



Evolog - Actions and Modularization in Lazy-Grounding Answer Set Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Michael Langowski, BSc.

Matrikelnummer 01426581

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Dr. Thomas Eiter

Mitwirkung: Dr. Antonius Weinzierl

Wien, 1. Juli 2022

Michael Langowski

Thomas Eiter



Evolog - Actions and Modularization in Lazy-Grounding Answer Set Programming

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Michael Langowski, BSc.

Registration Number 01426581

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Dr. Thomas Eiter

Assistance: Dr. Antonius Weinzierl

Vienna, 1st July, 2022

Michael Langowski

Thomas Eiter

Erklärung zur Verfassung der Arbeit

Michael Langowski, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juli 2022

Michael Langowski

Danksagung

Ihr Text hier.

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

Enter your text here.

Contents

Kurzfassung	xi
Abstract	xiii
Contents	xv
1 Introduction	1
2 Preliminaries	3
2.1 Answer Set Programming	3
2.2 Lazy-Grounding ASP Solving	7
3 The Evolog Language	13
3.1 Actions in Evolog	13
3.2 Program Modularization in Evolog	18
3.3 Relationship between Evolog- and Stable Model Semantics	18
4 Evolog Reference Implementation	21
List of Figures	23
List of Tables	25
List of Algorithms	27
Acronyms	29
Bibliography	31

Introduction

short summary
what ASP is, why
it is cool etc.

The goal of this work is to address the following issues with as little "semantics overhead" as possible and deliver an easy-to-use programming language that can be used in most software engineering contexts.

Triggering actions from programs Most program flows follow a chain of events, each a consequence of its predecessor, e.g. "If there exists a file A, read it. If reading was successful, do something with the content. If the operation succeeds, write the result to file B". It is highly desirable to be able to write this kind of program in a declarative, logic-based language that can leverage the strengths of ASP for the "business logic" part. Specifically, the proposed action semantics should deliver

- declarative programs, i.e. order in which actions occur in code does not affect semantics,
- actions behaving in a functional fashion, i.e. an action always gives the same result for the same input. Especially, actions have to be idempotent in the sense that, for an ASP rule that is associated with some action, the result of the action never changes, no matter how often the rule fires.
- transparent action execution, i.e. every action that is executed during evaluation of a program must be reflected in an answer set.

Program Modularization While not formally connected, triggering actions from programs and modularization (i.e. plugable and re-usable sub-programs), intuitively complement each other in our current high-level design. Introducing a simple, easy-to-use module system is therefore the second goal of this work. It is, however, secondary in priority to definition and prototypical implementation of action support and may be reduced to a technical design draft if required due to time constraints.

Incremental Evaluation and Lazy Grounding Experiences from existing systems for ASP application development such as ASAP [SW15] or ACTHEX [BEFI10] show that, in order to achieve the evaluation performance necessary for use in real-world applications, ASP application code needs to be evaluated in an incremental fashion (rather than iteratively re-evaluating the whole program) whenever possible. The lazy-grounding architecture employed by ASP systems such as Alpha [WBB⁺19] offers an intuitive solution.

Preliminaries

2.1 Answer Set Programming

When speaking of Answer Set Programming (ASP), we nowadays mostly refer to the language specified by the ASP-Core2 standard [CFG⁺20]. It uses the *stable model semantics* by Gelfond and Lifschitz [GL88] as a formal basis and enhances it with support for advanced concepts such as disjunctive programs, aggregate literals and weak constraints. This chapter describes the input language supported by the Alpha solver, which will serve as the basis on which we will define the Evolog language.

2.1.1 Syntax

Definition 2.1.1 (Integer numeral). An *integer numeral* in the context of an ASP program is a string matching the regular expression:

$(-)?[0-9]^+$

The set of all valid integer numerals is denoted as *INT*.

Definition 2.1.2 (Identifier). An *identifier* in the context of an ASP program is a string matching the regular expression:

$[a-z][a-zA-Z0-9_]*$

The set of all valid identifiers is denoted as *ID*.

Definition 2.1.3 (Variable Name). A *variable name* in the context of an ASP program is a string matching the regular expression:

$[A-Z][a-zA-Z0-9_]*$

The set of all valid variable names is denoted as *VAR*.

Definition 2.1.4 (Term). A *term* is inductively defined as follows:

- Any *constant* $c \in (INT \cup ID)$ is a term.
- Any *variable* $v \in VAR$ is a term.
- Given terms t_1, t_2 , any *arithmetical expression* $t_1 \oplus t_2$ with $\oplus \in \{+, -, *, /, **\}$ is a term.
- Given terms t_1, t_2 , any *interval expression* $t_1 \dots t_2$ is a term.
- For function symbol $f \in ID$ and argument terms t_1, \dots, t_n , the *functional expression* $f(t_1, \dots, t_n)$ is a term.

Definition 2.1.5 (Subterms). Given a term t , the set of *subterms* of t , $st(t)$, is defined as follows:

- If t is a *constant* or *variable*, $st(t) = \{t\}$.
- If t is an *arithmetical expression* $t_1 \oplus t_2$, $st(t) = st(t_1) \cup st(t_2)$.
- If t is an *interval expression* $t_1 \dots t_2$, $st(t) = st(t_1) \cup st(t_2)$.
- If t is a *functional expression* with argument terms t_1, \dots, t_n , $st(t) = st(t_1) \cup \dots \cup st(t_n)$.

A term is called *ground* if it is variable-free, i.e. none of its subterms is a variable.

Definition 2.1.6 (Basic Atom). Given a predicate symbol $p \in ID$ and argument terms t_1, \dots, t_n , the expression

$$p(t_1, \dots, t_n)$$

is called a *atom*. An atom is ground if all of its argument terms are ground. A ground atom with predicate p is called an *instance* of p .

Definition 2.1.7 (Comparison Atom). Given terms t_1 and t_2 and comparison operator \odot where $\odot \in \{<, \leq, =, \geq, >, \neq\}$, the expression

$$t_1 \odot t_2$$

is called a *comparison atom*. Syntactically, a comparison atom is a regular atom where the predicate symbol (i.e. comparison operator) is written in infix- rather than prefix-notation.

Definition 2.1.8 (External Atom). Given an *external predicate name* ext , *input terms* t_1, \dots, t_n and *output terms* t_{n+1}, \dots, t_m , the expression

$$@ext[t_1, \dots, t_n](t_{n+1}, \dots, t_m)$$

is called an *external atom*. Syntactically, external atoms are regular atoms where $@ext$ is the predicate symbol and t_1, \dots, t_m are argument terms.

Definition 2.1.9 (Literal). A literal in ASP is an atom a or ("default"-)negated atom $\text{not } a$. Literals wrapping comparison- or external atoms are called *fixed interpretation literals*.

Definition 2.1.10 (Rule, Program). A *rule* is an expression of form

$$a_H \leftarrow b_1, \dots, b_n.$$

for $n \geq 0$, where the *rule head* a_H is an atom and the *rule body* b_1, \dots, b_n is a set of literals. An ASP *program* is a set of rules. A rule with an empty body is called a *fact*. A rule is *ground* if both its head atom and all of its body literals are ground. By the same reasoning, a program is ground if all of its rules are ground.

Given a rule r , we refer to the head of r as $h(r)$ and the body of r as $b(r)$. Furthermore, $b^+(r)$ is used to reference the set of *positive body literals* of r , while $b^-(r)$ references the *negative body literals*.

Definition 2.1.11 (Constraint). A *constraint* is a special form of rule, written as a rule with an empty head, i.e.

$$\leftarrow b_1, \dots, b_n.$$

It is syntactic sugar for

$$q \leftarrow b_1, \dots, b_n, \text{not } q.$$

where q is a propositional constant not occurring in any other rule in the program.

2.1.2 Semantics

Definition 2.1.12 (Herbrand Universe). The Herbrand Universe HU_P of a Program P is the set of all valid terms that can be constructed with respect to Definitions 2.1.1, 2.1.2 and 2.1.4. Note that most papers use stricter definitions of the Herbrand Universe where HU_P consists only of terms constructible from constants occurring in P . The broader definition used here is chosen for ease of definition with respect to some of the extensions introduced in Section 3.1.

Definition 2.1.13 (Herbrand Base). The Herbrand Base HB_P of a Program P is the set of all ground atoms that can be constructed from the Herbrand Universe HU_P according to definition 2.1.6.

Definition 2.1.14 (Herbrand Interpretation). A Herbrand Interpretation is a special form of first order interpretation where the domain of the interpretation is a Herbrand Universe and terms are the interpretation of a term is the term itself, i.e. the corresponding element of HU_P . Intuitively, Herbrand Interpretations constitute listings of atoms that are true in a given program. Since the domain of a Herbrand Interpretation is always the Herbrand Universe HU_P , we only need to give a predicate interpretation for the predicates occurring in a program P in order to fully specify a Herbrand Interpretation. We can therefore denote Herbrand Interpretations as sets of atoms $I \subseteq HB_P$.

Grounding

Given a program P containing variables, *grounding* refers to the process of converting P into a semantically equivalent propositional, i.e. variable-free, program.

Definition 2.1.15 (Substitution, adapted from [Wei17]). A substitution $\sigma : VAR \mapsto (ID \cup INT)$ is a mapping from variables to constants. For a atom a , applying a substitution results in a substituted atom $a\sigma$ in which variables are replaced according to σ . Substitutions are applied to rules by applying them to every individual atom or literal within the rule. By the same mechanism, we can apply substitutions to programs by applying the to all rules.

Definition 2.1.16 (Grounding). Given a rule r , the *grounding* of r , $grnd(r)$, is a set of substitutions S , such that the set of ground rules resulting from applying the substitutions in S is semantically equivalent to r . In a slight abuse of terminology, *grounding* in this work also refers to the set of ground rules resulting from applying S as well as the process of finding said set.

Stable Model Semantics

Definition 2.1.17 (Fixed interpretation literals). Fixed interpretation literals, i.e. comparison- and external literals, respectively, are interpreted by means of a program-independent oracle function $f_O : H_U(P)^* \mapsto \{\top, \perp\}$, i.e. a fixed interpretation literal with argument terms t_1, \dots, t_n has the same truth value under all interpretations.

Definition 2.1.18 (Truth of Atoms and Literals). A positive ground literal l with atom a is true w.r.t. a Herbrand Interpretation I , i.e. $I \models l$ if

- a is a basic atom contained in I , i.e. $a \in I$,
- a is a fixed interpretation literal with terms t_1, \dots, t_n and $f_O(t_1, \dots, t_n) = \top$.

For a negative ground literal $not\ a$, the reverse holds, i.e. $I \models not\ a$ if

- a is a basic atom not contained in I , i.e. $a \notin I$,
- a is a fixed interpretation literal with terms t_1, \dots, t_n and $f_O(t_1, \dots, t_n) = \perp$.

A set of literals L is true w.r.t. an interpretation I if $I \models l$ holds for every literal $l \in L$.

Definition 2.1.19 (Positive Logic Program). A *positive* logic program is a program according to Definition 2.1.10, where all rule bodies are positive, i.e. no rule body contains a negated atom.

Definition 2.1.20 (Immediate Consequence Operator, adapted from [EIK09]). Given a Herbrand Interpretation I and a ground positive logic program P , the immediate consequence operator $T_P(I)$ defines a monotonic function $T_P : 2^{HB_P} \mapsto 2^{HB_P}$ such that

$$T_P(I) = \{h(r) \mid r \in P \wedge I \models b(r)\}$$

i.e. the result set of applying T_P with a Herbrand Interpretation I contains the heads of all rules whose body is true under I .

Definition 2.1.21 (Least Model of positive logic programs). The least model $LM(P)$ of a (ground) positive logic program P is the least fixpoint of the T_P operator of P , i.e. the set toward which the sequence $\langle T_P^i \rangle$, with $i \geq 0$, $T_P^0 = \emptyset$ and $T_P^i = T_P(T_P^{i-1})$ for $i \geq 1$, converges. The existence of said fixpoint and its characterisation as limit of $\langle T_P^i \rangle$ follow from the fixpoint theorems of Knaster, Tarski and Kleene, respectively.

Definition 2.1.22 (Gelfond-Lifschitz Reduct, adapted from [GL88] and [EIK09]). Given a ground ASP program P and a Herbrand Interpretation I , the *Gelfond-Lifschitz-Reduct* ("GL-reduct") P^I of P with respect to I is the program obtained by:

- removing from P all rules r that are "blocked", i.e. $I \not\models l$ for some literal $l \in b^-(r)$
- and removing the negative body of all other rules.

Note that P^I is a positive logic program.

Definition 2.1.23 (Answer Set [GL88] [EIK09]). A Herbrand Interpretation I of an ASP program P is an *answer set* or *stable model* of P iff it is the least model $LM(P^I)$ of the GL-reduct P^I of P . We denote the set of Answer Sets of a program P as $AS(P)$.

2.2 Lazy-Grounding ASP Solving

2.2.1 Two-phased ASP solving

In traditional ASP solving systems such as SModels [SN01], DLV [LPF⁺02] or Clingo [GKK⁺08], grounding an input program and solving the resulting propositional program are distinct sequential steps in the overall solving process. Consequently, in order to obtain answer sets of a program, one has to calculate a grounding for the entire program first, and can only then start the actual solver. Since the grounding of an arbitrary program may be exponentially larger than the nonground program or, in some extreme cases, not even finite, calculating a full grounding is often not feasible, especially for programs where only very few ground rules can actually fire. Lazy-grounding systems like Alpha try to alleviate this by interleaving the grounding- and solving steps and ideally ground only as much of the input programs as is necessary to find all answer sets.

2.2.2 Conceptual solving workflow in Alpha

The formal basis of lazy-grounding architectures lies in the notion of a *computation sequence*, i.e. a set of rules firing in a given order in order to get to an interpretation that is an answer set. Definition 2.2.1 formally introduces computation sequences.

Definition 2.2.1 (Computation Sequence, adapted from [Wei17] and [LN09]). Let P be an ASP program and $S = (A_0, \dots, A_\infty)$ a sequence of assignments, i.e. herbrand interpretations denoted by a set of atoms assumed to be true, then S is called a *computation sequence* iff

- $A_0 = \emptyset$
- $\forall i \geq 1 : A_i \subseteq T_P(A_{i-1})$, i.e. every A_i is a consequence of its predecessor in the sequence,
- $\forall i \geq 1 : A_{i-1} \subseteq A_i$, i.e. S is monotonic,
- $A_\infty = \bigcup_{i=0}^{\infty} A_i = T_P(A_\infty)$, i.e. S converges toward a fixpoint and
- $\forall i \geq 1 : \forall a \in A_i \setminus A_{i-1}, \exists r \in P : h(r) = a \wedge \forall j \geq i - 1 : A_j \models a$, i.e. applicability of rules is persistent.

A_∞ is an answer set of P iff S is a computation sequence. Note that there may exist an arbitrary number of computation sequences leading to the same answer set.

Obviously, computation sequences can be easily found by simple iterative application of the T_P operator for programs that do not use negation in rules. However, since in general once negation comes into play, solvers may have to retract assignments of atoms ("backtrack"), over the course of the solving process. Lazy-grounding solvers suffer from a performance penalty compared to two-phased systems in that respect. This penalty results from algorithms based on Conflict-driven Nogood Learning (CDNL) [GKS12] achieving higher performance since conflicts occur faster and more nogoods can be learned from them with a full grounding available. A key challenge in designing lazy-grounding systems is therefore identifying classes of programs as well as groups of rules within programs that can be evaluated using simplified deterministic algorithms in order to minimize the number of potential backtracks.

Example of mutually blocking rules!

Structural Dependency Analysis and Stratified Evaluation

Definition 2.2.2 (Unification). Let l_1, l_2 be literals. Then a substitution σ is a *unifier* of l_1 and l_2 iff $l_1\sigma = l_2\sigma$. Two literals for which a unifier exists are said to be *unifiable*. We use the notation $l_1 \uplus l_2$ to express that l_1 and l_2 are unifiable.

Since we're talking about non-ground stratification, should we define a nonground T_P operator! Just simply say we add the rules where all defining rules have already fired in each step

Definition 2.2.3 (Defining rules). Given an ASP program P and literal l of form a or $\text{not } a$ with atom a , the set $\text{def}(l)$ of *defining rules* of l is defined as

$$\text{def}(l) = \{r \mid r \in P \wedge h(r) \uplus a\}$$

i.e. all rules in P whose head is unifiable with a .

Definition 2.2.4 (Stratification, adapted from [ABW88]). Given a (non-ground) ASP program P , a stratification is a partition S of P into sub-programs called *strata* (P_0, \dots, P_1) such that

- $\cup_{i=0}^n P_i = P$, i.e. S is total,
- $\forall i \geq 0 : \forall r \in P_i : \forall l \in b^+(r) : \text{def}(l) \subseteq \cup_{j=0}^i P_j$, i. e. for every positive body literal l of every rule r , it holds that all rules defining l reside in a stratum with lower or equal index to the stratum r resides in, and
- $\forall i \geq 0 : \forall r \in P_i : \forall l \in b^-(r) : \text{def}(l) \subseteq \cup_{j=0}^{i-1} P_j$, i. e. for every negative body literal l of every rule r , it holds that all rules defining l reside in a stratum with strictly lower index than the stratum r resides in.

A program is called *stratified* iff a stratification exists for it.

Definition 2.2.5 (Stratified Evaluation, adapted from [EIK09] and [ABW88]). Let P be an ASP program, $S = (P_0, \dots, P_n)$ a stratification of P and T_{P_i} with $0 \leq i \leq n$ the immediate consequence operator for sub-program $P_i \in S$ respectively. Then the least model $LM(P)$ of P is defined as follows. The sequence $\langle M_{P_i} \rangle$, $0 \leq i \leq n$ with $M_{P_0} = \text{lfp}(T_{P_0})$ and $M_{P_i} = \text{lfp}(T_{P_i \cup M_{S_{i-1}}})$ for all $1 \leq i \leq n$ defines the least model for each stratum. The least model $LM(P)$ of program P is then the least model of the highest stratum M_{P_n} , i.e. the end of the sequence $\langle M_{P_i} \rangle$.

Definition 2.2.6 (Dependencies). Let P be an ASP program and $r \in P$ a rule contained in P . a rule d is a *positive dependency* of r , i.e. $r \succ_d^+ d$ iff one the following holds:

- $\exists l \in b^+(r) : d \in \text{def}(l)$, i.e. d is a defining rule for some positive body literal of r , or
- $\exists d_1 \in P : r \succ_d^+ d_1 \wedge d_1 \succ_d^+ d$, i.e. there is some positive dependency d_1 of r of which d is a (transitive) positive dependency.

Negative dependencies are defined in the same way, i.e. a rule d is a negative dependency of r , $r \succ_d^- d$ iff

- $\exists l \in b^-(r) : d \in \text{def}(l)$, i.e. d is a defining rule for some negative body literal of r , or
- $\exists d_1 \in P : r \succ_d^- d_1 \wedge d_1 \succ_d^- d$, i.e. there is some negative dependency d_1 of r of which d is a (transitive) negative dependency.

Any positive or negative dependency d of r is a *dependency* of r , i.e. $r \succ_d d$. We denote the set of dependencies of a rule as $D(r) = \{d \mid r \succ_d d\}$ and positive and negative dependencies as $D^+(r) = \{d \mid r \succ_d^+ d\}$ and $D^-(r) = \{d \mid r \succ_d^- d\}$ respectively. Any non-transitive dependency of a rule is called a *direct dependency*.

Definition 2.2.7 (Dependency Graph). Let P be an ASP program. Then the *dependency graph* $DG_P = (R, D)$ is a directed graph with vertex set R , which has one element for each rule in P , and (dependency-)edge set D such that

$$D = \{(r_1, r_2, +) \mid r_1, r_2 \in P \wedge r_2 \text{ is direct positive dependency of } r_1\} \\ \cup \{(r_1, r_2, -) \mid r_1, r_2 \in P \wedge r_2 \text{ is direct negative dependency of } r_1\}$$

Edges are represented as 3-tuples where the first two values represent target and destination vertices and the third value indicates the "polarity", i. e. positive ("+") or negative ("-"), of the dependency.

Definition 2.2.8 (Component Graph). The component graph $CG_P = (C, D)$ of a program P is defined as the "condensed" dependency graph, i. e. vertices of CG_P represent strongly connected components of DG_P . Each vertex of CG_P is labelled "+" if the respective strongly connected component is connected only by positive edges, or "-" if there is a negative edge in the component. Edges of CG_P are those edges of DG_P that connect vertices from different strongly connected components where double edges resulting from multiple rule-level dependencies of same polarity between components are condensed into single edges.

Definition 2.2.9 (Splitting Set, adapted from [LT94]). Given a program P , a set of atoms U is a *splitting set* of P if for every rule r , where $h(r) \in U$, also the atoms corresponding to all body literals of r are in U . The set of rules corresponding to U , i.e. the rules defining the atoms in U , is called *bottom* of P with respect to U , denoted as $B_U(P)$. Consequently, $P \setminus B_U(P)$ is called *top* of P , which is denoted as $T_U(P)$.

Definition 2.2.10 (Common Base Program, adapted from [Lan19]). Given a splitting set S of program P , the bottom $B_S(P)$ is called *common base program*, i.e. $CBP(P)$ if it is stratified and maximal in the sense that adding any further rule to $B_S(P)$ would destroy the property of $B_S(P)$ of being stratified.

Intuitively, a set of rule heads is a splitting set if for every rule head in the set, all rule heads on which it depends are in the set as well. The importance of the notion of a splitting set lies in the fact that answer set computation can be split up using splitting sets: Given a splitting set, we can first calculate all answer sets of the bottom, and then solve the top with respect to each of those answer sets [LT94].

Alpha's evaluation logic makes use of this by evaluating first the maximum stratified bottom (*common base program*, see Definition 2.2.10) using the simplified bottom-up algorithm outlined in Algorithm 2.1 and then only using the - computationally more complex - CDNL-based algorithm for the top part.

Algorithm 2.1: Procedure *evaluateCommonBaseProgram*

Input : Stratification S
Input : Program P
Output : partially evaluated program P_{eval}

```
1 Facts  $F = facts(P)$ ;  
2  $n \leftarrow (|S| - 1)$ ;  
3 foreach  $i$  in  $0 \dots n$  do  
4   Program  $grnd(P_{S_i}) = ground(P_{S_i})$ ;  
5   Facts  $F_{old} = \emptyset$ ;  
6   do  
7      $F_{old} = F$ ;  
8      $F \leftarrow pr(T_{grnd(P_{S_i})}(F))$ ;  
9   while  $F_{old} \neq F$ ;  
10   $P \leftarrow P \setminus P_{S_i}$ ;  
11 end  
12 return  $P_{eval} = P \cup F$ ;
```

The Evolog Language

The Evolog language extends (non-disjunctive) ASP as defined in the ASP-Core2 standard [CFG⁺20] with facilities to communicate with and influence the "outside world" (e.g. read and write files, capture user input, etc.) as well as program modularization and reusability features, namely *actions* and *modules*.

3.1 Actions in Evolog

Actions allow for an ASP program to encode operations with *side-effects* while maintaining fully declarative semantics. Actions are modelled in a functional style loosely based on the concept of monads as used in Haskell . Intuitively, to maintain declarative semantics, actions need to behave as pure functions, meaning the result of executing an action (i.e. evaluating the respective function) must be reproducible for each input value across all executions. On first glance, this seems to contradict the nature of IO operations, which inherently depend on some state, e.g. the result of evaluating a function *getFileHandle(f)* for a file *f* will be different depending on whether *f* exists, is readable, etc. However, at any given point in time - in other words, in a given state of the world - the operation will have exactly one result (i.e. a file handle or an error will be returned). A possible solution to making state-dependent operations behave as functions is therefore to make the state of the world at the time of evaluation part of the function's input. A function *f(x)* is then turned into *f'(s, x)* where *s* represents a specific world state. The rest of this section deals with formalizing this notion of actions.

cite something here!

3.1.1 Syntax

Definition 3.1.1 (Action Rule, Action Program). An *action rule* *R* is of form

$$a_H : @t_{act} = act_{res} \leftarrow l_1, \dots, l_n.$$

where

Define non-disjunctive ASP-Core2 in detail in preliminaries. Give detailed definition of all "standard ASP" elements referenced here!

- a_H is an atom called *head atom*,
- t_{act} is a functional term called *action term*,
- act_{res} is a term called *(action-)result term*
- and l_1, \dots, l_n are literals constituting the *body* of R .

An *action program* P is a set of (classic ASP-)rules and action rules.

3.1.2 Semantics

To properly define the semantics of an action program according to the intuition outlined at the start of this section, we first need to formalize our view of the "outside world" which action rules interact with. We call the world in which we execute a program a *frame* - formally, action programs are always evaluated *with respect to a given frame*. The behavior of actions is specified in terms of *action functions*. The semantics (i.e. interpretations) of action functions in a program are defined by the respective frame.

Action Rule Expansion

To get from the practical-minded action syntax from Definition 3.1.1 to the formal representation of an action as a function of some state and an input, we use the helper construct of an action rule's *expansion* to bridge the gap. Intuitively, the expansion of an action rule is a syntactic transformation that results in a more verbose version of the original rule called *application rule* and a second rule only dependent on the application rule called *projection rule*. A (ground) application rule's head atom uniquely identifies the ground instance of the rule that derived it. As one such atom corresponds to one action executed, we call a ground instance of an application rule head in an answer set an *action witness*.

Definition 3.1.2 (Action Rule Expansion). Given a non-ground action rule R with head atom a_H , action term $f_{act}(i_1, \dots, i_n)$ and body B consisting of literals l_1, \dots, l_m , the expansion of R is a pair of rules consisting of an *application rule* R_{app} and *projection rule* R_{proj} . R_{app} is defined as

$$a_{res}(f_{act}, S, I, f_{act}(S, I)) \leftarrow l_1, \dots, l_n.$$

where S and I and function terms called *state-* and *input-*terms, respectively. An action rule's state term has the function symbol *state* and terms $fn(l_1), \dots, fn(l_m)$, with the expression $fn(l)$ for a literal l denoting a function term representing l . The (function-)term representation of a literal $p(t_1, \dots, t_n)$ with predicate symbol p and terms t_1, \dots, t_n uses p as function symbol. For a negated literal *not* $p(t_1, \dots, t_n)$, the representing function term is *not*($p(t_1, \dots, p_n)$). The action input term is a "wrapped" version of all arguments of the action term, i.e. for action term $f_{act}(t_1, \dots, t_n)$, the corresponding input term is

define (classic ASP) grounding and substitutions in preliminaries

$input(t_1, \dots, t_n)$. The term $f_{act}(S, I)$ is called *action application term*. The projection rule R_{proj} is defined as

$$a_H \leftarrow a_{res}(f_{act}, S, I, v_{res}).$$

where a_H is the head atom of the initial action rule R and the (sole) body atom is the action witness derived by R_{app} , with the application term $f_{act}(S, I)$ replaced by a variable v_{res} called *action result variable*.

Looking at the head of an action application rule of format $a_{res}(f_{act}, S, I, t_{app})$ with action f_{act} , state term S , input term I and application term t_{app} , the intuitive reading of this atom is "The result of action function f_{act} applied to state S and input I is t_{app} ", i.e. the action application term t_{app} is not a regular (uninterpreted) function term as in regular ASP, but an actual function call which is resolved using an interpretation function provided by a *frame* during grounding.

Grounding of Action Rules

Grounding, in the context of answer set programming, generally refers to the conversion of a program with variables into a semantically equivalent, variable-free, version. Action application terms as introduced in Definition 3.1.2 can be intuitively read as variables, in the sense that they represent the result of applying the respective action function. Consequently, all action application terms are replaced with the respective (ground) result terms defined in the *frame* with respect to which the program is grounded.

Definition 3.1.3 (Frame). Given an action program P containing action application terms $A = \{a_1, \dots, a_n\}$, a frame F is an interpretation function such that, for each application term $f_{act}(S, I) \in A$ where $S \in H_U(P)^*$ and $I \in H_U(P)^*$, $F(f_{act}) : H_U(P)^* \times H_U(P)^* \mapsto H_U(P)$.

Definition 3.1.4 (Grounding of action rules). Grounding of Evolog rules (and programs) always happens *with respect to a frame*. Given a frame F , an expanded action rule r_a and a (grounding) substitution σ over all body variables of application rule $r_{a_{app}}$, during grounding, every ground action application term $t_{app}\sigma$ resulting from applying substitution σ is replaced with its interpretation according to F .

Example 3.1.1 demonstrates the expansion of an action rule as well as a compatible example frame for the respective action.

Example 3.1.1 (Expansion and Frame). Consider following Evolog Program P which contains an action rule with action a :

$$\begin{aligned} & p(a). \ q(b). \ r(c). \\ & h(X, R) : @a(X, Z) = R \leftarrow p(X), q(Y), r(Z). \end{aligned}$$

The expansion of R is:

$$\begin{aligned} a_{res}(a, state(p(X), q(Y), r(Z)), input(X, Z), a(state(p(X), q(Y), r(Z)), input(X, Z))) &\leftarrow \\ &p(X), q(Y), r(Z). \\ h(X, R) &\leftarrow a_{res}(a, state(p(X), q(Y), r(Z)), input(X, Z), R). \end{aligned}$$

Furthermore, consider following frame F :

$$F(a) = \{a(state(p(a), q(b), r(c)), input(a, c)) \mapsto success(a, c)\}$$

which assigns the result $success(a, c)$ to the action application term (i.e. function call $a(state(p(a), q(b), r(c)), input(a, c))$).

Then, the ground program P_{grnd} after action rule expansion is

$$\begin{aligned} &p(a). q(b). r(c). \\ a_{res}(a, state(p(a), q(b), r(c)), input(a, c), success(a, c)) &\leftarrow p(a). q(b). r(c). \\ h(a, success(a, c)) &\leftarrow a_{res}(a, state(p(a), q(b), r(c)), input(a, c), success(a, c)). \end{aligned}$$

The sole model of P with respect to frame F is

$$\begin{aligned} M = \{ &p(a), q(b), r(c), \\ &a_{res}(a, state(p(a), q(b), r(c)), input(a, c), success(a, c)) \\ &h(a, success(a, c)) \} \end{aligned}$$

Evolog Models

Having introduced action rule expansions as well as frames, we now use these to extend the stable model semantics to Evolog programs.

Definition 3.1.5 (Supportedness of Actions). Let r_{app} be a non-ground action application rule with head $H = a_{res}(f_{act}, S, I, f_{act}(S, I))$, F a frame, and $H_{grnd} = a_{res}(f_{act}, S_{grnd}, I_{grnd}, r)$ a ground instance of H with r being an arbitrary ground term. Then, H_{grnd} is *supported by* F , if and only if F contains a mapping of form $f_{act}(S_{grnd}, I_{grnd}) \mapsto r$, i.e. r is a valid result of action function f_{act} with arguments S_{grnd} and I_{grnd} according to frame F . We call a ground instance of an action rule *supported by a frame* if the head of the corresponding application rule in the rule's expansion is supported by that frame.

Definition 3.1.6 (Evolog-Reduct). Given a ground Evolog program P , a frame F and a set of ground atoms A , the *Evolog Reduct* of P with respect to F and A P_F^A is obtained from P as follows:

1. Remove all rules r from P that are "blocked", i.e. $A \not\models l$ for some negative body literal $l \in b^-(r)$.

2. Remove all action application rules from P which are not supported by F .
3. Remove the negative body from all other rules.

Note that the reduct outlined in Definition 3.1.6 extends the classic GL-reduct (see Definition 2.1.22) just by adding a check on action supportedness.

Definition 3.1.7 (Evolog Model). A herbrand interpretation I of an Evolog Program is an *Evolog Model* ("answer set") of an Evolog program P with respect to a frame F if and only if it is a minimal classical model of its Evolog-Reduct P_F^A . We denote the set of Evolog Models of a program P as $EM(P)$.

3.1.3 Restrictions on Program Structure

While the action semantics outlined so far addresses the requirements for both declarativity and functionality of actions outlined in the introduction (see 1), we haven't yet addressed the demand for transparency, i.e. that every action that is executed must be reflected in an answer set of the respective program. With just the semantics outlined in Section 3.1.2, it would be possible to write programs such as the one shown in Example 3.1.2 where an action rule can fire, but the program is unsatisfiable.

Example 3.1.2 (Unsatisfiable Program with Side-effects). The program below contains an action rule that can fire (because $p(a)$ is true), but is also unsatisfiable due to the constraint in the last line.

$$\begin{aligned} & p(a). \\ & q(X) \leftarrow p(X). \\ & act_done(X, R) : @act(X) = R \leftarrow p(X). \\ & \leftarrow q(X), act_done(X, _). \end{aligned}$$

This kind of programs raises some hard problems for implementations - given the contract that every side-effect of (i.e. action executed by) a program must be reported in an answer set, a solver evaluating the program from Example 3.1.2 would have to "retract" action $act/1$ after finding that the program is unsatisfiable. Since it is generally not possible to "take back" side-effects (e.g. when some message is sent over a network broadcast to an unknown set of recipients), the only practical way to deal with this is to impose some conditions on programs with actions. Definition 3.1.8 details this notion and introduces *transparency* as a necessary condition for an Evolog program to be considered valid.

Definition 3.1.8 (Action Transparency). An action rule r_a of an Evolog program P is *transparent* if, for every expanded ground instance gr_a it holds that, if gr_a fires, then the head $h(gr_{a_{app}})$ of the respective application rule $gr_{a_{app}}$ is contained in an answer set of P .

For an Evolog program to be valid, all its action rules must be transparent.

It follows from Definition 3.1.8 that only satisfiable programs can be transparent. Furthermore, note that Definition 3.1.8 aims to be as permissive as possible in terms of program structure. In general, it can not be assumed that transparency of all rules in a program can be guaranteed up-front. Implementations may therefore impose further restrictions on what is considered a valid Evolog program.

It might make sense to introduce the restriction $P = \text{CBP}(P)$ for a valid program already here (anything more permissive cannot be linted without a theorem prover!)

3.2 Program Modularization in Evolog

Modules aim to introduce a tool to ASP that helps with code reusability and testability.

Conceptually, an Evolog Module is a special kind of external atom (see 2.1.8) that refers to an ASP solver being called with some program and input.

3.2.1 Syntax

TODO The following design problems need solving:

- If a complete answer set of a module is packed into one function term, how do you parse it (List iteration)? It probably makes the most sense to just introduce list terms and be done with it...
- What's the best syntax to express whether we want just one, a certain number of, or all answer sets?

Definition 3.2.1 (Module Literal). A *module literal* has the following form:

$$\rightarrow \text{modname}[i_1, \dots, i_n]([o_1, \dots, o_n]).$$

3.3 Relationship between Evolog- and Stable Model Semantics

The extensions to the usual ASP programming language described in the previous sections extend the original formalism with a notion of an outside world (through frames in the context of which, actions can be expressed) as well as a grouping mechanism for sub-programs into modules.

Theorem 3.3.1 (Extension). Every ASP program P is a valid and semantically equivalent Evolog program in the sense that - for any given frame F , the Evolog Models of P are the same as its Answer Sets according to Stable Model Semantics 2.1.23.

Proof. First, we assert that every "regular" ASP program P is also a syntactically valid Evolog program. This follows directly from the definitions - since Evolog only adds syntactic support for actions 3.1.1, but does not restrict regular ASP syntax, it follows that a syntactically correct ASP program is also a syntactically correct Evolog program.

Next, we show that for every regular ASP program P and any frame F , the Evolog Models of P are the same as its Stable Models, i.e. $EM(P) = AS(P)$:

Given any set of ground Atoms A from the Herbrand Base HB_P of P , P is an Answer Set according to Stable Model Semantics if it is a minimal model of the GL-reduct 2.1.22 P^A of P w.r.t. A . We recall that P^A is constructed as follows (see 2.1.22):

- remove from P all rules r that are "blocked", i.e. $A \not\models l$ for some literal $l \in b^-(r)$
- and remove the negative body of all other rules.

Furthermore, consider how the Evolog-Reduct of P w.r.t. A and (any arbitrary) Frame F , P_F^A , is constructed (see 3.1.6):

- Remove all rules r from P that are "blocked", i.e. $A \not\models l$ for some negative body literal $l \in b^-(r)$.
- Remove all action application rules from P which are not supported by F .
- Remove the negative body from all other rules.

Since P is a "regular", i.e. non-Evolog, ASP Program it contains no action application rules by definition. It is therefore clear, that for any set answer set candidate A of P and any Frame F , the GL-Reduct P^A and Evolog-Reduct P_F^A coincide. Any minimal model of the GL-reduct is therefore also a minimal model of the Evolog-Reduct (and vice-versa) and therefore the Evolog Models of P are identical with its Stable Models, i.e. $EM(P) = AS(P)$. \square

CHAPTER 4

Evolog Reference Implementation

List of Figures

List of Tables

List of Algorithms

2.1	Procedure <i>evaluateCommonBaseProgram</i>	11
-----	--	----

Acronyms

ASP Answer Set Programming. 3, 5, 7–10, 18

CDNL Conflict-driven Nogood Learning. 8, 10

Bibliography

- [ABW88] Krzysztof R Apt, Howard A Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, pages 89–148. Elsevier, 1988.
- [BEFI10] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. Hex programs with action atoms. In *Technical Communications of the 26th International Conference on Logic Programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.
- [CFG⁺20] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. Asp-core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020.
- [EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In *Reasoning Web International Summer School*, pages 40–110. Springer, 2009.
- [GKK⁺08] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user’s guide to gringo, clasp, clingo, and iclingo. 2008.
- [GKS12] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.
- [GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming/Symposium on Logic Programming*, volume 88, pages 1070–1080, 1988.
- [Lan19] Michael Langowski. Partial evaluation of asp programs in a lazy-grounding solver, 11 2019.
- [LN09] Claire Lefevre and Pascal Nicolas. A first order forward chaining approach for answer set computing. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 196–208. Springer, 2009.

- [LPF⁺02] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Francesco Calimeri, Tina Dell’Armi, Thomas Eiter, Georg Gottlob, Giovambattista Ianni, Giuseppe Ielpa, Christoph Koch, et al. The dlv system. In *European Workshop on Logics in Artificial Intelligence*, pages 537–540. Springer, 2002.
- [LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, volume 94, pages 23–37, 1994.
- [SN01] Tommi Syrjänen and Ilkka Niemelä. The smodels system. In *International Conference on Logic Programming and NonMonotonic Reasoning*, pages 434–438. Springer, 2001.
- [SW15] Peter Schüller and Antonius Weinzierl. Answer set application programming: a case study on tetris. In *ICLP (Technical Communications)*, 2015.
- [WBB⁺19] Antonius Weinzierl, Bart Bogaerts, Jori Bomanson, Thomas Eiter, Gerhard Friedrich, Tomi Janhunen, Tobias Kaminski, Michael Langowski, Lorenz Leutgeb, Gottfried Schenner, et al. The alpha solver for lazy-grounding answer-set programming. 2019.
- [Wei17] Antonius Weinzierl. Blending lazy-grounding and cdnl search for answer-set solving. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 191–204. Springer, 2017.