Stepping through an Answer-Set Program*

Johannes Oetsch, Jörg Pührer, and Hans Tompits

Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at

Abstract. We introduce a framework for interactive stepping through an answer-set program as a means for debugging. In procedural languages, stepping is a widespread and effective debugging strategy. The idea is to gain insight into the behaviour of a program by executing statement by statement, following the program's control flow. Stepping has not been considered for answer-set programs so far, presumably because of their lack of a control flow. The framework we provide allows for stepwise constructing interpretations following the user's intuition on which rule instances to become active. That is, we do not impose any ordering on the rules but give the programmer the freedom to guide the stepping process. Due to simple syntactic restrictions, each step results in a state that guarantees stability of the intermediate interpretation. We present how stepping can be started from breakpoints as in conventional programming and discuss how the approach can be used for debugging using a running example.

Keywords: answer-set programming, program analysis, debugging

1 Introduction

Answer-set programming (ASP) is a well-established paradigm for declarative problem solving [1], yet it is rarely used by engineers outside academia so far. Arguably, one particular obstacle preventing software engineers from using ASP is the lack of support tools for developing answer-set programs.

In this paper, we introduce a framework that allows for stepping through answer-set programs. Step-by-step execution of a program is folklore in procedural programming languages, where developers can debug and investigate the behaviour of their programs in an incremental way. As answer-set programs have a genuine declarative semantics lacking any control flow, it is not obvious how stepping can be realised. Our approach makes use of a simple computation model that is based on states, which are ground rules that a user considers as active in a program. With each state, we associate the interpretation that is induced by the respective rules. This interpretation is guaranteed to be an answer set of the set of rules considered in the state. During the course of stepping, the interpretations of the subsequent states evolve towards an answer set of the overall program. Our stepping approach is *interactive* and *incremental*, letting the programmer choose which rules are added at each step. In our framework, states may

* This work was partially supported by the Austrian Science Fund (FWF) under project P21698 and by the European Commission under project IST-2009-231875 (OntoRule).

serve as *breakpoints* from which stepping can be started. We discuss how the programmer can generate breakpoints that allow him or her to jump directly to interesting situations. We also show how ground rules that are subsequently considered active can be quickly obtained from the non-ground source code using filtering techniques.

The main area of application of stepping is debugging. A general problem in debugging is to restrict the amount of debugging information that is presented to the user in a sensible way. In the stepping method, this is realised by focussing on one step at a time, which is in contrast to other debugging methods for ASP [2–5], where the program to be debugged is analysed as a whole. Moreover, due to the interactivity of the approach, the programmer can easily guide the search for bugs following his or her intuitions about which part of the program is likely to be the source of error. Besides debugging, stepping through a program can improve the understanding of the program at hand and can help to improve the understanding of the answer-set semantics for beginners.

The paper is outlined as follows. Section 2 gives the formal background on ASP. The framework for stepping is presented in Section 3. In Section 4, we explain how breakpoints can be generated and how ground rules can be conveniently selected as active ones. Moreover, we describe interesting settings where stepping can be beneficially applied using a running example. After discussing related work in Section 5, we conclude the paper in Section 6.

2 Preliminaries

We deal with *logic programs* which are finite sets of rules of form

$$l_0 \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n, \tag{1}$$

where $n \geq m \geq 0$, "not" denotes *default negation*, and all l_i are literals over a function-free first-order language $\mathcal{L}$. A literal is an atom possibly preceded by the *strong negation* symbol $\neg$. For a literal l, we define $\bar{l} = \neg a$ if $l = a$ and $\bar{l} = a$ if $l = \neg a$. In the sequel, we assume that $\mathcal{L}$ will be implicitly given. The literal l_0 may be absent in (1), in which case the rule is a *constraint*. Furthermore , for r of form (1), $\mathrm{B}(r) = \{l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n\}$ is the *body* of r, $\mathrm{B}^+(r) = \{l_1, \ldots, l_m\}$ is the *positive body* of r, and $\mathrm{B}^-(r) = \{l_{m+1}, \ldots, l_n\}$ is the *negative body* of r. The *head* of r is $\mathrm{H}(r) = \{l_0\}$ if l_0 is present and $\mathrm{H}(r) = \emptyset$ otherwise. If $\mathrm{B}(r) = \emptyset$ and $\mathrm{H}(r) \neq \emptyset$, then r is a *fact*. For facts, we usually omit the symbol "$\leftarrow$". Furthermore, we identify sets of literals with sets of facts.

A literal, rule, or program is *ground* if it contains no variables. Let C be a set of constants. A *substitution over* C is a function ϑ assigning each variable in some expression an element of C. We denote by $e\vartheta$ the result of applying ϑ to an expression e. The *grounding* of a program Π, $gr(\Pi)$, is defined as usual.

An *interpretation* I (over some language $\mathcal{L}$) is a finite set of ground literals (over $\mathcal{L}$) such that $\{a, \neg a\} \not\subseteq I$, for any atom a. The satisfaction relation $I \models \alpha$, where α is a ground atom, a literal, a rule, a set of possibly default negated literals, or a program α, is defined in the usual manner. A rule r such that $I \models \mathrm{B}(r)$ is called *active* under I. We denote the set of all active rules of a ground program Π with respect to an interpretation I as $Act^I(\Pi) = \{r \in \Pi \mid I \models \mathrm{B}(r)\}$. Following Faber, Leone, and Pfeifer [6], we

define an *answer set* of a program Π as an interpretation I that is a minimal model of $Act^I(gr(\Pi))$. For the programs we consider, this definition is equivalent to the traditional one by Gelfond and Lifschitz [7]. The collection of all answer sets of a program Π is denoted by $AS(\Pi)$.

3 Stepping Framework

In this section, we introduce the basic computation model that underlies our stepping approach. We are aiming for a scenario in which the programmer has strong control over the direction of the construction of an answer set. The general idea is to first take a part of a program and an answer set of this part. Then, step by step, rules are added by the user such that, at every step, the literal derived by the new rule is added to the interpretation which remains to be an answer set of the evolving program part. Hereby, the user only adds rules he or she thinks are active in the final answer set. The interpretation grows monotonically until it is eventually guaranteed to be an answer set of the overall program, otherwise the programmer is informed why and at which step something went wrong. This way, one can in principle without any backtracking direct the computation towards an expected or an unintended actual answer set. In debugging, having the programmer in the role of an oracle is a common scenario [8]. It is reasonable to assume that a programmer has good intuitions on where to guide the search if there is a mismatch between the intended and the actual behaviour of a program.

In our framework, the individual steps of a computation—which we regard as *states* of the program—are represented by a set of ground rules which the user considers as active. While these rules represent the state on the source-code level, close to what the programmer has written, we also want to represent a state on the output level in the form of an interpretation that constitutes a partial result of the program. Therefore, we associate a set of ground rules with the interpretation induced by the rules.

Definition 1. *Let S be a set of ground rules. Then, the* interpretation induced by *S is given by $Int[S] = \bigcup_{r \in S} \mathrm{H}(r)$.*

States have to satisfy two properties, ensuring that the interpretation induced by the state is an answer set of the state and that every rule in the state is active with respect to the interpretation. Intuitively, we want every step in the construction of an answer set to result in a stable condition, where we only have rules that are relevant to this condition. The metaphor for that is building up a house of cards, where every card—being the counterpart of a rule—supports the integrity of the evolving house—corresponding to the interpretation—and stability of the house must be ensured after each building activity.

Definition 2. *A set S of ground rules is* self-supporting *if $Int[S] \models \mathrm{B}(r)$, for all $r \in S$, and* stable *if $Int[S] \in AS(S)$. A* state *of a program Π is a set $S \subseteq gr(\Pi)$ of ground rules which is self-supporting and stable.*

Every state can be used as a potential starting point for stepping that allows a programmer to jump directly to an interesting situation, e.g., for debugging purposes.

We next define a successor relation between states and sets of ground rules. The intuition is that a successor of a state S corresponds to a potential state after one step in a computation.

Definition 3. *For a state S of a program Π and a set $S' \subseteq gr(\Pi)$ of ground rules, S' is a* successor *of S in Π, symbolically $S \prec_\Pi S'$, if $S' = S \cup \{r\}$, for some rule $r \in gr(\Pi) \setminus S$ with (i) $Int[S] \models \mathrm{B}(r)$, (ii) $\mathrm{H}(r) \neq \emptyset$, and (iii) $\mathrm{H}(r) \cap (\mathrm{B}^-(r) \cup \bigcup_{r' \in S} \mathrm{B}^-(r') \cup \bigcup_{l \in Int[S]} \bar{l}) = \emptyset$,*

Intuitively, rule r is a rule instance of the program Π that is not yet considered in the current state S but whose preconditions are already satisfied by the state's interpretation, as expressed by Condition (i). Conditions (ii) and (iii) ensure that r is not a constraint and that the literal derived by r is neither inconsistent with $Int[S]$ nor contradicting that all rules in the S' are active. Note that, in general, $Int[S] \subseteq Int[S']$ while $S \subset S'$. The successor relation suffices to "step" from one state to another, i.e., S' is always a state.

Proposition 1. *Let S be a state of a program Π, and $S' \subseteq gr(\Pi)$ a set of ground rules such that $S \prec_\Pi S'$. Then, S' is also a state of Π.*

In the following, we define *computations* based on the notion of a state.

Definition 4. *A* computation *for a program Π is a finite sequence $\mathrm{C} = S_0, \ldots, S_n$ of states such that, for all $0 \leq i < n$, $S_i \prec_\Pi S_{i+1}$.*

Given a computation $\mathrm{C} = S_0, \ldots, S_n$ for a program Π, in analogy to stepping in procedural programs, we identify the state S_0 at which computation C starts as the *breakpoint* of C. Furthermore, $Int[S_n]$ is called the *result*, $res(\mathrm{C})$, of C.

We next define when a computation has failed, gets stuck, or is complete. Intuitively, failure means that the computation reached a point where no answer set of the program can be reached. A computation is stuck when the last state activated rules deriving literals that are inconsistent with previously chosen active rules. It is considered complete when there are no more unconsidered active rules.

Definition 5. *A computation $\mathrm{C} = S_0, \ldots, S_n$ for Π*
- *has* failed at Step i *if there is no answer set I of Π such that $S_i \subseteq Act^I(gr(\Pi))$;*
- *is* stuck *if there is no successor of S_n in Π but there is a rule $r \in gr(\Pi) \setminus S_n$ that is active under $Int[S_n]$;*
- *is* complete *if, for each rule $r \in gr(\Pi)$ that is active under $Int[S_n]$, we have $r \in S_n$.*

The following result guarantees the soundness of our stepping framework.

Theorem 1. *Let Π be a program and $\mathrm{C} = S_0, \ldots, S_n$ a complete computation for Π. Then, $res(\mathrm{C})$ is an answer set of Π.*

The computation model is also complete in the sense that stepping, starting from an arbitrary state S_0 of Π as breakpoint, can reach every answer set $I \supseteq Int[S_0]$ of Π, where $S_0 \subseteq Act^I(gr(\Pi))$.

Theorem 2. *Let $I \in AS(\Pi)$ be an answer set of program Π and S_0 a state of Π such that $Int[S_0] \subseteq I$ and $S_0 \subseteq Act^I(gr(\Pi))$. Then, there is a complete computation $\mathrm{C} = S_0, \ldots, S_n$ with $S_n = Act^I(gr(\Pi))$ and $res(\mathrm{C}) = I$.*

Example 1. Consider the program

$$\Pi = \{obj(c), obj(d), \leftarrow ch(c), \ ch(X) \leftarrow \text{not } \neg ch(X), obj(X), \\ \neg ch(X) \leftarrow \text{not } ch(X), obj(X)\}.$$

The answer sets of Π are $I_1 = \{obj(c), obj(d), \neg ch(c), ch(d)\}$ and $I_2 = \{obj(c), obj(d), \neg ch(c), \neg ch(d)\}$. Consider the computation $\mathrm{C} = S_0, S_1, S_2$, where $S_0 = \{obj(c), obj(d)\}$, $S_1 = S_0 \cup \{\neg ch(c) \leftarrow \text{not } ch(c), obj(c)\}$, and $S_2 = S_1 \cup \{ch(d) \leftarrow \text{not } \neg ch(d), obj(d)\}$. Computation C is complete and $res(\mathrm{C}) = I_1$. Consider now the computation $\mathrm{C}' = S'_0, S'_1, S'_2$, where $S'_0 = \{obj(c), obj(d)\}$, $S'_1 = S'_0 \cup \{ch(c) \leftarrow \text{not } \neg ch(c), obj(c)\}$, and $S'_2 = S'_1 \cup \{ch(d) \leftarrow \text{not } \neg ch(d), obj(d)\}$. C' has failed at Step 1 as there is no answer set I of Π such that $Act^I(gr(\Pi))$ contains $ch(c) \leftarrow \text{not } \neg ch(c), obj(c)$. Moreover, C' is stuck as there is no state succeeding S'_2 but $\leftarrow ch(c)$ is active under $Int[S'_2]$.

Observe that once a computation C has failed at some step all computations that contain C as subsequence are guaranteed to get stuck. Hence, when failure is detected at the current step, the user knows that the last active rule chosen is crucial for the targeted interpretation not to be an answer set. Failure of a computation does not mean that it is useless for debugging. In fact, when a program Π does not have any answer set, building up a computation for Π will guide the user to rules responsible for the inconsistency.

The next corollary is a consequence of the fact that the empty set is a state of every program.

Corollary 1. *Let Π be a program and I an answer set of Π. Then, there is a complete computation $\mathrm{C} = S_0, \ldots, S_n$ such that $S_0 = \emptyset$, $S_n = Act^I(gr(\Pi))$, and $res(\mathrm{C}) = I$.*

The programmer will typically want to start with another breakpoint than the empty set. As we argue in Section 4, obtaining a breakpoint that is "near" to an interesting situation is desirable and, using the programmer's intuition, in most cases not difficult to achieve.

4 Interactive Stepping

In this section, we outline how the framework introduced in the previous section can be used in practice. We will use the *maze-generation problem*, a benchmark problem from the second ASP competition [9], as a running example. The task is to generate a two-dimensional grid where each cell is either a wall or empty. There are two dedicated empty cells located at the border, being the maze's entrance and its exit, respectively. The maze grid has to satisfy the following conditions: (i) except for the entrance and the exit, border cells are walls; (ii) there must be a path from the entrance to every empty cell (including the exit); (iii) if two walls are diagonally adjacent, one of their common neighbours is a wall; (iv) there must not be any 2×2 block of empty cells or walls; and (v) no wall can be completely surrounded by empty cells. The input of this problem are facts that specify the size and the positions of the entrance and the exit of the maze as well as facts that specify for an arbitrary subset of the cells whether they are walls or empty. The output corresponds to completions of the input that represent valid maze structures. As example input, we consider the following set of facts

$$F = \{row(1), row(2), row(3), row(4), row(5), col(1), col(2), col(3), col(4), col(5), \\ entrance(1,2), exit(5,4), wall(3,3), empty(3,4)\}$$

that is visualised in Fig. 1 together with a completion to a legal maze. White cells correspond to empty cells, black cells to walls, and grey areas are yet unassigned cells.

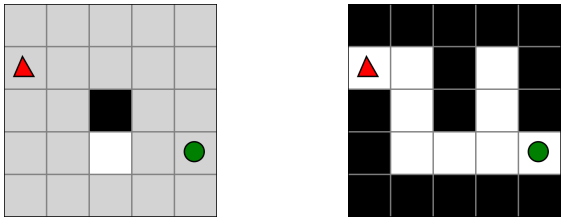

Fig. 1. A maze-generation input with a corresponding solution.

The entrance is marked by a triangle and the exit by a circle. The program developed in the sequel follows the encoding submitted by the Potassco team.[1] Note that the language used in the example is slightly richer than defined in Section 2. We allow for integer arithmetics and assume a sufficient integer range to be available as constants.

In our approach, we always have two options how to proceed: (i) (re-)initialise stepping and start a computation with a new state as breakpoint, or (ii) extend the current computation by adding a further active rule.

In the remainder of the section, we first describe the technical aspects of how to obtain a breakpoint and how ground rule instances can be chosen. Then, we discuss how stepping can be applied in several situations, including typical debugging scenarios.

4.1 Obtaining a Breakpoint

Having a suitable breakpoint at hand will often allow for finding a bug in just a few steps. As mentioned above, the empty set is a trivial state for every program. Besides that, the set of all facts in a program is also ensured to be a state, except for the practically irrelevant case that a literal and its strong negation are asserted.

Example 2. As a first step for developing the maze-generation encoding, we want to identify border cells. Our initial program is $\Pi_0 = F \cup \Pi_{\mathrm{Bdr}}$, where Π_{Bdr} is given by

$$\begin{aligned}
maxcol(X) &\leftarrow col(X), \text{not } col(X+1),\\
maxrow(Y) &\leftarrow row(Y), \text{not } row(Y+1),\\
border(1,Y) &\leftarrow col(1), row(Y),\\
border(X,Y) &\leftarrow row(Y), maxcol(X),\\
border(X,1) &\leftarrow col(X), row(1),\\
border(X,Y) &\leftarrow col(X), maxrow(Y).
\end{aligned}$$

The first two rules extract the numbers of columns and rows of the maze from the input facts of predicates $col/1$ and $row/1$. The next four rules derive $border/2$ atoms for the grid.

Now, taking the set F of facts as a breakpoint of a computation for Π_0, we can start stepping by choosing, e.g., the ground rule

$$r = maxcol(5) \leftarrow col(5), \text{not } col(6),$$

[1] See also `http://dtai.cs.kuleuven.be/events/ASP-competition/Teams/Potassco.shtml`.

that is active under F, as next rule to add. We obtain the computation $\mathrm{C} = F, F \cup \{r\}$.

In many cases, it will be useful to have states other than the empty set or the facts as starting points, as starting stepping from them can be time consuming. For illustration, to reach an answer set I of a program, the minimum length of a computation starting from the empty set is $|I|$. We next show how states that may serve as breakpoints can be generated. A state can be obtained by computing an answer set X of a trusted part of a program (or its grounding) and then selecting rule instances that are active under X.

Proposition 2. *Let Π be a program and $\Pi' \subseteq \Pi \cup gr(\Pi)$ such that $I \in AS(\Pi')$. Then, $Act^I(gr(\Pi'))$ is a state of Π.*

Hence, it suffices to find an appropriate Π' in order to get breakpoints. One option for doing so is to let the user manually specify Π' as a subset of Π (including facts).

Example 3. We want to step through the rules that derive the $border/2$ atoms. As we pointed out above, the respective definitions rely on information about the size of the maze. Hence, we will use a breakpoint where the rules deriving $maxcol/1$ and $maxrow/1$ were already applied. Following Proposition 2, we calculate an answer set of program $\Pi'_0 \subseteq \Pi_0$ that is given by $\Pi'_0 = F \cup \{maxcol(X) \leftarrow col(X), \mathrm{not}\ col(X+1), maxrow(Y) \leftarrow row(Y), \mathrm{not}\ row(Y+1)\}$. The unique answer set of Π'_0 is $I_0 = F \cup \{maxcol(5), maxrow(5)\}$. The desired breakpoint S_0 is given by $S_0 = Act^{I_0}(gr(\Pi'_0))$, which consists of the facts in F and the rules $maxcol(5) \leftarrow col(5), \mathrm{not}\ col(6)$ and $maxrow(5) \leftarrow row(5), \mathrm{not}\ row(6)$.

Note that if the subprogram Π' for breakpoint generation has more than one answer set, the selection of the set $I \in AS(\Pi')$ is based on the programmer's intuition, similar to selecting the next rule in stepping.

A different application of Proposition 2 is *jumping* from one state to another by considering further non-ground rules. This makes sense, e.g., in a debugging situation where the user initially started with a breakpoint S that is considered as an early state in a computation. After few steps and reaching state S', the user realises that the computation from S to S' seems to be as intended and wants to proceed at a point where more literals have already been derived, i.e., after applying a selection Π'' of non-ground rules from Π on top of the interpretation $Int[S']$ associated with S'. Then, Π' is given by $\Pi' = S' \cup \Pi''$. Note that, for an arbitrary answer set I of $AS(\Pi')$, it is not ensured that there is a computation of Π starting from S' and ending with the fresh state $Act^I(gr(\Pi'))$. The reason is that there might be rules in S' that are not active under I. If the programmer wants to assure that there is a computation of Π starting from S' and ending with $Act^I(gr(\Pi'))$, Π' can be joined with the set $Con_{S'} = \{\leftarrow \mathrm{not}\ l \mid l \in \mathrm{B}^+(r), r \in S'\} \cup \{\leftarrow l \mid l \in \mathrm{B}^-(r), r \in S'\}$ of constraints.

Jumping from one state to another is also needed if the user wants to skip several steps and considers one or more non-ground rules instead.

Example 4. Assume the program Π_0 has been extended to Π_1 by adding the rule

$$r_{\mathrm{bw}} = wall(X,Y) \leftarrow border(X,Y), \mathrm{not}\ entrance(X,Y), \mathrm{not}\ exit(X,Y)$$

that ensures that every border cell is a wall, except for the entrance and the exit. As $\Pi_0 \subseteq \Pi_1$, the state S_0 of Π_0 is also a state of Π_1. Hence, assume we started stepping Π_1 from S_0 and successively added the rules $border(1,1) \leftarrow col(1), row(1)$ and $border(1,2) \leftarrow col(1), row(2)$. Let S_1 be the resulting state, i.e., the union of these rules and S_0. The cells $(1,1)$ and $(1,2)$ have been identified as border cells. According to the problem specification, cell $(1,1)$ should be a wall, and $(1,2)$ should be empty as it contains the entrance. To test whether our current program realises that, we want to apply the non-ground rule r_{bw} on top of S_1. Therefore, we use Proposition 2 by first computing the answer set I_1 of $\Pi_1' = S_1 \cap \{r_{\mathrm{bw}}\}$ that is given by $I_1 = I_0 \cup \{border(1,1), border(1,2), wall(1,1)\}$ as expected. The new state is $S_2 = Act^{I_1}(gr(\Pi_1'))$.

4.2 Stepping

To obtain a successor of a given state S of program Π, by Definition 3 we need a rule $r \in gr(\Pi) \setminus S$ with (i) $Int[S] \models \mathrm{B}(r)$, (ii) $\mathrm{H}(r) \neq \emptyset$, and (iii) $\mathrm{H}(r) \cap (\mathrm{B}^-(r) \cup \bigcup_{r' \in S} \mathrm{B}^-(r') \cup \bigcup_{l \in Int[S]} \bar{l}) = \emptyset$. One can proceed in the following fashion: First, a non-ground rule $r \in \Pi$ with $\mathrm{H}(r) \neq \emptyset$ is selected for instantiation. Then, the user assigns constants to the variables occurring in r. Both steps can be assisted by filtering techniques. A stepping system can provide the user with information which non-ground rules in Π have instances that are active under $Int[S]$ but not contained in S. This can be done using ASP itself using *meta-programming* and *tagging transformations* [5, 2].

Example 5. The next version, Π_2, of the maze-generation encoding is obtained by joining Π_1 with the following set Π_{guess} of rules:

$$\begin{aligned} wall(X,Y) &\leftarrow \mathrm{not}\ empty(X,Y), col(X), row(Y), \\ empty(X,Y) &\leftarrow \mathrm{not}\ wall(X,Y), col(X), row(Y), \\ &\leftarrow entrance(X,Y), wall(X,Y), \\ &\leftarrow exit(X,Y), wall(X,Y). \end{aligned}$$

Program Π_{guess} guesses whether a cell is a wall or empty while assuring that no wall is guessed on an entrance or exit cell. We start the stepping session from breakpoint $S_3 = Act^{I_3}(gr(\Pi_1))$, where $AS(\Pi_1) = \{I_3\}$. Note that the answer set I_3 of Π_1, which is also the interpretation induced by S_3, encodes a situation where the cells from the input as well as the border cells have already been assigned to be a wall or empty (cf. Fig. 2). There are only two non-ground rules in Π_2 that have active ground instances under I_3 that are not yet contained in S_3: $wall(X,Y) \leftarrow \mathrm{not}\ empty(X,Y), col(X), row(Y)$, and $empty(X,Y) \leftarrow \mathrm{not}\ wall(X,Y), col(X), row(Y)$. An advanced source editor may directly highlight the two rules.

A user can also get assistance for a variable assignment. By assigning the variables in r one after the other, the domains of the remaining ones can always be accordingly restricted such that there is still a compatible ground instance of r that is active under $Int[S]$. Consider a partial substitution ϑ assigning constants in Π to some variables in r. When fixing the assignment of a further variable X occurring in $\mathrm{B}(r)$, where $\vartheta(X)$ is yet undefined, we may choose only a constant c such that there is a substitution ϑ' with

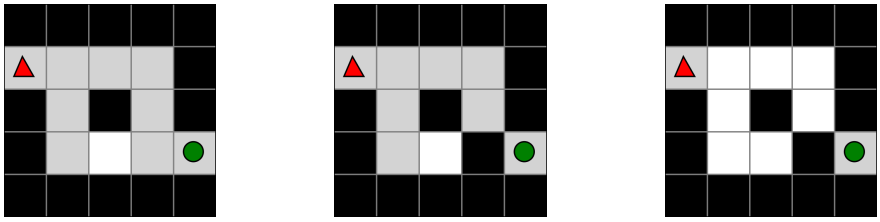

Fig. 2. Visualisation of I_3, I_4, and I_5 from Examples 5, 6, and 9.

- $\vartheta'(X') = \vartheta(X')$, where $\vartheta(X')$ is defined,
- $\vartheta'(X) = c$,
- $l\vartheta' \in \mathit{Int}[S]$, for all $l \in \mathrm{B}^+(r)$, and
- $l\vartheta' \notin \mathit{Int}[S]$, for all $l \in \mathrm{B}^-(r)$.

Respective computations can be done by a simple ASP meta-program that guesses ϑ' given r, ϑ, and $\mathit{Int}[S]$, and checks the conditions above plus $r\vartheta' \notin S$.

ASP solvers typically require safety, i.e., all variables occurring in r must also occur in $\mathrm{B}^+(r)$. Thus, the constants to be considered are restricted to those that appear in literals in $\mathit{Int}[S]$ to which literals $\mathrm{B}^+(r)$ can be substituted to. If safety is not given however, all constants in Π have to be considered for the respective substitutions.

Once a substitution ϑ for all variables in r is found, we check whether the newly obtained ground instance $r' = r\vartheta$ satisfies the final condition of Definition 3, i.e., checking whether the head of r' is consistent with all rules in the potential successor state of S being active. If this is not the case, the user's intention that for the considered stepping choices rule r' can be active was wrong.

Example 6. We resume the stepping session of Example 5 at state S_3 and choose rule $r_w = \mathit{wall}(X, Y) \leftarrow \text{not } \mathit{empty}(X, Y), \mathit{col}(X), \mathit{row}(Y)$ for instantiation. There are 24 instances of r_w that are active under I_3. Each such instance corresponds to $r_w\vartheta$, where $\vartheta(X), \vartheta(Y) \in \{1, \ldots, 5\}$ and not both $\vartheta(X) = 3$ and $\vartheta(Y) = 4$. Assume we want to determine the assignment $\vartheta(Y)$ for variable Y first. We choose $\vartheta(Y) = 4$ and determine the value of X next. Filtering now leaves only 1, 2, 4, and 5 as options. We define $\vartheta(X) = 4$ and use the obtained ground instance $r'_w\vartheta = \mathit{wall}(4, 4) \leftarrow \text{not } \mathit{empty}(4, 4), \mathit{col}(4), \mathit{row}(4)$ to step to state $S_4 = S_3 \cup r'_w$. The interpretation $I_4 = \mathit{Int}[S_4]$ is visualised in Fig. 2.

4.3 Application Scenarios

Stepping to an answer set. Stepping until an answer set of a program is reached can be helpful in many situations. Besides the general benefit of getting insights into the interplay of rules of a program, stepping can be used to search for a particular answer set when a program has many of them.

Example 7. Program Π_2 has 128 answer sets and does not yet incorporate all constraints from the problem specification. All answer sets that correspond to valid maze-generation

solutions for the given problem instance are among them. Starting stepping from breakpoint S_3, corresponding to the visualisation of I_3 in Fig. 2, we only need nine steps to get to a state S_{sol} where $Int[S_{\mathrm{sol}}]$ encodes the solution depicted in Fig. 1. In fact, we just need to add instances of the rule $empty(X,Y) \leftarrow \text{not } wall(X,Y), col(X), row(Y)$ from Π_{guess} for each unassigned cell (X,Y).

Whenever a state S is reached and $I = Int[S]$ is a desired answer set (projected to interesting literals) of a yet unfinished program, we can make use of the obtained interpretation I for further developing the program. For example, later versions of the program can be tested for being consistent with the intended solution I. If they are not, I can be used as input in a debugging approach, e.g., like the one from earlier work [5] that gives reasons why I is not an answer set of a program.

If the guessing part of a program is extensive, i.e., involving a large number of literals, the guessing rules have to be considered already when obtaining the breakpoint using Proposition 2. If the user has special requirements regarding the guess, for instance that certain cells are not walls, they can be added as constraints when computing the new breakpoint. It is advisable, however, to use small problem instances for testing during program development. Using 5×5 mazes as in our example will be sufficient for realising a reliable encoding and makes stepping easier as computations will be shorter.

Absence of answer sets. A common situation when writing an answer-set program is that the program's current version is unexpectedly incoherent, i.e., it does not yield any answer sets. A usual debugging strategy is to individually remove the constraints of the program to identify which one yields the incoherence. It may be the case that absence of answer sets is not caused by constraints (e.g., contradictory literals, odd loops through negation), but unfortunately, as we will see in the next example, even when the bug is due to constraints, removing constraints is not always sufficient to locate the error.

Example 8. As next features of the maze-generation program, we (incorrectly) implement rules that should express that there has to be a path from the entrance to every empty cell and that 2×2 blocks of empty cells are forbidden. We obtain a new version, Π_3, by joining Π_2 with the rules

$$\begin{aligned}
adjacent(X,Y,X,Y+1) &\leftarrow col(X), row(Y), row(Y+1),\\
adjacent(X,Y,X,Y-1) &\leftarrow col(X), row(Y), row(Y-1),\\
adjacent(X,Y,X+1,Y) &\leftarrow col(X), row(Y), col(X+1),\\
adjacent(X,Y,X-1,Y) &\leftarrow col(X), row(Y), col(X-1),\\
reach(X,Y) &\leftarrow entrance(X,Y), \text{not } wall(X,Y),\\
reach(X_2,Y_2) &\leftarrow adjacent(X_1,Y_1,X_2,Y_2), reach(X_1,Y_1),\\
&\qquad \text{not } wall(X_2,Y_2),
\end{aligned}$$

formalising when an empty cell is reached from the entrance, and the constraints

$$\begin{aligned}
c_1 &= \leftarrow empty(X,Y), \text{not } reach(X,Y),\\
c_2 &= \leftarrow empty(X,Y), empty(X+1,Y), empty(X,X+1), empty(X+1,Y+1)
\end{aligned}$$

to ensure that every empty cell is reached and that no 2×2 blocks of empty cells exist.

Assume that we did not spot the bug contained in c_2—in the third body literal the term $Y + 1$ was mistaken for $X + 1$. This could be the result of a typical copy-paste error. It turns out that Π_3 has no answer set. As we already trust the previous version Π_2 and expect the inconsistency to be caused by one of the constraints, the most obvious strategy is to remove one of c_1 or c_2. It turns out that both $\Pi_3 \setminus \{c_1\}$ as well as $\Pi_3 \setminus \{c_2\}$ do have answer sets, 84 answer sets in the former case and ten in the latter case. Inspecting ten answer sets manually is tedious but still manageable. Thus, one could go through them and check whether some of them encode proper maze-generation solutions. After doing so, c_2 would be identified as suspicious. An alternative approach—also feasible if there were more than ten answer sets to expect—is to use the approach of Example 7 and start a computation for Π_3 towards an intended solution, e.g., the one from Fig. 1, at breakpoint S_3. As soon as we reach a state S_{c_2} where the cells $(1, 2)$, $(2, 1)$ and $(2, 3)$ are considered to be empty, there is an instance of constraint c_2 which becomes active with respect to the interpretation induced by S_{c_2}. As noted in Section 4.2, automatic checks after each step could be used to reveal and highlight non-ground rules with active instances. In the case of active constraint instances, it would even be sensible to explicitly warn the programmer. Using the filtering techniques for variable substitutions, the user can be guided to a concrete active instance of c_2, viz. to $c_2' = \leftarrow \mathit{empty}(1, 2), \mathit{empty}(2, 2), \mathit{empty}(1, 2), \mathit{empty}(2, 3)$. It is obvious that the cells $(1, 2)$,$(2, 2)$,$(1, 2)$, and $(2, 3)$ do not form a valid 2×2 block and hence the wrong term in c_2 can be easily detected. A correct program Π_4 is obtained from Π_3 by changing c_2. As shown in Example 7, depending on the order in which the rules are added, c_2' becomes active in at most nine steps when reaching state S_{sol}.

Understanding someone else's code. Reading and understanding a program written by another developer can be difficult. Stepping through such a program can be quite helpful.

Example 9. Assume we were provided with the code

$$
\begin{aligned}
c_3 &= \leftarrow \mathit{wall}(X, Y), \mathit{wall}(X + 1, Y), \mathit{wall}(X, Y + 1), \mathit{wall}(X + 1, Y + 1),\\
c_4 &= \leftarrow \mathit{wall}(X, Y), \mathit{wall}(X + 1, Y + 1), \mathrm{not}\ \mathit{wall}(X + 1, Y), \mathrm{not}\ \mathit{wall}(X, Y + 1),\\
c_5 &= \leftarrow \mathit{wall}(X + 1, Y), \mathit{wall}(X, Y + 1), \mathrm{not}\ \mathit{wall}(X, Y), \mathrm{not}\ \mathit{wall}(X + 1, Y + 1),\\
c_6 &= \leftarrow \mathit{wall}(X, Y), \mathit{empty}(X + 1, Y), \mathit{empty}(X - 1, Y),\\
&\qquad \mathit{empty}(X, Y + 1), \mathit{empty}(X, Y - 1),
\end{aligned}
$$

implementing the yet uncovered parts of the specification by someone else. Constraint c_3 is similar to the corrected constraint c_2 but forbidding 2×2 blocks of walls instead of empty cells. Constraints c_4 and c_5 ensure that one common neighbour of two diagonally adjacent walls must be a wall. Consequently, the purpose of the remaining constraint c_6 must be to disallow walls to be completely surrounded by empty cells. We are puzzled, however, that the rule already forbids the case that only upper, lower, left, and right neighbours are empty. So, we are wondering how the case that a wall has a single adjacent wall that is the bottom right neighbour is addressed.

To shed light on this issue, we reuse S_4 as a breakpoint for stepping, where $\mathit{Int}[S_4] = I_4$ is illustrated in Fig. 2. We successively add instances of rule $\mathit{empty}(X, Y) \leftarrow \mathrm{not}\ \mathit{wall}(X, Y), \mathit{col}(X), \mathit{row}(Y)$ for fixing the six unassigned cells around the wall at $(3, 3)$ to be empty. Let us assume that S_5 is the state that results ($I_5 = \mathit{Int}[S_5]$ is also

depicted in Fig. 2). As the wall in the centre has as its only neighbouring wall the cell $(4, 4)$, we see the requirement that the wall may not be surrounded by empty cells is not violated. Checking for active rules at state S_5 reveals that constraint c_6 has active instances as expected. However, we notice that also constraint c_4 has an active instance under I_5. We now understand why the encoding is correct as the developer of constraints c_3 to c_6 has exploited the interplay of the requirements already that walls without wall neighbours are forbidden and that two diagonally adjacent walls must have one joint neighbour wall. Whenever a wall has only diagonally adjacent walls as neighbours, it is not harmful that constraint c_6 is violated: then, necessarily also the requirement of a common neighbour of the diagonally adjacent walls is violated.

5 Related Work

Previous work on visualising answer-set computations is realised by the `noMoRe`-system [10]. This system is graph-based and utilises rule dependency graphs (RDGs) which are directed labelled graphs where the nodes are the rules of a given program. Answer sets can be computed by stepwise colouring the nodes of the RDG of a ground program either green or red, reflecting whether a rule is considered active or not. An answer set is formed by the heads of the rules which are coloured green. A handicap for practical stepping is the separation of visualisation from the actual source code due to the graph-based representation and the limitation to ground programs.

Work on debugging in ASP includes *justifications* for ASP [11]. A justification is a labelled directed graph that explains the truth value of a literal with respect to an answer-set in terms of dependency on the truth values of fellow literals. Interesting with respect to our technique is the notion of an *online justification* that explains truth values with respect to partial answer sets emerging during the solving process. As our approach is compatible with the model of computation for online justifications, they can be used in a combined debugging approach. While interactively stepping through a computation allows for following individual intuitions concerning rule applications, justifications or related concepts could keep track of the chosen support for individual literals of interest. A potential shortcoming concerning the intuition of justifications is the absence of program rules, constituting the actual source code artifacts, in the graphs.

Other work on declarative debugging centred on the question why a given interpretation is not an answer set of a program [5]. The answers are given in terms of rule instances that are unsatisfied or loops that are unfounded. As noted in Section 4.2, the meta-programming techniques used in that work allow for identifying active rules in the stepping approach. Also, the interpretation needed as input in that debugging approach could be partially constructed by means of stepping. Note that unfounded loops cannot occur in the stepping process as states are required to be stable.

Another related approach is to trace the concrete execution of a solver. A respective system developed for DLV [12] is intended for debugging the solver itself rather than the answer-set programs. A disadvantage of solver-based tracing for debugging and program analysis is that some solver algorithms do not work on the rule level, are quite involved, and hard to grasp for an ordinary programmer. Interpreted as a strategy for computing answer sets, our stepping model is similar in spirit to a non-deterministic algorithm due

to Iwayama and Satoh [13]. Moreover, Gebser et al. [14] introduced an incremental semantics for logic programs based on ι-answer sets. These are relaxations of answer sets that are not necessarily models of the overall program and can be constructed by step-by-step applying active rules similar as in our approach. Their semantics guarantees to reach a ι-answer set, while under standard semantics, computations may fail.

6 Conclusion

We presented a framework for stepping through an answer-set program that is useful for debugging and program analysis. It allows the programmer to follow his or her intuitions regarding which rules to apply next and is based on an intuitive and simple computation model where rules are subsequently added to a state. Every state implicitly defines an interpretation that is stable with respect to that state. We also discussed how to obtain states that may serve as breakpoints from which stepping is started. Keeping a handful of these breakpoints during program development, the programmer can quickly initiate stepping sessions from situations he or she is already familiar with. A prototypical stepping system will be part of SeaLion, an integrated development environment for ASP we are currently developing. In future work, we plan to extend the approach to programs with function symbols, aggregates (possibly in rule heads), and disjunctions.

References

1. Gelfond, M., Leone, N.: Logic programming and knowledge representation - the A-Prolog perspective. Artificial Intelligence **138**(1-2) (2002) 3–38
2. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proc. LPNMR'07. (2007) 31–43
3. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proc. NMR'06. (2006) 77–83
4. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proc. AAAI'08, AAAI Press (2008) 448–453
5. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4–6) (2010) 513–529
6. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Proc. JELIA'04. (2004) 200–212
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9**(3-4) (1991) 365–386
8. Shapiro, E.Y.: Algorithmic Program Debugging. PhD thesis, Yale University, New Haven, CT, USA (May 1982)
9. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In: Proc. LPNMR'09. Volume 5753. (2009) 637–654
10. Bösel, A., Linke, T., Schaub, T.: Profiling answer set programming: The visualization component of the noMoRe system. In: Proc. JELIA'04. (2004) 702–705
11. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1) (2009) 1–56
12. Calimeri, F., Leone, N., Ricca, F., Veltri, P.: A visual tracer for DLV. In: Proc. SEA'09. (2009)
13. Iwayama, N., Satoh, K.: Computing abduction by using TMS with top-down expectation. Journal of Logic Programming **44**(1-3) (2000) 179 – 206
14. Gebser, M., Gharib, M., Mercer, R.E., Schaub, T.: Monotonic answer set programming. Journal of Logic and Computation **19**(4) (2009) 539–564