Engineering an Incremental ASP Solver

M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. Many real-world applications, like planning or model checking, comprise a parameter reflecting the size of a solution. In a propositional formalism like Answer Set Programming (ASP), such problems can only be dealt with in a bounded way, considering one problem instance after another by gradually increasing the bound on the solution size. We thus propose an incremental approach to both grounding and solving in ASP. Our goal is to avoid redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire (extended) problem. We start by furnishing a formal framework capturing our incremental approach in terms of module theory. In turn, we take advantage of this framework for guiding the successive treatment of program slices during grounding and solving. Finally, we describe the first integrated incremental ASP system, *iclingo*, and provide an experimental evaluation.

1 Introduction

Answer Set Programming (ASP; [1]) faces a growing range of applications. This is due to the availability of efficient ASP solvers and ASP's rich modeling language, jointly allowing for an easy yet efficient handling of knowledge-intensive applications. Among them, many real-world applications, like planning or model checking, comprise parameters reflecting solution sizes. However, in the propositional setting of ASP, such problems can only be dealt with in a bounded way by considering in turn one problem instance after another, gradually increasing the bound on the solution size. Such an approach can nonetheless be highly efficient as demonstrated by Satisfiability (SAT) solvers in the aforementioned application areas [2, 3]. However, while SAT has its focus on solving, ASP is also concerned with grounding in view of its modeling language.

We address this by proposing an incremental approach to both grounding and solving in ASP. Our goal is to avoid redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem. To this end, we express a (*parametrized*) *domain description* as a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers. In view of this, we sometimes denote P and Q by $P[k]$ and $Q[k]$. The base program B is meant to describe static knowledge, independent of parameter k. The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k. Our goal is then to decide whether the program

$$R[k/i] = B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i] \tag{1}$$

has an answer set for some (minimum) integer $i \geq 1$. In what follows, we write $R[i]$ rather than $R[k/i]$ whenever clear from the context.

For illustration, consider an action description in $\mathcal{C}+$ [4], involving an action a and a fluent p, along with a query in $\mathcal{Q}_n$ [5] about trajectories of length n. We translate these statements into the following domain description:

$$\left.\begin{array}{l} a \textbf{ causes } p \\ \textbf{exogenous } a \\ \textbf{inertial } p \end{array}\right\} \quad \mapsto \quad \left\{\begin{array}{l} B = \left\{\begin{array}{r} p(0) \leftarrow not\ \neg p(0) \\ \neg p(0) \leftarrow not\ p(0) \\ \leftarrow p(0), \neg p(0) \end{array}\right\} \\ P[k] = \left\{\begin{array}{r} a(k) \leftarrow not\ \neg a(k) \\ \neg a(k) \leftarrow not\ a(k) \\ p(k) \leftarrow a(k) \\ p(k) \leftarrow p(k{-}1), not\ \neg p(k) \\ \neg p(k) \leftarrow \neg p(k{-}1), not\ p(k) \\ \leftarrow p(k), \neg p(k) \\ \leftarrow a(k), \neg a(k) \end{array}\right\} \end{array}\right. \tag{2}$$

$$\left.\begin{array}{l} \neg p \textbf{ holds at } 0 \\ p \textbf{ holds at } n \\ \neg a \textbf{ occurs at } n \end{array}\right\} \quad \mapsto \quad \left\{ Q[k] = \left\{\begin{array}{l} \leftarrow not\ \neg p(0) \\ \leftarrow not\ p(k) \\ \leftarrow not\ \neg a(k) \end{array}\right\} \right. .$$

This domain description induces no answer sets for $R[1]$, but we obtain a single one for $R[2]$, that is, $AS(R[2]) = \{\{\neg p(0), a(1), p(1), \neg a(2), p(2)\}\}$.

Such an answer is usually found by appeal to iterative deepening search. That is, one first checks whether $R[1]$ has an answer set, if not, the same is done for $R[2]$, and so on. For a given i, this approach re-processes B for i times and $(i{-}j{+}1)$ times each $P[j]$, where $1 \leq j \leq i$, while each $Q[j]$ is dealt with only once. Unlike this, we propose to compute answers sets of (1) in an incremental fashion, starting from $R[1]$ but then gradually dealing with the program slices $P[i]$ and $Q[i]$ rather than the entire program $R[i]$ in (1). However, B and the previously processed slices $P[j]$ and $Q[j]$, $1 \leq j < i$, must be taken into account when dealing with $P[i]$ and $Q[i]$: while the rules in $P[j]$ are accumulated, the ones in $Q[j]$ must be discarded. For accomplishing this, an ASP system has to operate in a "stateful way." That is, it has to maintain its previous state for processing the current program slices. In this way, all components, B, $P[j]$, and $Q[i]$, of (1) are dealt with only once, and duplicated work is avoided when increasing i.

Given that an ASP system is composed of a grounder and a solver, our incremental approach has the following specific advantages over the standard approach. As regards grounding, it reduces efforts by avoiding reproducing previous ground rules. Regarding solving, it reduces redundancy, in particular, if a learning ASP solver is used, given that previously gathered information on heuristics, conflicts, or loops (cf. [6]), respectively, remains available and can thus be continuously exploited. We provide some empirical evidence using the new incremental ASP system *iclingo* [7].

2 Background

Our language is built from a set $\mathcal{F}$ of *function* symbols (including the natural numbers), a set $\mathcal{V}$ of *variable* symbols, and a set $\mathcal{P}$ of *predicate* symbols. In view of our goal, $\mathcal{V}$ contains a distinguished parameter symbol k (varying over natural numbers). The set $\mathcal{T}$

of *terms* is the smallest set containing $\mathcal{V}$ and all expressions of the form $f(t_1, \ldots, t_n)$, where $f \in \mathcal{F}$ and $t_i \in \mathcal{T}$ for $1 \leq i \leq n$. The set $\mathcal{A}$ of *atoms* contains all expressions of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}$ and $t_i \in \mathcal{T}$ for $1 \leq i \leq n$. A *literal* is an atom a or its (default) negation $not\ a$. Given a set L of literals, let $L^+ = \{a \in \mathcal{A} \mid a \in L\}$ and $L^- = \{a \in \mathcal{A} \mid not\ a \in L\}$. A *logic program* over $\mathcal{A}$ is a set of *rules* of the form $a \leftarrow b_1, \ldots, b_m, not\ c_{m+1}, \ldots, not\ c_n$, where $a, b_i, c_j \in \mathcal{A}$ for $0 < i \leq m < j \leq n$. The semantics of integrity constraints and choice rules is given through program transformations. For instance, $\{a\} \leftarrow$ is a shorthand for $a \leftarrow not\ a', a' \leftarrow not\ a$ and similarly $\leftarrow a$ for $a' \leftarrow a, not\ a'$, for a new atom a'. For a rule r, let $head(r) = a$ be the *head* of r, $body(r) = \{b_1, \ldots, b_m, not\ c_{m+1}, \ldots, not\ c_n\}$ be the *body* of r, and finally $atom(r) = \{head(r)\} \cup body(r)^+ \cup body(r)^-$. For a program P, define $head(P) = \{head(r) \mid r \in P\}$ and $atom(P) = \bigcup_{r \in P} atom(r)$. Given an expression $e \in \mathcal{T} \cup \mathcal{A}$, let $var(e)$ denote the set of all variables occurring in e; analogously, $var(r)$ gives all variables in rule r. Expression $e \in \mathcal{T} \cup \mathcal{A}$ is *ground*, if $var(e) = \emptyset$. The *ground instantiation* of a program P is defined as $grd(P) = \{r\theta \mid r \in P, \theta : var(r) \to \mathcal{U}\}$, where $\mathcal{U} = \{t \in \mathcal{T} \mid var(t) = \emptyset\}$; analogously, $grd(\mathcal{A}) = \{a \in \mathcal{A} \mid var(a) = \emptyset\}$.

A set $X \subseteq grd(\mathcal{A})$ is an *answer set* of a program P over $\mathcal{A}$, if X is the $\subseteq$-smallest model of $\{head(r) \leftarrow body(r)^+ \mid r \in grd(P), body(r)^- \cap X = \emptyset\}$. The set of answer sets of a program P is denoted $AS(P)$. Two programs, P and P', are *equivalent*, denoted by $P \equiv P'$, if $AS(P) = AS(P')$.

3 Semantic Underpinnings through Incremental Modularity

For providing a clear interface between program slices and guaranteeing their compositionality, we build upon the concept of a module developed in [8]: a *module* $\mathbb{P}$ is a triple (P, I, O) consisting of a (ground) program P over $grd(\mathcal{A})$ and sets $I, O \subseteq grd(\mathcal{A})$ such that $I \cap O = \emptyset$, $atom(P) \subseteq I \cup O$, and $head(P) \subseteq O$. The elements of I and O are called *input* and *output* atoms, also denoted by $I(\mathbb{P})$ and $O(\mathbb{P})$, respectively; similarly, we refer to P by $P(\mathbb{P})$. We say that $\mathbb{P}$ is *input-free*, if $I(\mathbb{P}) = \emptyset$.

For giving an incremental account of modularity, we begin with associating a (non-ground) program P and a set I of (ground) input atoms with a module, denoted by $\mathbb{P}(I)$, imposing certain restrictions on the ground program induced by P. To this end, we define for a program P over $grd(\mathcal{A})$ and a set $X \subseteq grd(\mathcal{A})$, the set $P|_X$ of rules as

$$\{head(r) \leftarrow body(r)^+ \cup L \mid r \in P, body(r)^+ \subseteq X, L = \{not\ c \mid c \in body(r)^- \cap X\}\}\ .$$

Note that $P|_X$ projects the bodies of rules in P to the atoms of X. If a body contains an atom outside X, either the corresponding rule or literal is removed, depending on whether the atom occurs positively or negatively. This allows us to associate (non-ground) programs with (ground) modules in the following way.

Definition 1. *Let P be a program over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$. We define $\mathbb{P}(I)$ as the module $(\ grd(P)|_Y, I, head(grd(P)|_X)\ )$, where $X = I \cup head(grd(P))$ and $Y = I \cup head(grd(P)|_X)$.*

The full ground instantiation $grd(P)$ of P is projected onto inputs and atoms defined in $grd(P)$. The head atoms of this projection, viz., $head(grd(P)|_{I \cup head(grd(P))})$, serve as output atoms and are used to simplify $grd(P)$, sparing only input and output atoms.

As a simple example, consider $P[k] = \{p(k) \leftarrow p(Y), not\ p(2); p(k) \leftarrow p(2)\}$. Note that $grd(P[1])$ is infinite. However, for $X = \{p(0), p(1)\}$, we get

$$grd(P[1])|_X = \{p(1) \leftarrow p(0);\ p(1) \leftarrow p(1)\} \text{ and } head(grd(P[1])|_X) = \{p(1)\}\ .$$

For $I = \{p(0)\}$, we obtain $I \cup head(grd(P[1])) = I \cup head(grd(P[1])|_X) = \{p(0)\} \cup \{p(1)\} = X$. Thus, $\mathbb{P}[1](\{p(0)\}) = (\ grd(P[1])|_{\{p(0),p(1)\}}\ ,\ \{p(0)\}\ ,\ \{p(1)\}\ )$, and $P(\mathbb{P}[1](I)) = grd(P[1])|_X$ is finite. Note that, if $p(1)$ had been in I, we would not have obtained a module since $P[1]$ defines $p(1)$. Hence, it must be an output atom.

Proposition 1. *Let P be a program over $\mathcal{A}$, $I \subseteq grd(\mathcal{A})$, and $\mathbb{P}(I) = (P', I, O)$. Then, we have $O \subseteq grd(\mathcal{A})$ and $atom(P') \subseteq I \cup O$.*

We define the *join* of two modules $\mathbb{P}$ and $\mathbb{Q}$, denoted by $\mathbb{P} \sqcup \mathbb{Q}$, as the module

$$(\ P(\mathbb{P}) \cup P(\mathbb{Q})\ ,\ I(\mathbb{P}) \cup (I(\mathbb{Q}) \setminus O(\mathbb{P}))\ ,\ O(\mathbb{P}) \cup O(\mathbb{Q})\ )\ ,$$

provided that $(I(\mathbb{P}) \cup O(\mathbb{P})) \cap O(\mathbb{Q}) = \emptyset$. This definition is simpler than the original one in [8], but also more restrictive. For instance, our definition does not permit (negative) recursion between two modules to be joined, similar to splitting [9]. (Note that positive and negative recursion are allowed within each module.) Also note that the join of $\mathbb{P}$ and $\mathbb{Q}$, as defined above, is not commutative: even if $\mathbb{P} \sqcup \mathbb{Q}$ is defined, $\mathbb{Q} \sqcup \mathbb{P}$ might be undefined. However, lacking commutativity is not an issue in our incremental context, where portions of a domain description are always processed in order.

We make use of the join to formalize the compositionality of modules induced by domain descriptions.

Definition 2. *A domain description $(B, P[k], Q[k])$ is modular, if the modules*

$$\mathbb{P}_i\ =\ \mathbb{P}_{i-1} \sqcup \mathbb{P}[i](O(\mathbb{P}_{i-1})) \qquad \textit{and} \qquad \mathbb{Q}_i\ =\ \mathbb{P}_i \sqcup \mathbb{Q}[i](O(\mathbb{P}_i))$$

are defined for $i \geq 1$, where $\mathbb{P}_0 = \mathbb{B}(\emptyset)$.

The requirement of the join being defined demands that gradually obtained ground programs must define distinct atoms. Also, the directedness of the join, in a sense, permits an information flow between ground programs in increasing order of values substituted for k, but not the other way round.

As an example, consider $(B, P[k], Q[k])$ over $\mathcal{A}$, where:

$$\begin{aligned} B &= \{\ dbl(0,0) \leftarrow\ \} \\ P[k] &= \{\ n(k) \leftarrow ; \qquad dbl(k, 2{*}Y) \leftarrow n(Y), not\ n(Y{+}1)\ \} \\ Q[k] &= \{\ \leftarrow dbl(Y, k{-}1)\ \}\ . \end{aligned} \tag{3}$$

This domain description induces the following modules:[1]

$$\begin{aligned} \mathbb{P}_0\ &=\ (\ B = \{dbl(0,0) \leftarrow\}\ ,\ \emptyset\ ,\ \{dbl(0,0)\}\ )\ , \\ \mathbb{P}_1\ &=\ (\ B \cup \{n(1) \leftarrow;\ dbl(1,2) \leftarrow n(1)\}\ ,\ \emptyset\ ,\ O(\mathbb{P}_0) \cup \{n(1), dbl(1,2)\}\ )\ , \\ \mathbb{Q}_1\ &=\ (\ P(\mathbb{P}_1) \cup \{\leftarrow dbl(0,0)\}\ ,\ \emptyset\ ,\ O(\mathbb{P}_1)\ )\ , \\ \mathbb{P}_2\ &=\ (\ P(\mathbb{P}_2)\ ,\ \emptyset\ ,\ O(\mathbb{P}_1) \cup \{n(2), dbl(2,2), dbl(2,4)\}\ ) \\ &\qquad \text{where } P(\mathbb{P}_2) = P(\mathbb{P}_1) \cup \{n(2) \leftarrow\} \cup \left\{ \begin{matrix} dbl(2,2) \leftarrow n(1), not\ n(2) \\ dbl(2,4) \leftarrow n(2) \end{matrix} \right\}\ , \\ \mathbb{Q}_2\ &=\ (\ P(\mathbb{P}_2)\ ,\ \emptyset\ ,\ O(\mathbb{P}_2)\ )\ , \end{aligned}$$

[1] For simplicity, we evaluate arithmetic expressions.

$$\mathbb{P}_3 \;=\; (\; P(\mathbb{P}_3)\,,\; \emptyset\,,\; O(\mathbb{P}_2) \cup \{n(3), dbl(3,2), dbl(3,4), dbl(3,6)\}\;)$$
$$\text{where } P(\mathbb{P}_3) = P(\mathbb{P}_2) \cup \{n(3) \leftarrow\} \cup \left\{ \begin{array}{l} dbl(3,2) \leftarrow n(1), not\ n(2) \\ dbl(3,4) \leftarrow n(2), not\ n(3) \\ dbl(3,6) \leftarrow n(3) \end{array} \right\},$$
$$\mathbb{Q}_3 \;=\; (\; P(\mathbb{P}_3) \cup \{\leftarrow dbl(1,2); \leftarrow dbl(2,2); \leftarrow dbl(3,2)\}\,,\; \emptyset\,,\; O(\mathbb{P}_3)\;)\text{, etc.}$$

All above modules are defined (in terms of the join) and input-free. Since this also applies to $\mathbb{P}_i$ and $\mathbb{Q}_i$ for every $i > 3$, we have that domain description (3) is modular. Hence, we can read off the results of the expressed queries from the answer sets of each $P(\mathbb{Q}_i)$. If $i \geq 1$ is odd, we get $AS(P(\mathbb{Q}_i)) = \emptyset$. Otherwise, if $i \geq 1$ is even, then $AS(P(\mathbb{Q}_i)) = \{\{dbl(0,0)\} \cup \{n(j), dbl(j, 2{*}j) \mid 1 \leq j \leq i\}\}$. In fact, for $1 \leq j \leq i$ and $Y = j$, literals $not\ n(Y{+}1)$ are removed from the body of the second rule in $P[k]$ during the incremental construction because the underlying atoms $n(j{+}1)$ are undefined in $P[j]$. In this way, the atoms $dbl(j, 2{*}j)$ are derived. Note that this is not possible for $j < i$ with program $\bigcup_{1 \leq j \leq i} P[j]$ in a non-incremental setting.

Proposition 2. *Let $(B, P[k], Q[k])$ be a modular domain description, and let $(\mathbb{P}_i)_{i \geq 0}$ and $(\mathbb{Q}_i)_{i \geq 1}$ as in Definition 2. Then, we have the following for $i \geq 1$:*

1. *$\mathbb{P}_i$ and $\mathbb{Q}_i$ are input-free;*
2. *$atom(P(\mathbb{P}_i)) \subseteq O(\mathbb{P}_i)$ and $atom(P(\mathbb{Q}_i)) \subseteq O(\mathbb{Q}_i)$;*
3. *$P(\mathbb{P}_i) = P(\mathbb{B}(\emptyset)) \cup \bigcup_{1 \leq j \leq i} P(\mathbb{P}[j](O(\mathbb{P}_{j-1})))$ and $P(\mathbb{Q}_i) = P(\mathbb{P}_i) \cup P(\mathbb{Q}[i](O(\mathbb{P}_i)))$;*
4. *$head(P(\mathbb{P}[i](O(\mathbb{P}_{i-1})))) \cap atom(P(\mathbb{P}_{i-1})) = \emptyset$ and $head(P(\mathbb{Q}[i](O(\mathbb{P}_i)))) \cap atom(P(\mathbb{P}_i)) = \emptyset$.*

The third item essentially states that the combined programs obtained for $i \geq 1$ equal the union of subprograms added for each $1 \leq j \leq i$. Importantly, the fourth item expresses that the head atoms of a newly added subprogram are different from all atoms encountered before. Hence, the sequence $(O(\mathbb{P}_i))_{i \geq 0}$ of output atoms amounts to a splitting sequence [9] for $\bigcup_{i \geq 0} P(\mathbb{P}_i)$. Nonetheless, we intentionally use modules and joins rather than splitting for formalizing our approach, as the composition of (ground) programs done in incremental steps is only indirectly addressed by splitting sequences.

Note that we only take advantage of module theory for establishing a well-defined formal setting for incremental ASP solving. Our computational approach deals directly with programs in order to exploit existing ASP technology. In view of this, the next result shows when the module-guarded formation of ground programs coincides with separate grounding. To this end, we define a domain description $(B, P[k], Q[k])$ as *bound*, if $atom(grd(B)) \subseteq head(grd(B))$ and $atom(grd(P[i])) \subseteq head(grd(B \cup \bigcup_{1 \leq j \leq i} P[j]))$ for all $i \geq 1$. With this concept at hand, we have the following result.

Theorem 1. *Let $(B, P[k], Q[k])$ be a bound modular domain description, and let $(\mathbb{P}_i)_{i \geq 0}$ and $(\mathbb{Q}_i)_{i \geq 1}$ as in Definition 2. Then, we have the following for $i \geq 1$:*

1. *$P(\mathbb{P}_i) \equiv grd(B \cup \bigcup_{1 \leq j \leq i} P[j])$;*
2. *$P(\mathbb{Q}_i) \equiv grd(B \cup \bigcup_{1 \leq j \leq i} P[j] \cup Q[i])$.*

That is, for bound modular domain descriptions, the same result is obtained when grounding is done either stepwise or in a single pass. Note that the domain description given in (2) is modular and bound. Likewise, the domain description in (3) is modular, but it is not bound because of $n(Y)$ and $n(Y{+}1)$ occurring in body literals of $P[k]$.

4 Incremental ASP Solving

The computation of answer sets consists of two phases: a *grounding* phase aiming at a compact ground instantiation of the original program and a *solving* phase computing the answer sets of the obtained ground program. As motivated in Section 1, our incremental approach is based on the idea that the grounder as well as the solver are implemented in a stateful way. Thus, both keep their previous states when increasing parameter k in (1). As regards grounding, at each step i, the goal is to produce only ground rules stemming from program slices $P[i]$ and $Q[i]$, without re-producing previous ground rules. The ground program slices are then gradually passed to the solver that accumulates all ground rules from $P[j]$, for $1 \leq j \leq i$, while discarding the rules from $Q[j]$, if $j < i$.

Grounding. Let us now characterize the consecutive program slices in terms of grounding programs. In practice, given a program P, the goal of a grounder is to produce a finite and compact yet equivalent representation of $grd(P)$ by applying answer set preserving simplifications (cf. [10, 11]). In our context, $P[i]$ and $Q[i]$ are not grounded in isolation for $i \geq 1$. Rather, the ground programs obtained from previous program slices are augmented with newly derived ground rules. We thus assume a grounder to be stateful, where states are represented by the head atoms of ground rules belonging to the output of previous grounding steps.

Given a program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$, we define an (*incremental*) *grounder* as a partial function $\texttt{ground} : (P, I) \mapsto (P', O)$, where P' is a program over $grd(\mathcal{A})$ and $O \subseteq grd(\mathcal{A})$. Thereby, P' stands for the ground program obtained from P, where the input atoms I provide domain information used to instantiate non-ground atoms in the rules of P. The output atoms in O essentially correspond to $head(P')$. Their main use is to carry state information, as O can serve as input to subsequent grounding steps. Also note that `ground` is not required to be total, given that existing grounders, like *lparse* [12] and *gringo* [7], impose certain restrictions on non-ground programs, such as being ω- or λ-restricted, not necessarily met by P.

Next, we formalize a grounder's *adequacy* to an incremental setting.

Definition 3. *A grounder* `ground` *is adequate, if for every program P over $\mathcal{A}$ and $I \subseteq grd(\mathcal{A})$ such that* $\texttt{ground}(P, I) = (P', O)$ *is defined, the following holds*:

1. $\big(P \cup \{\{a\} \leftarrow \mid a \in I\}\big) \equiv \big(P' \cup \{\{a\} \leftarrow \mid a \in I\}\big)$,
2. $\bigcup_{X \in AS(P \cup \{\{a\} \leftarrow \mid a \in I\})}(X \setminus I) \subseteq O \subseteq head(grd(P)|_Y)$, *where* $Y = I \cup head(grd(P))$, *and*
3. *for every* $r' \in P'$, *there is some* $r \in grd(P)$ *such that* $head(r) = head(r')$ *and* $body(r)^+ \setminus (I \cup O) \subseteq body(r')^+$.

The first condition expresses that P and P', each augmented with any combination of input atoms in I, must be equivalent. The second condition stipulates that all non-input atoms belonging to some answer set X of $P \cup \{\{a\} \leftarrow \mid a \in I\}$ are contained in O. In addition, O must not exceed the head atoms of $grd(P)|_{I \cup head(grd(P))}$ in order to suitably restrict subsequently produced ground rules, using O as an input (cf. Definition 4). Finally, the third condition forbids the introduction of rules that cannot be obtained from $grd(P)$ via permissible simplifications. Clearly, an adequate grounder may apply answer-set preserving simplifications to compact its output.

For illustration, consider $P[k]$ in (3) along with $I = \{n(1)\}$. An adequate grounder could, for instance, map $(P[2], I)$ to $(P', O = \{n(2), dbl(2,2), dbl(2,4)\})$, where

$$P' = \{n(2) \leftarrow;\ dbl(2,2) \leftarrow n(1), not\ n(2);\ dbl(2,4) \leftarrow n(2), not\ n(3)\}\ . \qquad (4)$$

Note that $AS(P' \cup \{\{n(1)\} \leftarrow\}) = \{\{n(1), n(2), dbl(2,4)\}, \{n(2), dbl(2,4)\}\} = AS(P[2] \cup \{\{n(1)\} \leftarrow\})$. Due to fact $n(2) \leftarrow$, the second rule could also be dropped from P'; similarly, $dbl(2,2)$ could be removed from O. Furthermore, literals $n(2)$ and $not\ n(3)$ could be dropped from the last rule, still satisfying Definition 3. Note that it is crucial to restrict the atoms in O to $head(P')$. For instance, this forbids the inclusion of $n(3)$ in O, permitting further simplifications of P' wrt O.

The following definition specifies the (ground) program slices gradually obtained from a domain description using a (stateful) grounder.

Definition 4. *Let $(B, P[k], Q[k])$ be a domain description, and let* `ground` *be a grounder. We define for $i \geq 1$:*

$$\begin{array}{ll} (P_0, O_0) = (P_0'|_{O_0}, O_0)\ , & \textit{where } (P_0', O_0) = \texttt{ground}(B, \emptyset)\ , \\ (P_i, O_i) = (P_i'|_{(\bigcup_{0 \leq j \leq i} O_j)}, O_i)\ , & \textit{where } (P_i', O_i) = \texttt{ground}(P[i], \bigcup_{0 \leq j < i} O_j)\ , \\ (Q_i, O_i') = \texttt{ground}(Q[i], \bigcup_{0 \leq j \leq i} O_j)\ . & \end{array}$$

Note that the successively identified output atoms in O_j, for $0 \leq j \leq i$, are used to simplify ground programs P_i' by eliminating either rules or negative body literals. We thus obtain ground program slices P_i such that $\bigcup_{r \in P_i} (body(r)^+ \cup body(r)^-) \subseteq \bigcup_{0 \leq j \leq i} O_j$. This reduction is important in view of the compositional semantics of domain descriptions in Definition 2. For instance, if not done by `ground` itself, literal $not\ n(3)$ must a posteriori be removed from the body of the third rule in (4), in order to obtain the intended ground program slice. However, ground programs Q_i need not be reduced, since their rules are neither accumulated nor reused.

The next result links the semantics of modular domain descriptions to that of ground programs gradually produced by an adequate grounder.

Theorem 2. *Let $(B, P[k], Q[k])$ be a modular domain description and* `ground` *an adequate grounder. Let $(\mathbb{P}_i)_{i \geq 0}$ and $(\mathbb{Q}_i)_{i \geq 1}$ be as in Definition 2 and $(P_i, O_i)_{i \geq 0}$ and $(Q_i, O_i')_{i \geq 1}$ as in Definition 4. If (P_j, O_j) is defined for $0 \leq j \leq i$, we have for $i \geq 1$:*

1. $P(\mathbb{P}_0) \equiv P_0$;
2. $P(\mathbb{P}_i) \equiv \bigcup_{0 \leq j \leq i} P_j$;
3. $P(\mathbb{Q}_i) \equiv \bigcup_{0 \leq j \leq i} P_j \cup Q_i$, *provided that (Q_i, O_i') is defined.*

Recall that `ground` can be partial. In fact, existing grounders impose certain restrictions on the non-ground programs of a domain description, such as being ω- or λ-restricted, guaranteeing the finiteness of equivalent ground programs. Assuming that such requirements are met, we next detail how grounding output can be processed by an answer set solver.

Solving. As with grounding, special care must be taken for customizing existing ASP solving technology in an incremental setting. First, we have to guarantee the compositionality of successive program slices. Second, a solver has to respect the cumulative and volatile roles of P_j and Q_i, respectively. And finally, we have to furnish a clear interface between the grounding and the solving component.

For capturing compositionality, we rely on [13], characterizing the answer sets of a program P over $grd(\mathcal{A})$ by the (classical) models of its completion and loop formulas. For $Y \subseteq grd(\mathcal{A})$, define the *completion* of P, $CF(P,Y)$, as the set of formulas

$$a \leftrightarrow \bigvee_{r \in P, head(r)=a} (\bigwedge_{b \in body(r)^+} b \wedge \bigwedge_{c \in body(r)^-} \neg c) \ ,$$

for all $a \in Y$. Moreover, $Y \subseteq grd(\mathcal{A})$ is a *loop* of P, if $(Y, E = \{(head(r), b) \mid r \in P, head(r) \in Y, b \in body(r)^+ \cap Y\})$ is a strongly connected graph such that $E \neq \emptyset$. Then, the set of *loop formulas* for P, $LF(P)$, is given by the set of formulas

$$\bigvee_{a \in Y} a \rightarrow \bigvee_{r \in P, head(r) \in Y, body(r)^+ \cap Y = \emptyset} (\bigwedge_{b \in body(r)^+} b \wedge \bigwedge_{c \in body(r)^-} \neg c) \ ,$$

for all loops Y of P. As shown in [13], a set $X \subseteq grd(\mathcal{A})$ is an answer set of P iff $X \models CF(P, grd(\mathcal{A})) \cup LF(P)$.

For programs induced by modular domain descriptions, completion and loop formulas can be sliced as follows.

Theorem 3. *Let $(B, P[k], Q[k])$ be a modular domain description, let* `ground` *be an adequate grounder, and let $(P_i, O_i)_{i \geq 0}$ and $(Q_i, O'_i)_{i \geq 1}$ as in Definition 4. If (P_j, O_j) is defined for $0 \leq j \leq i$ and if (Q_i, O'_i) is defined, we have the following for $i \geq 1$:*[2]

$$CF(\bigcup_{0 \leq j \leq i} P_j \cup Q_i, grd(\mathcal{A})) \equiv \bigcup_{0 \leq j \leq i} CF(P_j, O_j) \cup CF(Q_i, grd(\mathcal{A}) \setminus \bigcup_{0 \leq j \leq i} O_j)$$
$$LF(\bigcup_{0 \leq j \leq i} P_j \cup Q_i) \equiv \bigcup_{0 \leq j \leq i} LF(P_j) \cup LF(Q_i|_{head(\bigcup_{0 \leq j \leq i} P_j \cup Q_i)}) \ .$$

Recall that modular domain descriptions $(B, P[k], Q[k])$ induce splitting sequences [9]. This means that the answer sets of $\bigcup_{0 \leq j \leq i} P_j \cup Q_i$ can be decomposed into a sequence of answer sets for subprograms $P_0, \ldots, P_i, Q_i$. Theorem 3 reflects this decomposition in terms of completion and loop formulas, which are material to the data structures of ASP solvers. Thus, the practical consequence of the decomposability of completion and loop formulas is that a solver can successively build its data structures in a modular fashion. If this was not the case, it would be rather misleading to qualify an approach as incremental. Hence, a modularity condition is essential for incremental computations.

When processing consecutive program slices, we have to distinguish cumulative and volatile ones. That is, while the ground rules in P_j are accumulated within the solver for $0 \leq j \leq i$, the ones in Q_j must be discarded for $1 \leq j < i$ when Q_i is added. We accomplish this by adding to each rule in Q_j a new body atom α_j, along with rules achieving that α_j holds only at step j. To this end, we define the following set of rules for a program Q over $grd(\mathcal{A})$ and a new atom $\alpha \notin grd(\mathcal{A})$:

$$Q(\alpha) = \{head(r) \leftarrow body(r) \cup \{\alpha\} \mid r \in Q\} \ .$$

In our incremental setting, the addition of new atoms allows us to selectively (de)activate volatile program slices.

Proposition 3. *Let $(P_i)_{i \geq 0}$ and $(Q_i)_{i \geq 1}$ be sequences of programs over $grd(\mathcal{A})$, and let $F_j = \{\alpha_j \leftarrow\}$ for $\alpha_j \notin grd(\mathcal{A})$ and $j \geq 1$. Then, we have the following for $i \geq 1$:*

$$\bigcup_{0 \leq j \leq i} P_j \cup Q_i \cup F_i \ \equiv \ P_0 \cup \bigcup_{1 \leq j \leq i} (P_j \cup Q_j(\alpha_j)) \cup F_i \ .$$

[2] We abuse notation and let $\equiv$ stand for classical equivalence here.

The addition of F_i on the left hand side is merely for establishing formal equivalence, considering that α_i occurs in $Q_i(\alpha_i)$ but not in Q_i. The fact that programs $Q_j(\alpha_j)$ behave neutrally, as long as α_j is underivable, provides us with a handle to control the effective program slices. In addition to activating some $Q_j(\alpha_j)$ for $j \geq 1$, we also have to deactivate it in subsequent steps. Thus, a solver cannot include α_j persistently as a fact. But rather than explicitly deleting any fact (or rule) previously passed to the solver, we build upon an interface supporting *assumptions*. This trims the required solver interface to only two functions:

- `add`(P) incorporates a ground logic program P into the rule database of the solver;
- `solve`(L) takes a set L of ground literals and computes the answer sets X of the ground program comprised in the solver that satisfy $L^+ \subseteq X$ and $L^- \cap X = \emptyset$.

This simple interface is similar to the one for incremental SAT solving given in [14]. The literals L passed to `solve` constitute assumptions, which can semantically be viewed as the set of integrity constraints $\{\leftarrow not\ a \mid a \in L^+\} \cup \{\leftarrow a \mid a \in L^-\}$. However, as regards *clasp* [6], the crucial difference between integrity constraints and assumptions is that the former give rise to program simplifications affecting internal data structures, while the effect of the latter is temporary, i.e., restricted to an invocation of `solve`. While former assumptions can easily be withdrawn, for a learning solver, it would be much harder to support an explicit deletion of obsolete problem parts [14].

Let us now situate the *solver* in our incremental context.

Definition 5. *Let $(R_i)_{i\geq 0}$ and $(L_i)_{i\geq 0}$ be sequences of programs and literals over $grd(\mathcal{A}) \cup \{\alpha_i \mid i \geq 0\}$. A solver is a pair of total functions* `add` $: R_i \mapsto S_i$ *and* `solve` $: L_i \mapsto \chi$, *where $S_0 = R_0|_{head(R_0)}$, $S_i = S_{i-1} \cup R_i|_{head(S_{i-1} \cup R_i)}$ for $i \geq 1$, and $\chi \subseteq 2^{(grd(\mathcal{A}) \cup \{\alpha_i | i \geq 0\})}$.*

Note that only `add` affects a solver's state, where added programs are subject to simplification. In fact, as with P_i for $i \geq 0$ in Definition 4, we assume that atoms not occurring as the head of any rule are eliminated. Even if such an atom becomes derivable later on when another program is added, it can thus not interact with the rules already present. The reason for this design decision is that, although operating in an open environment, the possible addition of information or program slices, respectively, should not force the solver to continuously rebuild its existent data structures. Of course, this necessitates program slices to be provided in a bottom-up manner. The second function, `solve`, leaves the accumulated program slices (logically) unaffected, that is, the passed literals are only assumed locally within `solve`.

The objective of maintaining program slices, once they have been added, also motivates the following definition of *soundness*.

Definition 6. *A solver as in Definition 5 is sound, if for all sequences $(R_i)_{i\geq 0}$ and $(L_i)_{i\geq 0}$ of programs and literals over $grd(\mathcal{A}) \cup \{\alpha_i \mid i \geq 0\}$, and for every $i \geq 0$, we have: $X \in$* `solve`(L_i) *iff $L_i^+ \subseteq X \subseteq head(S_i) \setminus L_i^-$ such that $X \models \bigcup_{0\leq j\leq i}(CF(R_j|_{head(S_j)}, head(R_j|_{Y_j})) \cup LF(R_j|_{head(S_j)}))$, where $Y_0 = head(R_0)$ and $Y_j = head(S_{j-1} \cup R_j)$ for $1 \leq j \leq i$.*

First, observe that literals passed as assumptions in L_i must be respected by solutions X returned by a sound solver. Second, X must satisfy the completion and loop formulas

individually for each program slice, thereby, restricting the attention to the respective head atoms. This conception allows the solver to build its data structures in a modular way, without sacrificing soundness, but it also relocates the responsibility to properly partition a program away from the solver. However, as Theorem 3 shows, modular domain descriptions (along with an adequate grounder) permit the construction of a program's completion and loop formulas locally for program slices, obtaining the same answer sets as with the entire program.

We now define the program slices to be added to the solver for the ground rules obtained from a domain description.

Definition 7. *Let $(B, P[k], Q[k])$ be a domain description, let* `ground` *be a grounder, and let $(P_i, O_i)_{i\geq 0}$ and $(Q_i, O'_i)_{i\geq 1}$ as in Definition 4. If (P_0, O_0), (P_j, O_j), and (Q_j, O'_j) are defined for $1 \leq j \leq i$, we define a sequence $(R_i)_{i\geq 0}$ of programs and a sequence $(L_i)_{i\geq 0}$ of literals for $1 \leq j \leq i$ and $\alpha_{j-1}, \alpha_j \notin grd(\mathcal{A})$ by*:

$$\begin{array}{ll} R_0 = P_0 & R_j = P_j \cup Q_j(\alpha_j) \cup \{\{\alpha_j\} \leftarrow\} \cup \{\leftarrow \alpha_{j-1}\} \\ L_0 = \emptyset & L_j = \{\alpha_j\}\ . \end{array}$$

The difference between the cumulative rules in P_j and the volatile ones in Q_j is that an additional atom α_j is appended to the bodies of the latter. Moreover, choice rule $\{\alpha_j\} \leftarrow$ nominally permits the unconditional inclusion of α_j in an answer set. However, upon the invocation of `solve` in step j, literal α_j is passed as assumption, so that answer sets must necessarily contain α_j. In contrast, in step $j + 1$, integrity constraint $\leftarrow \alpha_j$ is persistently added to the solver, forcing α_j to be false. Due to this, all rules in Q_j are deactivated in later steps. Notably, *clasp* eliminates such false atoms and rules with false bodies from its data structures, thus deleting a whole obsolete program Q_j.

In theory, no added rule is deleted later on. Thus, we require an additional condition.

Definition 8. *We define a domain description $(B, P[k], Q[k])$ as separated, if for all $i \geq 1$ and $j > i$, $head(grd(Q[i])) \cap head(grd(P[j] \cup Q[j])) = \emptyset$.*

Separation can easily be achieved by using distinct predicates and parameter k in the heads of rules in $Q[k]$ as well as in body atoms corresponding to such heads. The domain descriptions given in (2) and (3), trivially, are separated.

Using an adequate grounder and a sound solver, we finally establish that our incremental solving strategy leads to the desired outcomes for modular domain descriptions.

Theorem 4. *Let $(B, P[k], Q[k])$ be a separated modular domain description, let* `ground` *be an adequate grounder, and let $(P_i, O_i)_{i\geq 0}$ and $(Q_i, O'_i)_{i\geq 1}$ as in Definition 4. Furthermore, let* $(\texttt{add}, \texttt{solve})$ *be a sound solver, $(R_i)_{i\geq 0}$ and $(L_i)_{i\geq 0}$ as in Definition 7, and $S_j = \texttt{add}(R_j)$ for $j \geq 0$ as in Definition 5. If (P_0, O_0), (P_j, O_j), and (Q_j, O'_j) are defined for $1 \leq j \leq i$, we have the following for $i \geq 1$:*

$X \in \texttt{solve}(L_i)$ iff $(X \setminus \{\alpha_i\}) \in AS(\bigcup_{0\leq j\leq i} P_j \cup Q_i)$.

Comparing with the third item in Theorem 2 shows that our approach, comprising incremental grounding and solving, matches exactly the semantics of (programs induced by) separated modular domain descriptions. In this context, the modularity condition in Definition 2 allows us to largely reuse existing ASP technology, as we see below.

Algorithm 1: `isolve`

Input : A domain description $(B, P[k], Q[k])$.
Output : A nonempty set of answer sets.
Internal: A grounder GROUNDER.
Internal: A solver SOLVER.

```
1  i ← 0
2  (P_0, O) ← GROUNDER.ground(B, ∅)
3  SOLVER.add(P_0)
4  loop
5      i ← i + 1
6      (P_i, O_i) ← GROUNDER.ground(P[i], O)
7      SOLVER.add(P_i)
8      O ← O ∪ O_i
9      (Q_i, O'_i) ← GROUNDER.ground(Q[i], O)
10     SOLVER.add(Q_i(α_i) ∪ {{α_i} ←} ∪ {← α_{i−1}})
11     χ ← SOLVER.solve({α_i})
12     if χ ≠ ∅ then return {X \ {α_i} | X ∈ χ}
```

Algorithm 1 combines our grounding and solving functions for successively computing the answer sets of programs induced by a domain description $(B, P[k], Q[k])$. To this end, `isolve` makes use of one instance of a grounder, denoted by GROUNDER, and one instance of a solver, viz., SOLVER. Programs B, $P[i]$, and $Q[i]$ are then gradually grounded by means of GROUNDER. Provided that GROUNDER can instantiate the given programs, i.e., if they satisfy any additional requirements GROUNDER may impose, the obtained ground programs are fed into SOLVER through function `add`.

i		Rules	L
0	B	$p(0) \leftarrow not\ \neg p(0)$	
		$\neg p(0) \leftarrow not\ p(0)$	
		$\leftarrow p(0), \neg p(0)$	
1	$P[1]$	$a(1) \leftarrow not\ \neg a(1)$	
		$\neg a(1) \leftarrow not\ a(1)$	
		$p(1) \leftarrow a(1)$	
		$p(1) \leftarrow p(0), not\ \neg p(1)$	
		$\neg p(1) \leftarrow \neg p(0), not\ p(1)$	
		$\leftarrow p(1), \neg p(1)$	
		$\leftarrow a(1), \neg a(1)$	
	$Q[1](\alpha_1)$	$\leftarrow not\ \neg p(0), \alpha_1$	α_1
		$\leftarrow not\ p(1), \alpha_1$	
		$\leftarrow not\ \neg a(1), \alpha_1$	
		$\{\alpha_1\} \leftarrow$	
		$\leftarrow \alpha_0$	
2	$P[2]$	$a(2) \leftarrow not\ \neg a(2)$	
		$\neg a(2) \leftarrow not\ a(2)$	
		$p(2) \leftarrow a(2)$	
		$p(2) \leftarrow p(1), not\ \neg p(2)$	
		$\neg p(2) \leftarrow \neg p(1), not\ p(2)$	
		$\leftarrow p(2), \neg p(2)$	
		$\leftarrow a(2), \neg a(2)$	
	$Q[2](\alpha_2)$	$\leftarrow not\ \neg p(0), \alpha_2$	α_2
		$\leftarrow not\ p(2), \alpha_2$	
		$\leftarrow not\ \neg a(2), \alpha_2$	
		$\{\alpha_2\} \leftarrow$	
		$\leftarrow \alpha_1$	

Fig. 1: Tracing Algorithm 1: `isolve`.

In Line 7, 10, and 11 of Algorithm 1, cumulative and volatile program slices are handled according to the sequences of programs and assumptions, respectively, specified in Definition 7. Note that `isolve` terminates as soon as function `solve` of SOLVER reports some answer set. Otherwise, if no answer set is found in any step $i \geq 1$, `isolve` (in theory) loops forever on increasing values for k.

For illustrating `isolve`, reconsider the example in (2). We give in Figure 1 the accumulation of ground rules within the solver during the formation of the answer set containing $\{\neg p(0), a(1), p(1), \neg a(2), p(2)\}$. The left column shows the value of i in Algorithm 1, the middle one groups the rules added in Line 2, 7, and 10 of Algorithm 1, and the right one gives the assumption, α_i, used in each iteration. The rules accumulated within the solver at the end of the first iteration yield no answer set under assumption α_1, while the addition of the rules obtained in the next step yields the above answer set under assumption α_2. Note that this answer set also includes α_2, while it does not contain α_1 due to integrity constraint $\leftarrow \alpha_1$.

If GROUNDER is adequate and if SOLVER is sound, for a separated modular domain description $(B, P[k], Q[k])$ such that $P(\mathbb{Q}_i)$ (cf. Definition 2) has an answer set for some $i \geq 1$, `isolve` returns the answer sets of $P(\mathbb{Q}_i)$ for the least such $i \geq 1$.

Theorem 5. *Let* $(B, P[k], Q[k])$ *be a separated modular domain description, let* GROUNDER *be an adequate grounder, and let* SOLVER *be a sound solver. Let* $(P_i, O_i)_{i\geq 0}$ *and* $(Q_i, O'_i)_{i\geq 1}$ *be as in Definition 4 for* `ground` $=$ GROUNDER.`ground`*, and let* $(\mathbb{Q}_i)_{i\geq 1}$ *as in Definition 2. If* (P_0, O_0)*,* (P_i, O_i)*, and* (Q_i, O'_i) *are defined for all* $i \geq 1$*, we have* `isolve`$((B, P[k], Q[k])) = AS(P(\mathbb{Q}_i))$ *for the least* $i \geq 1$ *such that* $AS(P(\mathbb{Q}_i)) \neq \emptyset$*.*

Note that the above result builds upon the assumption that $(B, P[k], Q[k])$ is modular. When feeding a non-modular domain description (that GROUNDER can instantiate) into `isolve`, interpretations computed by SOLVER.`solve` do not necessarily match the answer sets of the combined program slices.

We next provide simple syntactic conditions under which B, $P[k]$, and $Q[k]$ assemble a modular domain description.

Proposition 4. *Let* $(B, P[k], Q[k])$ *be a domain description, and let* $\mathbf{P} = \bigcup_{i\geq 1} P[i]$ *and* $\mathbf{Q} = \bigcup_{i\geq 1} Q[i]$*. Then,* $(B, P[k], Q[k])$ *is modular if the following conditions hold*:

1. $atom(grd(B)) \cap (head(grd(\mathbf{P})) \cup head(grd(\mathbf{Q}))) = \emptyset$*,*
2. $atom(grd(\mathbf{P})) \cap head(grd(\mathbf{Q})) = \emptyset$*, and*
3. $\{head(grd(P[i])) \mid i \geq 1\}$ *is a partition of* $head(grd(\mathbf{P}))$*.*

Pragmatically, these conditions can be granted by using predicates not occurring in $B \cup P[k]$ for the heads of rules in $Q[k]$, and by including 0 as parameter in every atom of B as well as parameter k in the head of every rule in $P[k]$. Of course, parameter 0 can also be omitted in atoms of B if the corresponding predicates are not used in the heads of rules in $P[k]$. Recalling the domain descriptions given in (2) and (3), one can observe that the respective programs B, $P[k]$, and $Q[k]$ fit into this scheme. In fact, many problems over time parameters are naturally stated via modular domain descriptions.

5 Experiments with the Incremental ASP System *iclingo*

We implemented our approach to incremental ASP solving within the system *iclingo* by building on grounder *gringo* (2.0.0) and solver *clasp* (1.1.0) (all available at [7]). As input, *gringo* accepts λ-restricted programs, inducing finite equivalent ground programs. Procedurally, *iclingo* uses *gringo* as delineated in Algorithm 1. The customization of *clasp* conceptually affects two components, namely, the treatment of a program's completion and loop formulas, respectively. Note that neither of these adaptations would be necessary in a SAT solver, since the underlying semantics does not rely on Clark's completion. Over time, *clasp* accumulates ground program slices and, moreover, learns further constraints during solving. As a matter of fact, *clasp* is equipped with dynamic deletion and simplification techniques disposing of superfluous constraints.

Our experiments consider *iclingo* in four settings: keeping over successive solving steps *(1)* learned constraints, *(2)* learned constraints and heuristic values, *(3)* heuristic values only, and *(4)* neither. We compare these variants with iterative deepening search using *clingo*, the direct combination of *gringo* and *clasp* via an internal interface, as well as *gringo* and *clasp* via a textual interface (using the output language of *lparse* [12]).

Except for using different communication channels, *clingo* as well as piped *gringo* and *clasp* run identically, and *clingo* is consistently faster at a fraction of run-time.

The benchmarks in Table 1 belong to four different classes. In the Blocksworld example, the goal is to reconstruct a tower of n blocks in inverse order, requiring a plan of length n. In the Queens example, we compute (at most) one answer set for each value of k, iterating from 1 to n. For Sokoban and Towers of Hanoi, we use handmade instances from [15, 16], each instance requiring n steps for achieving its goal condition. With both of these planning problems, the default encoding includes the initial state in a base program B and the goal condition in a query program $Q[k]$. We also provide alternative encodings (attributed by "back" in Table 1), in which B contains the goal and $Q[k]$ the initial state. Table 1 summarizes run-time results in seconds, taking the average of three runs per instance. The rows marked with Σ show the sums of run-times over all instances of a benchmark class, also distinguishing encodings, with timeouts taken as 1200s. The last row ($\Sigma\Sigma$) sums run-times over all benchmark classes. All benchmarks as well as extended results are available at [7].

On the Blocksworld and Queens examples, we see that *iclingo* clearly outperforms *clingo* by one order of magnitude, which is primarily due to reduced grounding overhead. In fact, the simple Blocksworld problems are solved without any search, but *clingo* has to redo full grounding and propagation in each iterative deepening step, working on ground programs of considerable size. For example, considering the Blocksworld problem with four blocks, viz., $n = 4$, *gringo* produces 158 ground rules in the first step and 236 ground rules for each further step. While *iclingo* adds this number of rules in each incremental step, resulting in $158+(n-1)*236 = 866$ ground rules for $n = 4$, *clingo* processes $n*158+(n*(n-1)/2)*236 = 2048$ ground rules before obtaining a solution. Of course, the ratio of ground rules processed by *iclingo* gets even smaller as n increases, explaining the dramatic performance gains on Blocksworld. On the Queens example, we observe a similar effect, but here, *clasp* has to search for a solution for $n \geq 4$. Interestingly, *iclingo (1)*, keeping learned constraints, has a clear edge, but *iclingo (2)*, additionally keeping heuristic values, is by far the slowest among all *iclingo* variants. However, *iclingo (3)*, keeping heuristic values, is again consistently faster than *iclingo (4)*, keeping neither heuristic values nor learned constraints. This suggests that the strategy of *iclingo (2)* here tends to bias future runs too much, while a moderate amount of memory via either learned constraints or heuristic values is helpful.

Other than the simple Blocksworld and combinatorial Queens examples, Sokoban and Towers of Hanoi contain more realistic instances, shifting the focus to search for a plan. In fact, all systems underlie non-deterministic heuristic effects and traverse the search space differently. Though all systems spend most of their run-time in the solving component, the savings in grounding are still noticeable for *iclingo*, but smaller than on Blocksworld and Queens. On Sokoban, we observe varying relative performance of the considered systems on individual instances, which is due to the elevated difficulty of the problem. However, on the instance requiring the most steps, viz., $n = 21$, we have that the learning variants, *iclingo (1)* and *iclingo (2)*, perform much better than the remaining ones, *iclingo (3)* and *iclingo (4)*, which are also outperformed by *clingo*. The "back" encoding of Sokoban does not yield overall performance gains for any of the considered systems, but we observe that *iclingo (1)* copes best with this encoding. Note

Name	n	*iclingo (1)*	*iclingo (2)*	*iclingo (3)*	*iclingo (4)*	*clingo*	*gringo*|*clasp*
Blocksworld	20	2.61	2.61	2.62	2.62	37.09	42.41
	25	6.78	6.84	6.80	6.80	124.35	138.68
	30	15.68	15.80	15.71	15.81	330.15	362.39
	35	32.43	32.36	32.29	32.31	753.90	821.96
	40	60.99	60.75	60.71	61.04	-	-
	Σ	118.49	118.36	118.13	118.58	2445.49	2565.44
Queens	80	19.46	65.83	39.98	47.79	144.28	153.61
	90	36.72	135.19	70.81	81.70	249.13	264.21
	100	49.25	227.69	111.99	128.62	409.69	431.23
	110	64.05	424.03	176.16	201.67	636.91	669.75
	120	99.54	612.76	274.29	354.00	958.34	1003.67
	Σ	269.02	1465.50	673.23	813.78	2398.35	2522.47
Sokoban	16	243.22	287.46	320.07	334.08	376.74	384.41
	12	26.50	37.55	50.61	28.19	27.83	28.43
	16	124.26	124.44	320.97	341.94	189.48	194.12
	16	135.72	164.70	128.66	183.74	120.60	123.57
	18	140.80	145.07	233.71	275.12	236.60	242.19
	16	26.86	40.60	29.41	27.88	45.94	47.04
	17	1165.67	906.00	734.44	730.09	887.26	904.75
	14	119.95	140.11	106.40	213.22	96.26	98.10
	14	35.42	42.74	58.79	46.81	70.16	71.81
	21	286.46	200.43	600.19	777.68	278.97	285.09
	17	120.33	140.44	139.19	156.85	171.01	174.90
	14	39.09	36.21	36.00	47.48	66.12	67.43
	Σ	2464.28	2265.75	2758.44	3163.08	2566.97	2621.84
Sokoban back	16	-	-	-	-	-	-
	12	51.23	44.62	98.09	57.42	72.59	74.30
	16	264.81	201.48	265.21	359.38	296.45	302.46
	16	148.19	121.19	150.06	145.40	148.25	151.43
	18	723.07	-	-	-	1059.02	1081.34
	16	243.81	185.00	340.97	190.32	402.27	410.72
	17	599.74	714.40	1051.60	825.61	-	-
	14	149.37	126.04	164.98	191.33	170.36	173.74
	14	29.73	69.46	73.03	28.04	43.06	43.89
	21	346.56	428.43	400.81	295.69	402.78	411.70
	17	181.00	143.20	172.83	317.82	234.21	239.56
	14	15.06	58.45	39.27	17.50	59.63	60.78
	Σ	3952.57	4492.27	5156.85	4828.51	5288.62	5349.92
Towers	33	38.00	42.96	48.46	27.15	31.98	32.76
	34	61.40	36.78	47.09	45.95	61.77	63.39
	36	81.26	60.77	88.52	131.29	86.56	88.46
	39	223.46	155.76	184.63	204.13	216.89	222.74
	41	429.82	327.74	392.47	342.11	459.97	471.22
	Σ	833.94	624.01	761.17	750.63	857.17	878.57
Towers back	33	4.62	6.42	5.68	5.80	12.59	12.79
	34	55.79	33.42	56.27	42.39	52.80	54.00
	36	16.66	16.46	14.69	17.11	24.81	25.38
	39	27.88	25.43	28.60	32.83	46.01	46.85
	41	48.20	36.38	62.75	40.62	83.78	85.60
	Σ	153.15	118.11	167.99	138.75	219.99	224.62
	$\Sigma\Sigma$	7791.45	9084.00	9635.81	9813.33	13776.59	14162.86

Table 1. Benchmark results on a 2.2GHz PC under Linux; each run limited to 1200s time.

that both the initial and the goal states of Sokoban instances are total. Hence, with both encodings, *clasp* searches for a trajectory from one complete state to another. Finally, on Towers of Hanoi, the differences between the systems are rather small, and all of them show significant gains on the "back" encoding. In contrast to Sokoban, goal conditions do here not define total states. Thus, learning may further constrain the goal in B, while the total initial state in $Q[k]$ can easily be propagated. The differences between Sokoban and Towers of Hanoi regarding the impact of encodings show that incremental problems constitute a whole new setting, different from traditional ones, and further investigations are needed for optimizing computational strategies to deal with them.

6 Discussion

We presented the first theoretical and practical account of incremental ASP solving. Our framework allows for tackling bounded problems in ASP, paving the way for more ambitious real-world applications. Our approach is driven by the desire to minimize redundancies while gradually treating program slices. However, fixing the incremental solving process required the integration and adaption of successive grounding and solving steps in a globally consistent way. To this end, we developed an incremental module theory guiding the formal setting of iterative grounding and solving by means of existing ASP grounders and solvers. Module theory does not only provide us with a natural semantics for non-ground, parametrized program slices but moreover makes precise their composition by appeal to input/output interfaces. Such compositionality provides the primary basis for incremental computations. Our experimental results indicate the computational impact of our incremental approach on parametrized domain descriptions. While savings in grounding are evident, on different encodings of search-intensive problems, we have seen that the effectiveness of solving techniques in an incremental setting is (currently) less predictable. Indeed, incremental problems differ from traditional ones, so that dedicated computational strategies for them can be developed and explored. In this respect, our system *iclingo* makes merely a first step. Future work also includes more elaborate incremental algorithms than `isolve`, allowing for non-elementary program slices while still guaranteeing optimality of solutions.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Kautz, H., Selman, B.: Planning as satisfiability. Proc. of ECAI'92, Wiley (1992) 359–363
3. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
4. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153**(1-2) (2004) 49–104
5. Gelfond, M., Lifschitz, V.: Action languages. Electron. Trans. on AI **3**(6) (1998) 193–210
6. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proc. of IJCAI'07, AAAI Press (2007)
7. `http://www.cs.uni-potsdam.de/wv/software`
8. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. Proc. of ECAI'06, IOS Press (2006) 412–416
9. Lifschitz, V., Turner, H.: Splitting a logic program. Proc. of ICLP, MIT Press (1994) 23–37
10. Brass, S., Dix, J.: Semantics of (disjunctive) logic programs based on partial evaluation. Journal of Logic Programming **40**(1) (1999) 1–46
11. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. Proc. of LPNMR'04, Springer (2004) 87–99
12. `http://www.tcs.hut.fi/Software`
13. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
14. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science **89**(4) (2003)
15. `http://www.ne.jp/asahi/ai/yoshio/sokoban/handmade/`
16. `http://asparagus.cs.uni-potsdam.de/`