbolic names of atoms.

What remains is the distinction between the sets I and O. Input atoms can be simulated in terms of *external/non-domain*[5] predicates if LPARSE is used but such declarations are not recognized by GRINGO for the moment. A further obstacle is that the intermediate file format used by SMODELS-compatible tools does not allow the specification of input atoms either. This shortage is fixed in the ASPILS proposal [4] but the format in question is not widely understood by ASP systems yet. To resolve this issue pragmatically, we have coordinated an extension of the SMODELS file format which enables the declaration of external (input) atoms in the same way as *compute statements* are used to assign truth values to particular atoms. This enables the reliable transmission of external atoms from a grounder to solvers and other tools. Note that a typical answer set solver would assign input atoms false by default in the absence of defining rules.

To help with interfacing programs involving external atoms with existing systems, we provide a simple program called IGEN which replaces such declarations by an *input generator*. This amounts to adding a choice rule (2) with $n = 0$ and $m = 0$ for the entire set of input atoms $I = \{a_1, \ldots, a_h\}$ in a module $\Pi = \langle P, I, O, H\rangle$. This construction was actually illustrated by Examples 3 and 4: the module Π' plays the role of an input generator and $\Pi \oplus \Pi'$ can be viewed as a conventional SMODELS program where d, e, and f are hidden. In this way, one can compute stable models for individual modules in separation from the rest of the answer set program and it is not necessary to modify the source code of a module to achieve this.

We turn our attention to another application of program modules now and consider the composition of larger (ground) programs in terms of the operator $\oplus$ from (5). We have implemented this functionality as a *link editor* called LPCAT whose design is discussed next. Additionally, the tool checks the *module condition* $\mathrm{At_o}(\Pi_1) \cap \mathrm{At_o}(\Pi_2) = \emptyset$ from Definition 3 upon request. The current version leaves the check for mutual independence at the programmer's responsibility. This is merely a design decision to save memory: the memory reserved by the rules of a module can be deallocated as soon as

[5] Consult `http://www.tcs.hut.fi/Software/smodels/lparse.ps` for details.

they have been written out. This would not be possible if the SCCs of the resulting program were incrementally constructed. The main tasks of LPCAT are the following.

1. The symbolic names of (visible) atoms given in symbol tables are matched. This accounts for the computation of $(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$ in Equation (5). Thus an atom which is defined by a module and used by another will have a unique atom number in the resulting symbol table. This functionality is implemented in terms of a standard hash table which also stores the natural number associated with each name. Thus, when the hash table is consulted to check whether a particular name is already in use and this turns out to be the case, the respective atom number can be returned. This number is needed for subsequent substitutions in the following step.
2. Each module is *relocated*, i.e., the atom numbers used in the intermediate representation of rules are replaced by contiguous values from $n + 1$ to $n + m$ where n is the number of atoms encountered so far and m is the number of distinct atoms in the module not seen yet. Of course, the idea is that the numbers of atoms that are encountered again remain unchanged in the relocation process. Such atoms have already a number between 1 and n.

The output of LPCAT uses contiguous numbers for atoms in the range $1 \ldots n$ where n is the number of distinct atoms. Hence all unused atom numbers will be effectively removed and the symbol table gets compressed. This will most likely reduce the amount of memory reserved by the solver which, in turn, can also favor running times.

Large programs can involve many modules and storing them in separate files can become a burden of its own. This is especially the case if existing ground programs are afterwards split into modules, e.g., using SCCs as the criterion. As a result, there might be simply too many files to handle: even basic shell commands may fail to the user's surprise if thousands of arguments are provided (consider, e.g., "`ls *`" in this respect). To deal with huge numbers of modules, LPCAT is able to read in modules recursively from file or, perhaps more conveniently, from streams. Then, at least in certain cases, expensive file operations can be avoided altogether, e.g., if *shell pipelines* are used for passing modules around. Applications of this feature are addressed in Section 6.

4 Encoding and Slicing the n-Queens Problem

The famous n-queens problem is related to the game of chess: the goal is to place simultaneously n queen pieces on a $n \times n$ chess board so that they do not threaten each other. In this section, we present a basic encoding as an SMODELS program $\mathbf{Q}_n^{\mathrm{sm}}$ that goes back to [16]. However, we will express the placement of queens using choice rules (2) rather than basic/normal rules for better readability. The use of other rule types such as cardinality rules will be discussed in the end of this section. In addition to the basic encoding, we develop a sliced version of $\mathbf{Q}_n^{\mathrm{sm}}$ which consists of modules $\mathbf{Q}_{n,k}^{\mathrm{mod}}$ where the parameter $1 \leq k \leq n$. Roughly speaking, the rules in $\mathbf{Q}_{n,k}^{\mathrm{mod}}$ formalize choices and constraints related to the k^{th} row and the k^{th} column of the chess board.

We decided to use the n-queens problem in this paper for several reasons. First, the problem definition is simple and intuitively clear and thus the domain lends itself for

1. $\mathsf{d}(1). \quad \ldots \mathsf{d}(n).$
2. $\{\mathsf{q}(X,Y)\} \leftarrow \mathsf{d}(X), \mathsf{d}(Y).$
3. $\mathsf{u} \leftarrow \mathsf{q}(X,Y_1), \mathsf{q}(X,Y_2), \mathsf{d}(X), \mathsf{d}(Y_1), \mathsf{d}(Y_2), Y_1 < Y_2.$
4. $\mathsf{u} \leftarrow \mathsf{q}(X_1,Y), \mathsf{q}(X_2,Y), \mathsf{d}(X_1), \mathsf{d}(X_2), \mathsf{d}(Y), X_1 < X_2.$
5. $\mathsf{u} \leftarrow \mathsf{q}(X_1,Y_1), \mathsf{q}(X_2,Y_2), \mathsf{d}(X_1), \mathsf{d}(X_2), \mathsf{d}(Y_1), \mathsf{d}(Y_2),$
   $X_1 < X_2, |X_1 - X_2| = |Y_1 - Y_2|.$
6. $\mathsf{hasq}(X) \leftarrow \mathsf{q}(X,Y), \mathsf{d}(X), \mathsf{d}(Y).$
7. $\mathsf{u} \leftarrow \sim\mathsf{hasq}(X), \mathsf{d}(X).$
8. $\mathsf{f} \leftarrow \mathsf{u}, \sim\mathsf{f}.$

Fig. 1. An encoding of the n-queens problem using basic and choice rules

easy illustration. Second, there is a natural parameter, the number of queens n, involved which affects the complexity of the problem in a non-linear way. Third, although the problem is easy to solve for small numbers of queens using ASP techniques, size factors come into play when the number of queens is increased up to 100 or more. Then it is no longer easy to deal with resulting ground programs which involve millions of rules. Fourth, variants of the n-queens problem are standard benchmarks in ASP competitions.

A program formalizing the n-queens problem is presented in Figure 1. A syntax that is close to the input syntax of LPARSE is used. We exploit term variables X, Y, etc. in the rules but, otherwise, try to use rules of the forms (1) and (2) as far as possible. In the encoding, predicate $\mathsf{d}(\cdot)$ is used as a domain predicate to define valid coordinate values $1 \ldots n$ for the n queens to be placed on the board. These predicates are evaluated by LPARSE and effectively removed from the resulting ground program. The same applies to extra conditions expressed in terms of infix operators $<$ and $=$. The domain of $\mathsf{d}(\cdot)$ is specified on line 1. The choice whether there is a queen in square (X,Y) or not is stated on line 2. Rules on line 3–5 formalize the constraints involving the columns, rows, and diagonals of the chess board, respectively. The atom u denoting *unsatisfiability* is derived in case of any two queens threatening each other. Rules listed on lines 6–7 ensure that there is at least one queen in each column X of the board. The last rule excludes answer sets containing u, i.e., those violating the constraints governing the problem. In practice, the program can be written without the new atom u and the last rule. In other words, rules without heads can be used directly to formalize constraints.

Our next objective is to find a way of splitting the encoding above in n slices parameterized by $1 \leq k \leq n$. To this end, we introduce a *restricted* domain $\mathsf{rd}(\cdot)$ for the coordinate values $1 \ldots k-1$. Actually, this can be defined in terms of a rule

$$\mathsf{rd}(X) \leftarrow \mathsf{d}(X), X < k.$$

Then placements on the k^{th} column and the k^{th} row are captured by these choice rules:

$$\{\mathsf{q}(k,Y)\} \leftarrow \mathsf{rd}(Y). \qquad \{\mathsf{q}(k,k)\}. \qquad \{\mathsf{q}(X,k)\} \leftarrow \mathsf{rd}(X).$$

In addition to rules above, it is necessary to declare $\mathsf{q}(\cdot,\cdot)$ as an external predicate, since the choices regarding other columns and rows are formalized by other slices of the program. The column-wise constraints involving k^{th} squares in a column/row are:

$$\mathsf{u} \leftarrow \mathsf{q}(k, Y_1), \mathsf{q}(k, Y_2), \mathsf{rd}(Y_1), \mathsf{rd}(Y_2), Y_1 < Y_2.$$
$$\mathsf{u} \leftarrow \mathsf{q}(X, Y), \mathsf{q}(X, k), \mathsf{d}(X), \mathsf{rd}(Y).$$

Due to obvious symmetry between rows and columns, we omit the required row-wise constraints. As regards diagonals, the constraints incident with k^{th} squares are:

$$\mathsf{u} \leftarrow \mathsf{q}(X_1, k), \mathsf{q}(X_2, Y_2), \mathsf{rd}(X_1), \mathsf{rd}(X_2), \mathsf{rd}(Y_2), |X_1 - X_2| = |k - Y_2|.$$
$$\mathsf{u} \leftarrow \mathsf{q}(k, Y_1), \mathsf{q}(X_2, Y_2), \mathsf{rd}(Y_1), \mathsf{rd}(X_2), \mathsf{rd}(Y_2), |k - X_2| = |Y_1 - Y_2|.$$
$$\mathsf{u} \leftarrow \mathsf{q}(X, X), \mathsf{q}(k, k), \mathsf{rd}(X).$$
$$\mathsf{u} \leftarrow \mathsf{q}(k, Y), \mathsf{q}(Y, k), \mathsf{rd}(Y).$$

Last, we slice constraints that ensure the existence of at least one queen on each column:

$$\mathsf{hasq}(X) \leftarrow \mathsf{q}(X, k), \mathsf{rd}(X). \qquad \mathsf{hasq}(k) \leftarrow \mathsf{q}(k, Y), \mathsf{d}(Y). \qquad \mathsf{u} \leftarrow \sim\mathsf{hasq}(k).$$

If we compare the rules derived above with the original ones in Figure 1, the number of source code lines is roughly doubled due to slicing. This goes back the fact that the problem is inherently two-dimensional. To create a ground SMODELS program for the n-queens problem for a particular value of n, the rules derived above are to be grounded for each $1 \leq k \leq n$ in turn and linked together using LPCAT. The efficiency of this strategy will be addressed in Section 5 using the encoding and slicing described above.

It is worth pointing out that there are also alternative ways to encode the n-queens problem. If cardinality rules (3) are used, it is very easy to formulate that, for instance, exactly one queen is placed on each column X:

$$\mathsf{u} \leftarrow 2 \leq \{\mathsf{q}(X, 1), \ldots, \mathsf{q}(X, n)\}. \qquad \mathsf{s} \leftarrow 1 \leq \{\mathsf{q}(X, 1), \ldots, \mathsf{q}(X, n)\}.$$
$$\mathsf{f} \leftarrow \mathsf{u}, \sim\mathsf{f}. \qquad \mathsf{f} \leftarrow \sim\mathsf{s}, \sim\mathsf{f}.$$

In words, an assignment is to be disqualified if there are two or no queens on column X. We did not resort to rules of this kind because they do not lend themselves for slicing in a straightforward way. For instance, if the rule for s above is to be reused in the formulation of $\mathsf{s} \leftarrow 1 \leq \{\mathsf{q}(X, 1), \ldots, \mathsf{q}(X, n+1)\}$ for increased problem size $n+1$, we end up writing normal rules recursively in the way we did for $\mathsf{hasq}(\cdot)$. Thus we note that there are more concise encodings of the n-queens problem in sight if cardinality constraints are considered. But, since finding a space-optimal SMODELS program for the problem is not the main issue of this paper, we omit the study of such encodings.

The encoding presented above serves as an example of our slicing approach but for a particular problem. To grasp other potential application domains of the same idea but on a more general level, we refine the description given in the end of Section 2. As presented therein, the slicing $P_1, \ldots, P_n$ of a program P is based on a certain parameter $1 \leq i \leq n$ that is specific to the problem being solved. To make the role of domain predicates explicit, we assume that each slice P_i consists of the domain part D_i and the rest $Q_i = P_i \setminus D_i$ so that $\mathrm{Gnd}(D_i)$ is always a set of facts. As demonstrated above, Q_i's are likely to be disjoint whereas domains grow monotonically: $\mathrm{Gnd}(D_1) \subseteq \ldots \subseteq \mathrm{Gnd}(D_n)$. Thus it may be worthwhile to delete instances of domain predicates from the ground programs as possible with LPARSE. Then $\mathrm{Gnd}(P)$ is obtained as the join of $\mathrm{Gnd}(Q_i \cup D_i) \setminus \mathrm{Gnd}(D_i)$ for each $1 \leq i \leq n$ and $\mathrm{Gnd}(D)$ where $D = \bigcup_{i=1}^{n} D_i$.

n	50	100	150	200	250	unit
number of atoms	2.6k	10k	23k	40k	63k	
number of rules	210k	1.7M	5.6M	13M	26M	
LPARSE-1.0.17	2.26	28.1	130	388	939	s
	12	79	260	610	1200	MB
LPARSE-1.0.17+LPCAT-1.17	2.25	**21.9**	**89.7**	**248**	**555**	s
	-	8.6	26	56	100	MB
LPARSE-1.1.2	9.30	86.1	325	845	1850	s
	3.4	3.9	**4.8**	**6.0**	**7.1**	MB
LPARSE-1.1.2+LPCAT-1.17	9.27	78.5	279	693	1420	s
	-	6.7	26	56	94	MB
GRINGO-2.0.5	**2.17**	29.1	140	417	998	s
	2.3	**3.6**	5.6	8.5	11	MB
GRINGO-2.0.5+LPCAT-1.17	2.03	22.1	95	275	631	s
	-	-	-	-	-	MB

Table 1. Time and memory resources used by various grounding strategies

5 Performance Analysis

The goal of this section is to evaluate the effectiveness of various grounding strategies using the two encodings introduced in Section 4. As regards grounders, we will use two SMODELS-compatible grounders available today: LPARSE (versions 1.0.17 and 1.1.2) and GRINGO (version 2.0.5). The reason for considering two different versions of LPARSE is that the treatment of symbolic names changed considerably since the last version of 1.0.* series. There is also another experimental grounder, BINGO, being developed as part of the Potassco[6] collection. It was recently extended to support the declaration of external predicates in a very refined way: even particular instances of a predicate can be assumed to be defined outside the current module. However, we had to exclude BINGO from our experiments due to its restricted input language.

In our benchmark, the idea is to first ground the basic encoding $\mathbf{Q}_n^{\mathrm{sm}}$ using the three grounders for the values of n in the range $50 \ldots 250$ using an increment of 50 queens. Then the second representation will be used to form the corresponding ground program $\mathbf{Q}_n^{\mathrm{mod}}$ in slices which are finally linked together using LPCAT. Again, each of the three grounders will be responsible for grounding the required slices one at a time. As the hardware we use Intel Core2 1.86 Ghz CPUs with 2 GBs of main memory. Running times are measured by the Linux `/usr/bin/time` utility whereas sizes of processes are perceived by taking snapshots of `/proc/*/stat` ten times per second. Because GRINGO does not support external predicates for the moment we have to estimate its running time when grounding the sliced program. In order to prevent GRINGO from dropping relevant ground rules involving external predicates, we add an extra choice rule $\{\mathsf{q}(X,Y)\} \leftarrow \mathsf{rd}(X), \mathsf{rd}(Y)$ to the k^{th} slice. Afterwards, we subtract the time elapsed when grounding and linking these extra rules for each $1 \leq k \leq n$.

[6] See `http://potassco.sourceforge.net/` for details.

The results of our grounding benchmark have been collected in Table 1. The used memory cannot be calculated accurately when $n = 50$ and LPCAT is used or when extra rules have to be used with GRINGO (hence the dashes in certain entries). Moreover, we expect errors up to 10% in size measurements due to snapshots. The following observations can be made. The old version of LPARSE consumes a lot of memory as it gradually creates data structures for the newly introduced atoms. However, if slicing and LPCAT are used, the effects of this activity are limited to each slice and the symbol table cumulated by LPCAT reserves roughly one tenth of memory compared to LPARSE-1.0.17. This favors also running times so that the grounding strategy which uses slicing, LPARSE-1.0.17, and LPCAT scales best. The new version of LPARSE (1.1.2) spends more time on grounding but uses least space. A reduction is obtained, too, if slicing and LPCAT are used. It is interesting that GRINGO scales almost identically with LPARSE-1.0.17 but eventually uses only slightly more memory than LPARSE-1.1.2. Our estimates suggest that running times can be reduced using the slicing strategy but not quite as much as in the case of LPARSE-1.0.17.

In summary, we conclude that the slicing technique leads to systematic speed-up in this benchmark. In the n-queens benchmark, the combination of LPARSE-1.0.17 and LPCAT-1.17 leads to the best performance as regards time whereas GRINGO-2.0.5 is able to operate in the least amount of memory. There is a further benefit of the modular approach demonstrated in this section. Since the largest instance $\mathbf{Q}_{250}^{\mathrm{mod}}$ includes all smaller instances as its proper submodules, these instances can be created as by-products of generating the largest one. This takes only a couple of seconds more for file I/O compared to the cumulative time required to produce $\mathbf{Q}_{250}^{\mathrm{mod}}$ from slices. Following the incremental approach from [5], the resulting intermediate ground programs instances can be used for solving purposes if appropriate for the problem domain. By contrast, there is no natural way to the reuse of ground programs produced for smaller numbers of queens if the basic encoding from Section 4 is used.

6 Modular Program Transformations

So far we have demonstrated the use of our linker in the construction of ground logic programs out of modules which have been grounded separately. In this section, we look at another application of LPCAT when the goal is to transform ground logic programs in a systematic fashion, e.g., in order to simplify or to translate them.

The idea is to apply any transformation of interest *module by module* rather than to the entire ground program, and eventually link the transformed programs together. If LPCAT is to be used for linking, the only requirement is that the outcome of the transformation can be represented in the SMODELS format. Hence, for instance, translations within the SMODELS format are easily covered. But if the target format is different, then it becomes necessary to implement a link editor for that format. To implement the scenario just sketched we need two further tools in addition to the link editor itself:

1. A tool for splitting ground SMODELS programs into modules such as MODLIST described in the introduction. A drawback of the current implementation is that it tries to produce as small modules as possible and hence the number of modules can get very high for large program instances. This may slow down the linking

phase and for this reason, we intend to implement alternative ways of splitting ground logic programs into fewer modules. The user could, specify in advance that a particular instance is to be split, e.g., in 100 pieces of roughly equal size—each of which is then transformed in turn. On the other hand, if the ground program is produced in slices as in Section 4, then modules for the transformation are readily available before linking and splitting tools, or *splitters* for short, are not needed.

2. A tool for running the transforming program for each module in a stream of modules in turn. We have implemented this functionality as a new tool MODRUN (version 1.3) which reads modules from its standard input one at a time and invokes the given shell command for each module. The output of MODRUN concatenates the outputs of the commands executed for each module. If they are SMODELS program modules, then LPCAT can be invoked in order to compute their composition.

We present three shell pipelines below for the sake of illustrating how the tools described above are integrated with other ASP tools. It is assumed that a hypothetical tool, TRANSFORM, implements the desired program transformation for SMODELS programs. The first command line shows how a monolithic program is grounded using LPARSE and then transformed. The second example produces a stream of modules using MODLIST, runs TRANSFORM on each of these, and joins the results of individual transformations with LPCAT. It is perhaps worth mentioning that the intermediate results are not stored in physical files and are forwarded very efficiently through standard I/O streams. The third pipeline begins with an (incompletely specified) loop that grounds a bunch of programs using LPARSE and forwards the resulting stream of modules for similar processing as in the second case. Any of the given pipelines could be extended by a call to a solver if the computation of answer sets were of interest straight away.

```
$ lparse program.lp | transform
$ lparse program.lp | modlist | modrun transform | lpcat -r
$ ( for f in p1.lp p2.lp ... ; do lparse $f; done ) \
  | modrun transform | lpcat -r
```

To do some experiments in this direction, we decided to try out a couple of transformations that have been previously implemented for SMODELS programs. The first, LP2NORMAL (version 1.9) [11], translates a ground SMODELS program into a program having only basic/normal rules. This transformation can be done on a rule-by-rule basis. The second transformation, implemented by LP2ATOMIC (version 1.15) [9], removes all positive body literals, i.e., $b_1, \ldots, b_n$ in (1), appearing in the rules of a normal program.[7] The second is potentially more demanding transformation as it is non-linear (of the order of $m \log m$) in the worst case, but the transformation stays linear since our encoding of the n-queens problem does not involve positive recursion. Nevertheless, the SCCs of the program must be computed by LP2ATOMIC to realize this. We use LPARSE-1.1.2 to create ground program (modules) needed in these experiments in the same way as done in Section 5. The results are collected in Table 2 and the data for LPARSE-1.1.2 (both with and without LPCAT-1.17) in Table 1 give baselines for comparison. Running times scale very similarly although translations are incorporated. The good news is that modularized variants become slightly faster as program instances grow.

[7] In the terminology of [9], *atomic* rules are normal rules of the form $a \leftarrow \sim c_1, \ldots, \sim c_m$.

Transformation/number of nodes	50	100	150	200	250
LP2NORMAL	9.8	90.4	334	869	1920
LP2NORMAL+LP2ATOMIC	10.3	94.8	351	912	1950
MODRUN+LP2NORMAL+LPCAT	10.1	82.9	294	728	1500
MODRUN+LP2NORMAL+LP2ATOMIC+LPCAT	10.9	88.9	315	780	1590

Table 2. Running times in seconds for nonmodular vs. modular program transformations

An obvious benefit of the modular approach is that the transformation can be computed using far less memory. The consumption of memory becomes soon crucial if extremely large program instances with millions of rules are dealt with. An instance that reserves the whole memory of a CPU may cause a lot of page faults and force the virtual memory system to thrash—slowing down computations considerably. In this respect, consider the task of translating a ground program instance which reserves 2GBs of memory. By splitting the program into 200 modules, each module fits into roughly 10M of memory. Such instances fit easily into memory on modern workstations and even more expensive transformations may become feasible. Note that it is also possible to reduce the resulting programs before linking them together. To this end, many ASP solvers offer this kind of functionality (see options `-internal` and `--pre` in SMODELS and CLASP, respectively). Such functionalities can be used to relieve worst-case behaviors of translators which are often relatively unoptimized in order to ensure soundness. The modular approach described herein enables a programmer to implement a transformation without worrying too much about modules which can be *externalized* from the tool design using auxiliary tools such as MODLIST, MODRUN, and LPCAT.

7 Discussion

One of the high-level goals of our work is to bring good software engineering practice to the realm of ASP. In this respect, we are especially interested in the potential applications of modularity in the development of answer set programs. Our previous theoretical work on this aspect [12, 17] is tightly connected to and motivated by the development of tool support for exploiting modularity in the context of SMODELS system. In this paper, we are especially interested in the construction of very large ground program instances up to millions of rules. We propose a modular method according to which programs are grounded in slices and the resulting program modules are linked together using a link editor. We have demonstrated that our tool prototype LPCAT is already effective for such purposes using a benchmark problem where the number of ground rules varies from 200 thousand to 26 million. In the experiment, the largest instances reserved roughly 100MB of memory which indicates that even larger instances could be created given more time. When stored in a file such instances take 590MBs disk space (but only 90MBs if compressed by GZIP). A particularly new feature of our approach is the support for streams of modules. This is a new way of exploiting intermediate file formats (see [10] for a detailed analysis) in ASP and it offers a very efficient means to transfer large numbers of modules from a phase of computation to another.

It should be emphasized that the techniques demonstrated in this paper are not limited to the SMODELS system. Other SMODELS-compatible solvers, such as CLASP and CMODELS, are also supported. In addition to modular grounding, there are also other use cases of LPCAT such as query evaluation and, in particular, when used in conjunction with MODLIST for modular processing of SMODELS programs. It is also possible to combine arbitrary SMODELS programs together having no atoms in common using LPCAT. The resulting program will have the Cartesian product of the sets of answer sets of its component programs as its set of answer sets, i.e., $\bowtie$ coincides with $\times$ in (6).

As regards future work, we are looking forward to new releases of the grounders LPARSE and GRINGO that would natively support external (input) atoms using the extension of the SMODELS format described in Section 3. This would enable the full application of the tools reported in this paper. Until now, we have been obliged to *simulate* input atoms by assuming that an input atom a has a name in in the symbol table of the program but no *defining rules*, i.e., rules with a as one of the head atoms. A drawback of this approach is that input atoms might not be correctly recovered, e.g., after simplifying a program.[8] The recent extensions of BINGO look also very interesting from the point of view of the slicing technique proposed in this paper. It is important that the definition of a predicate can be distributed among several program slices or even all of them. This requires a very controlled way of declaring which instances of a predicate are external indeed. As regards further tool development, it may be worthwhile to study alternative ways of splitting ground programs and to implement them as new functionality complementary to MODLIST. We also expect that the idea of using streams of modules and a linker for composing them can lead to completely new architectures for grounders.

Acknowledgments The author wishes to thank Martin Gebser, Roland Kaminski, and Patrik Simons for their comments on extending the internal SMODELS file format.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. C. Baral, G. Brewka, and J. S. Schlipf, editors. *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*. Springer, 2007.
3. P. A. Bonatti, E. Pontelli, and T. C. Son. Credulous resolution for answer set programming. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 418–423. AAAI Press, 2008.
4. M. Gebser, T. Janhunen, M. Ostrowski, T. Schaub, and S. Thiele. A versatile intermediate language for answer set programming. In M. Pagnucco and M. Thielscher, editors, *Proceedings of the 12th International Workshop on Nonmonotonic Reasoning*, pages 150–159, Sydney, Australia, 2008. University of New South Wales, School of Computer Science and Engineering, Technical Report, UNSW-CSE-TR-0819.
5. M. Gebser, R. Kaminski, R. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In *ICLP*, pages 190–205, 2008.
6. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. *Clasp* : A conflict-driven answer set solver. In Baral et al. [2], pages 260–265.

[8] Consider, for instance, the effect of deleting the only rule of $P = \{a \leftarrow a\}$ when a is visible.

7. M. Gebser, T. Schaub, and S. Thiele. Gringo : A new grounder for answer set programming. In Baral et al. [2], pages 266–271.
8. E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.
9. T. Janhunen. Representing normal programs with clauses. In R. López de Mántaras and L. Saitta, editors, *ECAI*, pages 358–362. IOS Press, 2004.
10. T. Janhunen. Intermediate languages of ASP systems and tools. In M. De Vos and T. Schaub, editors, *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming*, number CSBU-2007-05 in Department of Computer Science, University of Bath, Technical Report Series, pages 12–25, Tempe, Arizona, USA, 2007.
11. T. Janhunen, I. Niemelä, and M. Sevalnev. Computing stable models via reductions to difference logic. In E. Erdem, F. Lin, and T. Schaub, editors, *LPNMR*, volume 5753 of *Lecture Notes in Computer Science*, pages 142–154. Springer, 2009.
12. T. Janhunen and E. Oikarinen. Automated verification of weak equivalence within the SMODELS system. *Theory and Practice of Logic Programming*, 7(6):697–744, 2007.
13. V. Lifschitz. Answer set planning (abstract). In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR*, volume 1730 of *Lecture Notes in Computer Science*, pages 373–374. Springer, 1999.
14. Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, pages 23–37, 1994.
15. V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.
16. I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):241–273, 1999.
17. E. Oikarinen and T. Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *Theory and Practice of Logic Programming*, 8(5-6):717–761, 2008.
18. E. Oikarinen and T. Janhunen. A translation-based approach to the verification of modular equivalence. *Journal of Logic and Computation*, 19(4):591–613, 2009.
19. P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.
20. T. Syrjänen. *Logic Programs and Cardinality Constraints: Theory and Practice*. Doctoral dissertation, TKK Dissertations in Information and Computer Science TKK-ICS-D12, Helsinki University of Technology, Faculty of Information and Natural Sciences, Department of Information and Computer Science, Espoo, Finland, 2009.