TU WIEN Informatics

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

TU WIEN Informatics

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

CHAPTER 1

Introduction

1.1 Answer Set Programming

Answer Set Programming (ASP) is a formalism for declarative problem solving based on the Stable Model semantics introduced by Gelfond and Lifschitz in 1988 [GL88]. At its core, an ASP program is a collection of conditional rules along the lines of *"if A holds true, then B must also hold"* as well as negative rules, so-called *constraints*, which prohibit certain conditions, e.g. *"If A has property p(A), then it cannot have property q(A)"*. Example 1.1.1 demonstrates the basic idea of ASP based on a program for course planning in a (very simplified) school setting.

Example 1.1.1. Listing 1.1 shows a knowledge base written in ASP, which encodes facts - things that are always true, such as *"There is a subject called maths"* - and rules, e.g. *A teacher T that is qualified to teach subject S, can be assigned to teach S*, about a simplified school domain.

Listing 1.1: School planning in ASP

```
1  % The curriculum consists of different courses.
2  subject(german).
3  subject(english).
4  subject(maths).
5  subject(biology).
6  subject(history).
7
8  % Each teacher can teach one or more subjects
9  teacher(bob).
10 can_teach(bob, english).
11 can_teach(bob, maths).
12 teacher(alice).
```

```
13    can_teach(alice, maths).
14    can_teach(alice, history).
15    teacher(claire).
16    can_teach(claire, german).
17    can_teach(claire, history).
18    teacher(joe).
19    can_teach(joe, biology).
20    can_teach(joe, history).
21
22    % Assign subjects to teachers
23    { teaches(T, S) } :- teacher(T), can_teach(T, S).
24    % such that..
25    % Every teacher teaches at least one subject
26    :- teacher(T), not teaches(T, _).
27    % Every subject is taught by exactly one teacher
28    :- subject(S), not teaches(_, S).
29    :- teaches(T1, S), teaches(T2, S), T1 != T2.
```

Evaluating the above program using an *answer set solver*, i.e. an interpreter for the ASP language, yields the following collection of so-called *answer sets*:

- $A_1 = \{teaches(bob, english), teaches(bob, maths), teaches(alice, history), teaches(claire, german), teaches(joe, biology)\}$
- $A_2 = \{teaches(bob, english), teaches(alice, maths), teaches(claire, german), teaches(claire, history), teaches(joe, biology)\}$
- $A_3 = \{teaches(bob, english), teaches(alice, maths), teaches(claire, german), teaches(joe, biology), teaches(joe, history)\}$
- $A_4 = \{teaches(bob, english), teaches(alice, maths), teaches(alice, history), teaches(claire, german), teaches(joe, biology)\}$

Intuitively, each answer set constitues a valid solution to the problem specified in the program in the sense that, if one adds the propositions form the answer set to the original program, all rules are fulfilled, while no constraint is violated. What sets ASP apart from many other logic programming formalisms is its *multi-model semantics*, i.e. that it can express more than one solution to a problem as well as mutually exclusive solutions.

As Example 1.1.1 illustrates, ASP offers a declarative and concise description language for complex problems, where formulating an imperative algorithm is non-trivial. Since its inception, ASP has been applied in a wide range of applications such as logistics [RGA+12] [AJO+21], automated music composition [EGPASB20] and even spaceflight [NBG+01]. ..

use stuff from "answer set programming at a glance"

First implementations "blabla as example **??** shows, ASP is well suited to all kinds of planning problems... it has successfully been used for ... more software-engineering-like tasks could also benefit from that kind of declarative brevity (hint: parsers, configuration, etc), but for ease of coding ,we don't want to write connectors all day...."

Apart from the established use of ASP as a formalization language for decision- or optimization problems, more recent developments in the field are increasingly targeted toward working with continuous external data in ASP programs. A prominent example from this direction of research is the stream reasoning framework LARS [BDTE18]. Closely related to the concept of processing external data is the notion of actually influencing the outside world, for instance by writing data to a network buffer, from an ASP program, while still preserving declarative semantics. The DLV-extension Acthex [BEFI10] is an example of a system with basic action support. The goal of this thesis is to implement action support in the lazy-grounding solver Alpha [WBB+19], along with a basic modularization concept, thus enabling the development of arbitrary programs fully within ASP.

1.2 Actions and Modularization in ASP - Motivation

Example 1.2.1. Listing 1.2 shows a simple ASP program that parses binary strings and calculates the corresponding decimal numbers. It uses external atoms implemented in Java for basic stirng operations: The predicate str_x_xs takes inspiration from list syntax in Haskell and splits off the first character of a given string, e.g str_x_xs["abc"]("a","bc"), while $stdlib_string_length$ and $stdlib_string_matches_regex$ test the length of a string and whether it matches some regular expression, respectively.

Listing 1.2: Parsing binary strings in ASP

```
1  encoding_scheme("1", "0", "[01]+").
2
3  % Helper - binstring_intm_decoded is the "internal"
4  % predicate we're using to recursively add up the
5  % bit values
6  binstring_intm_decoded(START_STR, START_STR, 0) :-
7      binstr(START_STR).
8
9  % Handle the case where the current bit is set
10 binstring_intm_decoded(START_STR, CURR_STR, CURR_VALUE) :-
11     binstring_intm_decoded(START_STR, LAST_STR, LAST_VALUE),
12     \&str_x_xs[LAST_STR](HIGH_CODE, CURR_STR),
13     \&stdlib_string_length[LAST_STR](LAST_LEN),
14     CURR_VALUE = LAST_VALUE + 2 ** (LAST_LEN - 1),
15     encoding_scheme(HIGH_CODE, _, REGEX),
16     &stdlib_string_matches_regex[START_STR, REGEX].
```

```
17
18      % Handle the case where the current bit is not set
19      binstring_intm_decoded(START_STR, CURR_STR, CURR_VALUE) :-
20          binstring_intm_decoded(START_STR, LAST_STR, LAST_VALUE),
21          \&str_x_xs[LAST_STR](LOW_CODE, CURR_STR),
22          \&stdlib_string_length[LAST_STR](LAST_LEN),
23          CURR_VALUE = LAST_VALUE,
24          encoding_scheme(_, LOW_CODE, REGEX),
25          \&stdlib_string_matches_regex[START_STR, REGEX].
26
27      % These are the final values
28      bin_number(BIN, DEC) :-
29          binstring_intm_decoded(BINSTR, "", DEC).
```

The program from Example 1.2.1 is a (very simple) example of a parser component that may be easier to implement in a declarative language than some (typically imperative) general purpose language such as Java or Python. With most current ASP solvers, if one wanted to write such a declarative parsing component, reading of parser inputs (typically from some file or stream) and writing of parser output would have to be done in some other language. Especially in applications where input and output is relatively simple, but parsing and data transformation logic is more involved, it would streamline application development to be able to write the whole application in one language. Furthermore, to re-use code parts - for example some generic parser as exhibited in Example 1.2.1 - one would currently have to keep that code in a separate ASP file and manually avoid conflicts in naming of predicates without any language-level support for code encapsulation and modularization. These considerations lead to the goals laid out in Section 1.3. Section 1.4 gives an overview of the state of the art on action support and approaches to program modularizations in current ASP solving systems. Section 1.5 gives an outline of the rest of the thesis.

1.3 Problem Statement

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.

1.4 State of the Art

Mostly copied from proposal for now – refine a bit!

This work aims to blend action support with modularization in the context of lazy-grounding ASP solving - all three of these areas have seen a substantial amount of research in the past.

Both Clingo [GKKS14] and DLVHEX, through the ACTHEX [BEFI10] extension, offer their own flavours of support for triggering actions from programs. While Clingo does not directly support actions as a dedicated feature - and therefore offers no strictly enforced semantics for this - similar behavior can be achieved using external functions and the reactive solving features first introduced in oClingo [GGKS11]. ACTHEX has thoroughly defined semantics for actions. In the ACTHEX model, answer set search and action execution are separate steps, where executability of actions is only determined after answer sets are calculated. While this gives users a high degree of flexibility in working with actions, it does not directly lend itself to the idea of a general purpose language where program behavior may be influenced by continuous two-way communication between a program and its environment.

With regards to Modularity, i.e. the process of "assembling" an ASP program from smaller building blocks (i.e. modules), a comprehensive semantics for so-called *nonmonotonic modular logic programs* has been introduced in [Kre18] and [DTEFK09]. While it does

not impose any restrictions on language constructs used in modules and recursion within and between modules, it also comes with rather high computational complexity and no easily available implementations so far. A more "lightweight" approach to modularization are *Templates* [IIP+04]. As the name implies, this purely syntactic approach aims to define isolated sub-programs that can be generically used to avoid code duplication throughout an application and is conceptually similar to the well-known generics in object-oriented languages such as C++ and Java. Templates are rewritten into regular ASP rules using an "explosion" algorithm which basically "instantiates" the template by generating the needed body atoms (and rules deriving them) wherever templates are used. While easy to implement and flexible, a potential disadvantage of this concept is that - due to its purely syntactical nature - programmers need to be on the watch for potential bugs arising from unintended cyclic dependencies or recursive use of templates (leading to potential non-termination of the explosion algorithm) themselves. Yet another powerful toolset for modular application development is provided by Clingo's *multi-shot-solving* [GKKS19] features. Clingo allows for parameterized sub-programs which are then repeatedly grounded in a process that is conceptually similar to the notion of module instantiation in [OJ08] and solved during solving of the overall program. However, as this "contextual grounding" needs to be programmatically controlled by an external application through Clingo's API, the inherent flexibility and usefulness for incremental solving of this approach is counterweighed by a high level of proficiency with and knowledge of the Clingo system necessary to leverage these capabilites. The concept of *lazy grounding*, i.e. interleaving of the - traditionally sequential - grounding and solving steps present in most prevalent ASP solvers, is relatively new. It has been spearheaded by the GASP [DPDPR09] and ASPERIX [LN09] solvers which avoid calculating the full grounding of an input program by performing semi-naive bottom-up evaluation along the input's topologically sorted (non-ground) dependency graph. While efficient in terms of memory use, this approach cannot stand up to the solving performance of systems like DLV or Clingo which employ their knowledge of all possible ground rules to perform conflict-driven nogood learning (CDNL) as part of their solving algorithm to great effect. Alpha [WBB+19], a more recent lazy-grounding solver, aims to bridge this gap in performance by employing CDNL-style solving techniques [Wei17] incrementally on partially ground program parts as part of its central ground-and-solve loop.

Conceptually, the common ingredient linking the - on first glance not directly connected - areas of actions in ASP, program modularization, and lazy grounding is a need for detailed static program analysis prior to solving, be it to detect potentially invalid action sequences, calculate module instantiation orders, or for up-front evaluation of stratified program parts in a lazy-grounding context. In addition, both actions and modularization can greatly benefit from - or even depend on - incremental evaluation facilities of a solver for efficient operation. Since lazy-grounding by its very definition embodies an incremental evaluation approach, it seems only natural to incorporate actions and modularization into a lazy-grounding solver's input language in order to provide ASP programmers with a powerful tool for application development. Alpha, with its good solving performance compared to other lazy-grounding systems, support for a large part

of the current ASP-Core2 language standard, and active development status, appears the natural choice as the technical backbone of this work.

1.5 Thesis Roadmap

The core part of this work is the formal specification of the Evolog language in Chapter **??**, where we formally define Evolog's action semantics and modularization concept and make some observations on the relationship between Evolog programs versus regular ASP programs. Chapter **??** gives and overview of how the formal specifications from Chapter **??** were implemented in the Alpha ASP solver, along with some examples of actual programs written in Evolog. Finally, Chapter **??** reflects on the experiences gained in using the implementation from Chapter **??** for hands-on software development. We try to gauge the practical applicability of the implementation, highlight challenges yet to be adressed, and take a look at related work.

CHAPTER 2

Preliminaries

2.1 Answer Set Programming

When speaking of ASP, we nowadays mostly refer to the language specfied 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 *artihmetic expression* $t_1 \oplus t_2$ with $\oplus \in \{+, -, *, /, **\}$ is a term.
- Given terms t_1, t_2, any *interval expression* $t_1 \ldots t_2$ is a term.
- For function symbol $f \in ID$ and argument terms $t_1, \ldots, t_n$, the *functional expression* $f(t_1, \ldots, t_n)$ 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 *arithmetic expression* $t_1 \oplus t_2$, $st(t) = st(t_1) \cup st(t_2)$.
- If t is an *interval expression* $t_1 \ldots t_2$, $st(t) = st(t_1) \cup st(t_2)$.
- If t is a *functional expression* with argument terms $t_1, \ldots, t_n$, $st(t) = st(t_1) \cup \ldots \cup st(t_n)$.

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 $t_1, \ldots, t_n$, the expression

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

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* $t_1, \ldots, t_n$ and *output terms* $t_{n+1}, \ldots, t_m$, the expression

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

is called an *external atom*. Syntacticaly, external atoms are regular atoms where $\&ext$ is the predicate symbol and $t_1, \ldots, t_m$ are argument terms.

Definition 2.1.9 (Literal)**.** A literal in ASP is an atom a or ("default"-)negated atom *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

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

for $n \geq 0$, where the *rule head* a_H is an atom and the *rule body* $b_1, \ldots, b_n$ 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, he 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.

$$\leftarrow b_1, \ldots, b_n.$$

It is syntactic sugar for

$$q \leftarrow b_1, \ldots, b_n, not\ q.$$

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 ground 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 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 oder 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 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 $f_O : H_U(P)^* \mapsto \{\top, \bot\}$, i.e. a fixed interpretation literal with argument terms $t_1, \ldots, t_n$ 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 $t_1, \ldots, t_n$ and $f_O(t_1, \ldots, t_n) = \top$.

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 $t_1, \ldots, t_n$ and $f_O(t_1, \ldots, t_n) = \bot$.

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 $S = (A_0, \ldots, A_\infty)$ 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,
- $A_\infty = \cup_{i=0}^{\infty} A_i = T_P(A_\infty)$, 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!

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

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 $l2$ 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.

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

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

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* $(P_0, \ldots, P_1)$ such that

- $\cup_{i=0}^{n} P_i = P$, i.e. S is total,
- $\forall i \geq 0 : \forall r \in P_i : \forall l \in b^+(r) : def(l) \subseteq \cup_{j=0}^{i} P_j$, 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
- $\forall i \geq 0 : \forall r \in P_i : \forall l \in b^-(r) : def(l) \subseteq \cup_{j=0}^{i-1} P_j$, 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, $S = (P_0, \ldots, P_n)$ 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 $M_{P_0} = lfp(T_{P_0})$ and $M_{P_i} = lfp(T_{P_i \cup M_{S_{i-1}}})$ 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:

- $\exists l \in b^+(r) : d \in def(l)$, i.e. d is a defining rule for some positive body literalof 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

- $\exists l \in b^-(r) : d \in def(l)$, i.e. d is a defining rule for some negative body literalof 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

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

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 douuble 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 ← (| S | − 1);
3  foreach i in 0...n do
4  |   Program grnd(P_{S_i}) = ground(P_{S_i});
5  |   Facts F_old = ∅;
6  |   do
7  |   |   F_old = F;
8  |   |   F ← pr(T_{grnd(P_{S_i})}(F));
9  |   while F_old ≠ F;
10 |   P ← P\P_{S_i};
11 end
12 return P_eval = P ∪ F;
```

CHAPTER 3

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!

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

3.1.1 Syntax

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

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

where

- 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 $l_1, \ldots, l_n$ 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*.

define (classic ASP) grounding and substitutions in preliminaries

Definition 3.1.2 (Action Rule Expansion)**.** Given a non-ground action rule R with head atom a_H, action term $f_{act}(i_1, \ldots, i_n)$ and body B consisting of literals $l_1, \ldots, l_m$, 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

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

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

$input(t_1, \ldots, t_n)$. 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 $A = \{a_1, \ldots, a_n\}$, 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:

$$p(a).\ q(b).\ r(c).$$
$$h(X, R) : @a(X, Z) = R \leftarrow p(X), q(Y), r(Z).$$

The expansion of R is:

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

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

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

The sole model of P with respect to frame F is

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

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 1.3), we haven't yet addressed the demand for transparency, i.e. that every action that is executed must be reflected in an aswer 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 repective 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 = 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 new ways to re-use and unit-test to ASP code by enabling programmers to put frequently used sub-programs into *modules* which can be used in multiple programs.

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

Definition 3.2.1 outlines the syntax for an Evolog module, i.e. a reusable sub-program the can be used in other programs.

Definition 3.2.1 (Module Definition)**.** The following EBNF describes a *module definition* in Evolog.

```
module_definition :
        "#module" ID "(" input_predicate "=>"
                         output_predicates ")"
                "{" statements "}";

input_predicate : predicate;
output_predicates :
        ("*" | ("{" predicate ("," predicate)* "}"));
predicate: ID "/" INTEGER;
statements : (fact | classic-rule | constraint)*;
```

In the context of the above grammar,

- `ID` are identifiers according to Definition 2.1.2,
- `INTEGER` are integers according to Definition 2.1.1,
- and `classical-rule` refers to a regular, i.e. non-evolog, ASP rule according to Definition 2.1.10

Intuitively, a module definition establishes a name for the sub-program in question, and specifies how input and output are translated to and from the module program. Definition 3.2.2 introduces *module atoms*, i.e. atoms in ASP rules that refer to modules.

Definition 3.2.2 (Module Atom)**.** Given a *module name mod*, a positive integer k, *input terms* $t_1, \ldots, t_n$ and *output terms* $t_{n+1}, \ldots, t_m$, the expression

$$\#mod\{k\}[t_1, \ldots, t_n](t_{n+1}, \ldots, t_m)$$

is called a *module atom*. Syntacticaly, module atoms are regular atoms where $\#mod$ is the predicate symbol and $t_1, \ldots, t_m$ are argument terms. The integer k is called the *answer set limit*. It denotes the number of answer sets of the module program to return. If k is omitted, all answer sets are returned.

Example 3.2.1 shows an encoding of the graph 3-coloring problem as an Evolog module.

Example 3.2.1 (3-Coloring Module)**.** The following module definition encodes the 3-coloring problem for a graph $G = (V, E)$ as an Evolog module. Note that the module program assumes its input to be a single fact of form `graph(V, E)`, where `V` and `E` hold the lists of vertices and edges of G, respectively.

```
1  #module 3col(graph/2 => {col/2})  {
2          % Unwrap input
3          vertex_element(V, TAIL) :- graph(list(V, TAIL), _).
4          vertex_element(V, TAIL) :-
5                  vertex_element(_, list(V, TAIL)).
6          vertex(V) :- vertex_element(V, _).
7          edge_element(E, TAIL) :- graph(_, list(E, TAIL)).
8          edge_element(E, TAIL) :-
9                  edge_element(_, list(E, TAIL)).
10         edge(V1, V2) :- edge_element(edge(V1, V2), _).
11
12         % Make sure edges are symmetric
13         edge(V2, V1) :- edge(V1, V2).
14
15         % Guess colors
16         red(V) :- vertex(V), not green(V), not blue(V).
17         green(V) :- vertex(V), not red(V), not blue(V).
18         blue(V) :- vertex(V), not red(V), not green(V).
19
20         % Filter invalid guesses
21         :- vertex(V1), vertex(V2),
22                 edge(V1, V2), red(V1), red(V2).
23         :- vertex(V1), vertex(V2),
24                 edge(V1, V2), green(V1), green(V2).
25         :- vertex(V1), vertex(V2),
26                 edge(V1, V2), blue(V1), blue(V2).
27
28         col(V, red) :- red(V).
```

```
29          col(V, blue) :- blue(V).
30          col(V, green) :- green(V).
31  }
```

3.2.2 Semantics

Intuitively, a module atom is a specialized external atom, and thus follows the general semantics of *fixed interpretation literals* (see 2.1.17). Rather than mapping to some annotated Java Method as is the case with regular external atoms, a module atom is interpreted by calling an ASP solver.

In order to make input terms of a module atom available to the module implementation program as facts, a translation step is necessary. Definition 3.2.3 describes this process in more detail.

Definition 3.2.3 (Module Input Translation)**.** Given a module definition M with input predicate p of arity k and implementation program P_M and a ground module atom a_m referencing M with input terms $t_1, \ldots, t_k$, then the ASP program that is solved to interpret a_m is obtained by adding a fact $p(t_1, \ldots, t_k)$ to P_M. We denote this conversion of input terms to an atom as the *"factification"* function $fct(t_1, \ldots, t_k)$.

Similar to module input translation, it is also necessary, to translate answer sets of an instantiated module program into output terms of the corresponding module atom. Every answer set gets converted into one set of terms, each - together with the inout terms - constituting one ground instance of the module atom. Predicates get converted into function symbols, and atoms become list items. Definition 3.2.4 introduces the notion of a list term, which is used to represent lists of terms in ASP.

Definition 3.2.4 (List Term)**.** A list term is defined inductively as follows:

- The constant denoting the empty list, *empty_list* is a list term.
- If t is a term and l is a list term, then $s_{lst}(t, l)$ is a list term.

In the above, s_{lst} is a reserved function symbol denoting list terms.

Definition 3.2.5 (Module Output Translation)**.** Given a module definition with output predicates $p_1, \ldots, p_n$, implementation program P_M and an answer sets A of P_M, the output terms of a corresponding module atom are obtained by converting the answer set into a set of terms. For every output predicate specified in the module definition, a function term is created with the predicate symbol as function symbol and a single list term as argument. The list term is constructed by collecting all atoms of the respective predicate in A into a list. We denote this conversion of an answer set as the *"termification"* function $trm(A)$.

Using the conversions from Definitions 3.2.3 and 3.2.5, we can now define the interpretation function for module atoms.

Definition 3.2.6 (Module Atom Interpretation)**.** Given (non-ground) module atom a_M with input terms $t_1, \ldots, t_k$ and output terms $t_1, \ldots, t_m$, a module definition M with input predicate p and output predicates $p_1, \ldots, p_m$, implementation program P_M, and ground substitution σ, the ground instance $grnd(a_M)$ obtained by applying σ to a_M is true iff $\{\exists A \in AS(P_M \cup fct(\sigma(t_1, \ldots, t_k))) : trm(A) = \sigma(t_1, \ldots, t_m)\}$, i.e. the module atom is true if and only if there exists an answer set of the module program together with the factified input terms that, when termified, equals the output terms of the module atom.

Corollary 3.2.1 (Fixed Module Interpretation)**.** It follows from Definition 3.2.6 that module literals are *fixed interpretation literals* according to Definition 2.1.17, where the interpretation function is the calculation described in Definition 3.2.6. Since the implementation of a module does not change at runtime, the answer sets of the module program, i.e. the output terms of a corresponding module atom, only depend on the input terms supplied by a specific ground instance. The truth value of a given module literal l is therefore independent of the rest of the program l occurs in.

Example 3.2.2 demonstrates the use of modules to find solutions to an application of the bin-packing problem.

Example 3.2.2 (Bin-packing Module)**.** Suppose a group of computer scientists $G = \{bob, alice, dilbert, claire, cate, bill, carl, mary\}$ plan to attend a conference in the neighboring town of Buzzwordville, which is 150 km distant. They have a number of cars at their disposal - Claire's camper van which seats 7 people, Dilbert's sedan seating 5, and Bob's roadster with 2 seats. Each car has different fuel efficiency, and the group wants to minimize the amount of fuel used.
To complicate matters even more, there are a number of additional constraints:

- For insurance reasons, each car must be driven by its owner.
- Alice and Carl are a couple and must travel together.
- Dilbert and Mary are not on speaking terms and must not travel together.

It is intuitively clear, that the problem at hand is a specialization of the bin-packing problem (with differently sized bins). We therefore introduce the Evolog module `bin_packing(instance2) ¯ {item_packed2}`, which calculates solutions to the bin-packing decision problem, i.e. assignments of items to bins, such that all items are packed, and no bin capacity is exceeded. Listing 3.1 shows the module in question.

Listing 3.1: Bin-packing module

```
1 #module bin_packing(instance/2 => {item_packed/2}) {
2         % Unpack input lists.
```

```
3         bin_element(E, TAIL) :-
4                 instance(lst(E, TAIL), _).
5         bin_element(E, TAIL) :-
6                 bin_element(_, lst(E, TAIL)).
7         bin(B, S) :-
8                 bin_element(bin(B, S), _).
9
10        item_element(E, TAIL) :-
11                instance(_, lst(E, TAIL)).
12        item_element(E, TAIL) :-
13                item_element(_, lst(E, TAIL)).
14        item(I, S) :-
15                item_element(item(I, S), _).
16        % For every item, guess an assignment to each bin.
17        { item_packed(I, B) : bin(B, _) } :- item(I, _).
18        % An item may only be assigned to one bin at a time
19        :- item_packed(I, B1),
20                item_packed(I, B2),
21                B1 != B2.
22        % We must not exceed the capacity of any bin.
23        capacity_used(B, C) :-
24                C = #sum{S : item(I, S),
25                item_packed(I, B)}, bin(B, _).
26        :- capacity_used(B, C),
27                C > S,
28                bin(B, S).
29        % Every item must be packed.
30        item_packed_somewhere(I) :-
31                item_packed(I, _).
32        :- item(I, _),
33        not item_packed_somewhere(I).
34 }
```

Since the implementation from 3.1 only solves the basic decision problem, we need additional code to represent the constraints in this specific variant of the problem and derive a cost value for each valid assignment. Listing 3.2 shows a validation module that accepts a bin assignment and verifies it against the additional constraints.

Listing 3.2: Custom Constraints

```
1 #module assignment_valid(assignment/1 => {assignment/1}) {
2         assignment_element(lst(E, TAIL), E, TAIL) :-
3                 assignment(lst(E, TAIL)).
4         assignment_element(ASSGN, E, TAIL) :-
5                 assignment_element(ASSGN, _, lst(E, TAIL)).
```

```
6         assigned(I, B, A) :-
7                 assignment_element(A, item_packed(I, B), _).
8
9         owner_of(claire, van).
10        owner_of(bob, roadster).
11        owner_of(dilbert, sedan).
12
13        % For insurance reasons, each car must be
14        % driven by its owner, i.e. for each car,
15        % if it is used, the owner must be assigned to it.
16        car_in_use(C) :- assigned(_, C, A).
17        :- car_in_use(C), owner_of(P, C),
18                assigned(P, C2, A), C2 != C.
19
20        % Alice and Carl are a couple and
21        % must travel together.
22        :- assigned(alice, C1, A),
23                assigned(carl, C2, A), C1 != C2.
24
25        % Dilbert and Mary are not on speaking
26        % terms and must not travel together.
27        :- assigned(dilbert, C, A), assigned(mary, C, A).
28 }
```

Finally, we calculate the cost of driving one kilometer for each assignment, and collect all assignments with the lowest possible cost value using predicate `optimal_cost_assignment/2`. Listing 3.3 shows the full program, excluding the module definitions given in Listings 3.1 and 3.2.

Listing 3.3: Main Program

```
1  person(bob).
2  person(alice).
3  person(dilbert).
4  person(claire).
5  person(cate).
6  person(bill).
7  person(carl).
8  person(mary).
9
10 car(van, 7).
11 car(roadster, 2).
12 car(sedan, 5).
13
14 distance(150).
```

```
15 cost_per_km(van, 5).
16 cost_per_km(roadster, 8).
17 cost_per_km(sedan, 3).
18
19 owner_of(claire, van).
20 owner_of(bob, roadster).
21 owner_of(dilbert, sedan).
22
23 % Create input instance for bin-packing module
24 instance(BINS, ITEMS) :-
25         BINS = #list{bin(C, S) : car(C, S)},
26         ITEMS = #list{item(P, 1) : person(P)}.
27
28 % Calculate assignments and unfold list terms
29 assignment_validation_output(ASSGN_VALID) :-
30         instance(BINS, ITEMS),
31         #bin_packing[BINS, ITEMS](ASSGN),
32         #assignment_vali[ASSGN](ASSGN_VALID).
33 assignment(A) :-
34         assignment_validation_output(
35                 lst(assignment(A), lst_empty)).
36 assignment_element(lst(E, TAIL), E, TAIL) :-
37         assignment(lst(E, TAIL)).
38 assignment_element(ASSGN, E, TAIL) :-
39         assignment_element(ASSGN, _, lst(E, TAIL)).
40 assigned(I, B, A) :-
41         assignment_element(A, item_packed(I, B), _).
42
43 % Calculate the total travelling cost for each assignment
44 assignment_cost_per_km(A, C) :-
45         C = #sum{ CKM : cost_per_km(CAR, CKM),
46         assigned(_, CAR, A) },
47         assignment(A).
48
49 % Find the minimum cost per km over all eligible
50 % assignments and assignments with that cost
51 optimal_cost_per_km(C) :-
52         C = #min{CKM : assignment_cost_per_km(_, CKM)}.
53 optimal_cost_assignment(C, A) :-
54         optimal_cost_per_km(C), assignment_cost_per_km(A, C).
```

The program from Listing 3.3 has a single answer set, with 16 instances of `optimal_cost_assignment` i.e. 16 possible assignments of people to cars which minimize the cost of a driven kilometer

(Note that we use the notation $[a, b, c]$ for a list term holding terms a, b, c, and write terms of form $item\_packed(PERSON, CAR)$ as tuples $(PERSON, CAR)$, for brevity):

- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, sedan), (bob, sedan), (carl, sedan), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, sedan), (bob, sedan), (carl, sedan), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, sedan), (bob, van), (carl, sedan), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, sedan), (bob, van), (carl, sedan), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, van), (bob, sedan), (carl, sedan), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, van), (bob, sedan), (carl, sedan), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, van), (bob, van), (carl, sedan), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, sedan), (bill, van), (bob, van), (carl, sedan), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, sedan), (bob, sedan), (carl, van), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, sedan), (bob, sedan), (carl, van), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, sedan), (bob, van), (carl, van), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, sedan), (bob, van), (carl, van), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, van), (bob, sedan), (carl, van), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, van), (bob, sedan), (carl, van), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, van), (bob, van), (carl, van), (cate, sedan), (claire, van), (dilbert, sedan), (mary, van)])$
- $optimal\_cost\_assignment(8, [(alice, van), (bill, van), (bob, van), (carl, van), (cate, van), (claire, van), (dilbert, sedan), (mary, van)])$

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. $EM(P) = AS(P)$. □

CHAPTER 4

Evolog Reference Implementation

While chapter 3 gives a complete formal specification of the Evolog language extension, this chapter describes an implementation of said specification based on the Alpha ASP solver. All code referenced in this chapter is available on the official Github repository for Alpha [Lan].

in appendix, reference a git tag with alpha version of final thesis state that can run everything in here – we wanna do this reproduceably!

4.1 Architectural overview of Alpha

Alpha is a lazy-grounding ASP solver implemented in Java. In a nutshell, Alpha calculates Computation Sequences (see Definition 2.2.1) for an input program using a CDNL-inspired solving algorithm. Starting from the set of facts contained in the program, an initial truth assignment is constructed. Based on this assignment, ground instances are calculated for all rules that *could potentially fire* based on the assignment. These ground instances are converted into *noGoods* and passed to the solver component which, using an adapted CDNL approach, guesses a new assignment based on the last set of noGoods. This process is repeated until no more guesses are possible, at which point the current assignment is either returned as an answer set, or some conflict is detected, in which case a new noGood is learned, and the solver backtracks. However, the actual ground-solve-loop is - while arguably at the heart of Alpha - just a small part of the process through which programs get evaluated. Figure 4.1 gives an overview of the building blocks making up the Alpha system.

4.1.1 Parsing and Compilation

The core ground-and-solve component of Alpha supports only a subset of the input language described in Section 2.1.1. All language features supported by the parser, but not the solver itself, get compiled into equivalent constructs in the solver's internal representation. The following transformations are applied:

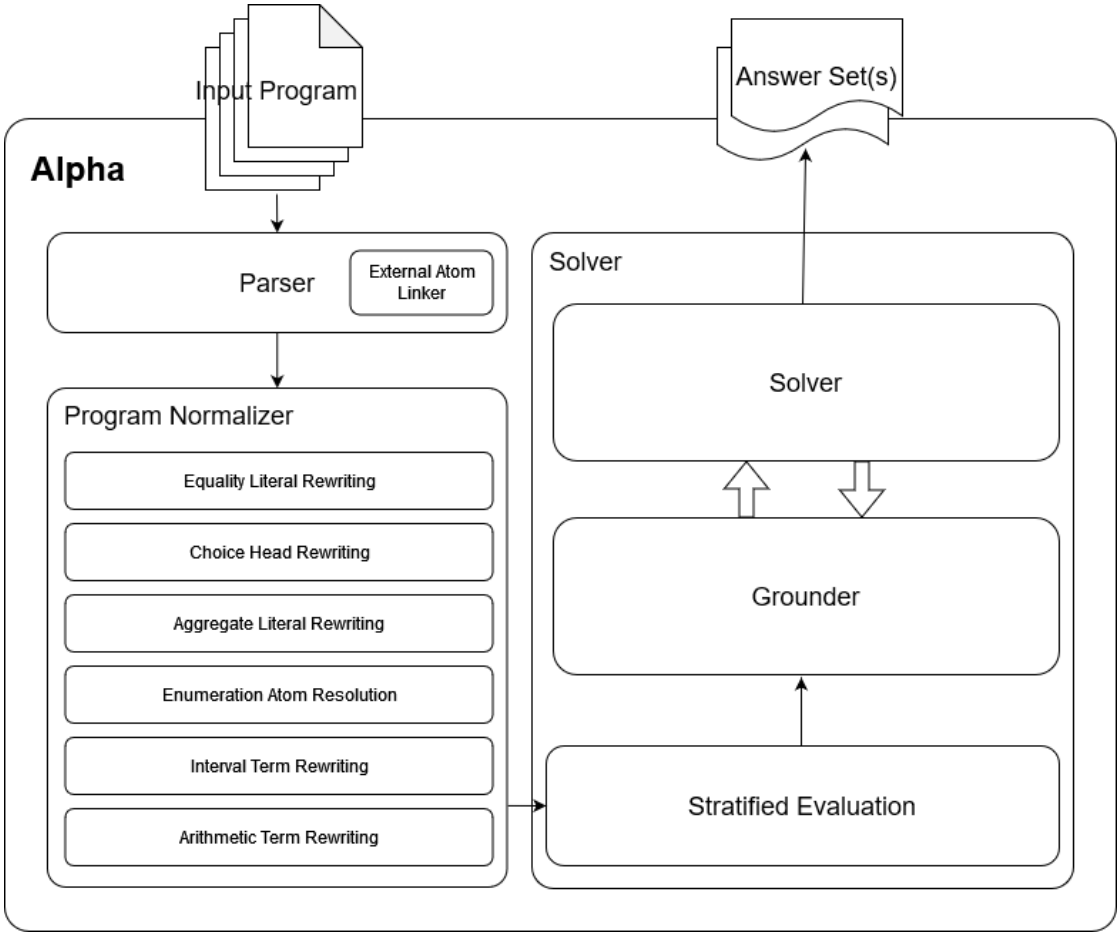

Figure 4.1: Alpha System Architecture

- *External Atom Linking*: Programs may only use external atoms whose implementations are known to the parser prior to parsing. Alpha provides a set of frequently used built-in external atoms out of the box, user-supplied code must be scanned through Alpha's API. Example A.3.1 demonstrates the use of user-supplied atom definitions. All external atom implementations are linked during parsing, i.e. every external atom in the parsed program holds a reference to the implementing Java Method.
- *Equality Literal Rewriting*: Literals like $A = B$ in rule bodies that establish an equality between variables are removed by replacing one variable with the other (e.g. B with A in the example).
- *Choice Head Rewriting*: Rules with a choice head get replaced by a set of rules and constriants that is semantically equivalent to the choice rule.
- *Aggregate Literal Rewriting*: Aggregate literals like `N = #count{ X : interesting(X)}`

are replaced by a regular literal and a set of rules deriving instances of the replacement literal equivalent to the original aggregate literal. Subsection 4.1.1 goes into more detail on the rewriting process.

- *Enumeration Atom Resolution*: Alpha provides a feature where terms can be enumerated, i.e. the solver maps user-supplied terms to integers. This is used internally for Aggregate Literal Rewriting. Section 4.1.1 describes how the atoms in question are resolved.

- *Interval Term Rewriting*: Interval terms, i.e. terms of form $A..B$ are transformed into regular variables that are bound through special internal literals which supply all values of the interval as ground instances of the variable.

- *Arithmetic Term Rewriting*: In order to simplfy grounding later, terms constituting arithmetic expressions such as $p(f(X*3, Y-4))$ are rewritten such that no arithmetic expressions occur in nested terms, i.e. the atom from before would be rewritten to $p(f(R1, R2)), R1 = X * 3, R2 = Y - 4$.

The result of the above list of transformations is what is called a *normalized* program in Alpha, which can directly be passed to the evaluation component and solved. Rewriting of Aggregate Literals and Resolution of Enumeration Atoms are of interest in the context of the Evolog reference implementation, and shall therefore be described in more detail.

Enumeration Atoms

Alpha permits use of an "enumeration" solver directive, which allows programs to associate terms with consecutive integer keys. Listing 4.1 demonstrates using an enumeration to assign integer ordinals to a set of colors.

Listing 4.1: Using the Enumeration Directive to enumerate color symbols.

```
1 #enumeration_predicate_is ordinal.
2
3 color(white). color(red). color(magenta).
4 color(yellow). color(green). color(cyan).
5 color(blue). color(black).
6
7 numbered_color(COL, NUM) :-
8     color(COL), ordinal(colors, COL, NUM).
```

The directive `#enumeration_predicate_is` designates the predicate `ordinal/3` as an *Enumeration Predicate*. All occurrences of the an enumeration predicate get replaced with a special internal predicate `_Enumeration/3`. Listing 4.2 shows the program from before after transformation.

Listing 4.2: Transformed color numbering.

```
1 color(white). color(red). color(magenta).
2 color(yellow). color(green). color(cyan).
3 color(blue). color(black).
4
5 numbered_color(COL, NUM) :-
6     color(COL), _Enumeration(colors,COL,NUM).
```

Alpha's grounding component calculates valid ground substitutions for enumeration atoms as follows:
Given an enumeration atom a_e with terms t_{enum}, t_{value} and t_{ord} and a partial substitution σ, assigning ground values to t_{enum} and t_{value},

- If the value σt_{enum} is encountered for the first time, initialize a new empty map (i.e. set of pairs with unique first elements), and associate it with term σt_{enum}.
- If the map for σt_{enum} does not contain a mapping for key σt_{value}, extend σ by the mapping $t_{ord} \mapsto o$, where $= s + 1$ and s denotes the current map size for σt_{enum}. Add the mapping $(\sigma t_{ord}, o)$ to the map and return the extended version of σ.
- If a mapping for σt_{enum} and σt_{value} exists, read the associated ordinal o, add it to σ and return the extended substitution.

From a semantics point of view, enumeration literals can intuitively be seen as "lazily assigned fixed-interpretation literals" (see Definition 2.1.17) in the way that every enumeration atom is true for exactly the ground substitution generated upon grounding it for the first time.

show the answer set

Aggregate Atoms

Alpha supports *Aggregate Listerals* as defined in [CFG+20, p. 3] by rewriting programs containing aggregate literals into sematically equivalent aggregate-free programs. A detailed description of Alpha's implementation of Aggregate Rewriting is available at [Con], but would exceed the scope of this Thesis. We will therefore focus on general concepts applying to all aggregate functions that are rewritten, and outline the rewriting procedure for aggregate literals where the minimum or maximum over a set of terms is calculated, which is the starting point for compilation of the newly introduced *#list* aggregate described in Section **??**.

In the context of this section, we consider literals of form $X \odot \#func\{t_1, \ldots, t_n : l_1, \ldots, l_m\}$, where X is a term, $\odot \in \{=, \leq\}$ and $func \in \{min, max\}$, $t_1, \ldots, t_n$ are terms and $l_1, \ldots, l_m$ literals, respectively.

Definition 4.1.1 (Aggregate Terms, Elements, Local and Global Variables, Dependencies)**.** In the following, we use the notation $var(l_1, \ldots, l_n)$ for literals $l_1, \ldots, l_n$ to denote

the set of variable terms occuring in said literals. Consider a rule r containing aggregate literal $l_{agg} = X \odot \#func\{t_1, \ldots, t_n : l_1, \ldots, l_m\}$:

$$H \leftarrow l_{agg}, b_1, \ldots, b_k.$$

Then, the set $var(l_1, \ldots, l_n) \cap var(b_1, \ldots, b_k)$ is called *global variables* of l_{agg}, denoted $glob(l_{agg})$. Roughly speaking, global variables of an aggregate literal are all variables occurring within the aggregate literal as well as other body literals of r. Given a set $V = glob(l_{agg})$, the set of literals $dep(l_{agg})$ is the mnimal set of literals that, given a substitution σ, must be ground after application of σ, in order for σ to also ground $l_{agg} \cup dep(l_{agg})$. Intuitively, global variables of an aggregate literals are all variables for which Alpha's lazy grounding component needs a ground value, in order to be able to calculate a ground instance of the aggregate literal itself. Dependencies of an aggregate literal $l_{agg}$are all literals of which the grounder needs ground instances in order to calculate a grounding of all global variables of l_{agg}.

In order to translate a rule r containing an aggregate literal $l_{agg} = X \odot \#func\{t_1, \ldots, t_n : l_1, \ldots, l_m\}$ with global variables $glob(l_{agg})$, dependencies $dep(l_{agg})$ into a semantically equivalent set of rules, the following steps are taken:

- Generate a unique identifier $id(l_{agg})$ (typically some integer) for l_{agg}
- Construct rule $r\dot{}$ in which l_{agg} is replaced by a literal $aggregate\_result(id(l\_agg)\_args, X)$
- Generate an *element rule* which derives one atom per element that is being aggregated over. Given the aggregate element $t_1, \ldots, t_n : l_1, \ldots, l_m$, the corresponding element rule is $id(l_{agg})\_element\_tuple(id(l\_agg)\_args, t_1, \ldots, t_n) \leftarrow l_1, \ldots, l_m, dep(l_{agg})$, i. e. the element rule body consists of all literals of the aggregate element together with all dependencies of the aggregate literal.
- Generate a set of *encoding rules* that encode the actual aggregate function over all elements as derived by the element rule and derives instances of the $aggregate\_result/2$ predicate.

Example 4.1.1 demonstrates how an aggregate literal for the minimum function gets rewritten by Alpha.

Example 4.1.1. Consider the program from Listing 4.3. Based on some facts of type $employee/3$ which assert that an employee works in some department and earns a given salary, we use a $\#min$-aggregate to find the employee with the lowest salary in each department.

Listing 4.3: ASP program to find the worst paid employee per department.

```
1 employee(bob, sales, 2000).
2 employee(alice, development, 6000).
```

```
employee(dilbert, development, 4500).
employee(jane, sales, 3500).
employee(carl, controlling, 5000).
employee(bill, controlling, 4000).
employee(claire, development, 5000).
employee(mary, sales, 3000).
employee(joe, controlling, 5500).

department(DEP) :- employee(_, DEP, _).

min_salary(SAL, DEP) :-
    SAL = #min{S : employee(_, DEP, S)},
    department(DEP).
worst_paid(DEP, EMP) :-
    min_salary(S, DEP), employee(EMP, DEP, S).
```

Listing 4.4 shows a rewritten version of the original program.

Listing 4.4: The program from Listing 4.3 in its rewritten version.

```
employee(bob, sales, 2000).
employee(alice, development, 6000).
employee(dilbert, development, 4500).
employee(jane, sales, 3500).
employee(carl, controlling, 5000).
employee(bill, controlling, 4000).
employee(claire, development, 5000).
employee(mary, sales, 3000).
employee(joe, controlling, 5500).

department(DEP) :- employee(_0, DEP, _1).
worst_paid(DEP, EMP) :-
    min_salary(S, DEP), employee(EMP, DEP, S).
min_salary(SAL, DEP) :-
    min_1_result(min_1_args(DEP), SAL), department(DEP).
min_1_element_tuple_less_than(ARGS, LESS, THAN) :-
    min_1_element_tuple(ARGS, LESS),
    min_1_element_tuple(ARGS, THAN), LESS <THAN.
min_1_element_tuple_has_smaller(ARGS, TPL) :-
    min_1_element_tuple_less_than(ARGS, _3, TPL).
min_1_min_element_tuple(ARGS, MIN) :-
    min_1_element_tuple(ARGS, MIN),
    not min_1_element_tuple_has_smaller(ARGS, MIN).
min_1_result(ARGS, M) :-
    min_1_min_element_tuple(ARGS, M).
```

```
26 min_1_element_tuple(min_1_args(DEP), S) :-
27     employee(_2, DEP, S), department(DEP).
```

In the rewritten version, the rule in line 14, which derives *min_salary/2* has the aggregate literal replaced with a regular literal, instances for which are derived by newly added rules that together encode the aggregate function. The individual elements over which a minimum is being calculated are derived as instances of the `min_1_element_tuple/2` predicate using the rule in line 26. In order to find the minimal instance of `min_1_element_tuple/2`, we first establish a "less than"-relation (i.e. a partial order based on numeric comparison of the second term of the element tuple instances) using the rule in line 16. The minimum element is then the one for which we cannot find a "smaller" element. This element is derived by the rule in line 21. The single answer set of the rewritten program (filtered for instances of `worst_paid/2`) is $A = \{worst\_paid(controlling, bill), worst\_paid(development, dilbert), worst\_paid(sales, bob)\}$.

4.1.2 Evaluation

TODO: general outpline of evaluation with heavy focus on stratified evaluation, ground/-solve loop is not of interest in this context, point to appropriate resources for that!

4.2 Implementing the Evolog extension in Alpha

TODO Architecture Diagram with added parts highlighted in color

4.2.1 Implementing Action Support

TODO

4.2.2 Implementing Program Modularization

Since, formally, module atoms are just a special type of external atoms, implementing support for Modularization as described in 3.2 mainly has to deal with how to parse module definitions and how to construct external atoms according to Alpha's internal representation from a set of parsed module definition for atoms which reference these definitions. While not strictly necessary, this section also discusses the newly added `#list{...}` aggregate, which has been added as syntactic sugar for combining a set of terms into a single list term according to Definition 3.2.4.

List aggregation

External atoms are defined as having input- and output-*terms* (but not multisets of terms). At the same time, it is self-evident that - in most cases - one has to pass multiple input facts to a module program - for instance, a graph has sets of edges and vertices, all of which need to be passed to a module dealing with graph problems. This is where lists

come in. By establishing the convention that the *single* input fact to a module program always has one or more *list terms* as arguements, we are able to encode all information that needs to be passed to the module into a single atom, while still staying true to external atom semantics as supported by Alpha. The parctical problem arising from this is that constructing list terms in ASP is rather cumbersome. One has to write rules to

- specify *what* goes into the list, i.e. one or more rules encoding eligible *list elements*,
- specify *in what sequence* elements go into the list, i.e. rules which establish a total order between list elements,
- actually construct the list, i.e. starting from the end, recursively construct the term holding all list elements.

Listing 4.5 shows an example of this approach, where a list term is constructed for vertices and edges of a graph, respectively.

Listing 4.5: ASP code to create vertex and edge lists for a given graph.

```
1  %% pack vertices into a vertex list
2  vertex_element(E) :- vertex(E).
3  % First, establish ordering of elements
4  vertex_element_less(N, K) :-
5      vertex_element(N), vertex_element(K), N < K.
6  vertex_element_not_predecessor(N, K) :-
7      vertex_element_less(N, I), vertex_element_less(I, K).
8  vertex_element_predecessor(N, K) :-
9      vertex_element_less(N, K),
10     not vertex_element_not_predecessor(N, K).
11 vertex_element_has_predecessor(N) :-
12     vertex_element_predecessor(_, N).
13 % Now build the list as a recursively nested function term
14 vertex_lst_element(IDX, list(N, list_empty)) :-
15     vertex_element(N),
16     not vertex_element_has_predecessor(N),
17     IDX = 0.
18 vertex_lst_element(IDX, list(N, list(K, TAIL))) :-
19     vertex_element(N),
20     vertex_element_predecessor(K, N),
21     vertex_lst_elemen(PREV_IDX, list(K, TAIL)),
22     IDX = PREV_IDX + 1.
23 has_next_vertex_element(IDX) :-
24     vertex_lst_element(IDX, _),
25     NEXT_IDX = IDX + 1,
26     vertex_lst_element(NEXT_IDX, _).
```

```
27 vertex_lst(LIST) :-
28     vertex_lst_element(IDX, LIST),
29     not has_next_vertex_element(IDX).
30
31 %% pack edges into an edge list
32 edge_element(edge(V1, V2)) :- edge(V1, V2).
33 % First, establish ordering of elements
34 edge_element_less(N, K) :-
35     edge_element(N), edge_element(K), N < K.
36 edge_element_not_predecessor(N, K) :-
37     edge_element_less(N, I), edge_element_less(I, K).
38 edge_element_predecessor(N, K) :-
39     edge_element_less(N, K),
40     not edge_element_not_predecessor(N, K).
41 edge_element_has_predecessor(N) :-
42     edge_element_predecessor(_, N).
43 % Now build the list as a recursively nested function term
44 edge_lst_element(IDX, list(N, list_empty)) :-
45     edge_element(N),
46     not edge_element_has_predecessor(N),
47     IDX = 0.
48 edge_lst_element(IDX, list(N, list(K, TAIL))) :-
49     edge_element(N),
50     edge_element_predecessor(K, N),
51     edge_lst_element(PREV_IDX,list(K, TAIL)),
52     IDX = PREV_IDX + 1.
53 has_next_edge_element(IDX) :-
54     edge_lst_element(IDX, _),
55     NEXT_IDX = IDX + 1,
56     edge_lst_element(NEXT_IDX, _).
57 edge_lst(LIST) :-
58     edge_lst_element(IDX, LIST),
59     not has_next_edge_element(IDX).
```

Given facts $vertex(a)$, $vertex(b)$, $vertex(c)$, $edge(a,b)$, $edge(b,c)$, $edge(c,a)$, we get the following answer set (filtered for predicates $edge\_lst/1$ and $vertex\_lst/1$) $A = \{edge\_lst(list(edge(c,a), list(edge(b,c), list(edge(a,b), list\_empty)))), vertex\_lst(list(c, list(b, list(a, list\_em$

Considering Listing 4.5, it is easy to see that construction of a list term looks always the same, regardless of the actual elements going into the list. Furthermore, we can observe a distinct similarity to the code generated to rewrite a $\#min$-aggregate outlined in Section 4.1.1. In the following, we define a new aggregate function, $\#list$, which Alpha rewrites into a generalized version of the list encoding from Listing 4.5.

Definition 4.2.1 (List Aggregate)**.** A *list aggregate* is an aggregate atom of the following form

$$t_{res} = \#list\{t_{elem} : l_1, \ldots, l_n\}$$

where t_{res} and t_{elem} are terms called result- and element-term, respectively, and $l_1, \ldots, l_n$ are literals.

Definition 4.2.1 formally defines the syntax of a list aggregate. Note that - in contrast to genral aggregate atoms, only equality comparisons, i.e. $X = \#list\{...\}$, are allowed, and we permit only a single element term rather than arbitrary tuples. Listing 4.6 shows a much shorter version of the program from Listing 4.5, in which list aggregation is used instead of step-by-step construction of list terms.

Listing 4.6: Creating vertex- and edge-lists using list aggregates.

```
1 %% pack vertices into a vertex list
2 vertex_lst(LIST) :-
3     LIST = #list{V : vertex(V)}.
4
5 %% pack edges into an edge list
6 edge_lst(LIST) :-
7     LIST = #list{edge(V1, V2) : edge(V1, V2)}.
```

In order to compile list-aggregates into their semantically equivalent aggregate-free versions, the following handling of list aggregates has been added to ALpha's aggregate rewriting logic:

- Substitute list aggregates in rules according to the general rules for aggregates outlined in 4.1.1.
- Element rules for list aggregates get constructed the same as for other aggregates, but permit only one element term rather than a tuple.
- In order to encode list construction, rules are added to construct a total order over all elements that should go into the list.
- List construction starts with the last element, which is defined to be the maximum element of the constructed order, successor elements are rcursively added to the front of the list.
- The list is complete when a list has been constructed, such that there is no element smaller than the current head of the list.

Listing 4.7 shows the rewritten version of the program form Listing 4.6 (Note that in the hand-crafted example from Listing 4.5, lists are constructed in descending element order, rather than ascending as is the case for the aggregate-based version).

Listing 4.7: Rewritten list aggregates from Listing 4.6

```
vertex_lst(LIST) :-
    list_1_result(list_1_no_args, LIST).
edge_lst(LIST) :-
    list_2_result(list_2_no_args, LIST).
list_1_element_greater(ARGS, N, K) :-
    list_1_element(ARGS, N),
    list_1_element(ARGS, K),
    N > K.
list_1_element_not_successor(ARGS, N, K) :-
    list_1_element_greater(ARGS, N, I),
    list_1_element_greater(ARGS, I, K).
list_1_element_successor(ARGS, N, K) :-
    list_1_element_greater(ARGS, N, K),
    not list_1_element_not_successor(ARGS, N, K).
list_1_element_has_successor(ARGS, N) :-
    list_1_element_successor(ARGS, _0, N).
list_1_lst_element(ARGS, IDX, lst(N, lst_empty)) :-
    list_1_element(ARGS, N),
    IDX = 0,
    not list_1_element_has_successor(ARGS, N).
list_1_lst_element(ARGS, IDX, lst(N, lst(K, TAIL))) :-
    list_1_element(ARGS, N),
     list_1_element_successor(ARGS, K, N),
     list_1_lst_element(ARGS, PREV_IDX, lst(K, TAIL)),
     IDX = PREV_IDX + 1.
list_1_has_next_element(ARGS, IDX) :-
    list_1_lst_element(ARGS, IDX, _1),
    NEXT_IDX = IDX + 1,
    list_1_lst_element(ARGS, NEXT_IDX, _2).
list_1_result(ARGS, LIST) :-
    list_1_lst_element(ARGS, IDX, LIST),
    not list_1_has_next_element(ARGS, IDX).
list_1_element(list_1_no_args, V) :-
    vertex(V).
list_2_element_greater(ARGS, N, K) :-
    list_2_element(ARGS, N),
    list_2_element(ARGS, K),
    N > K.
list_2_element_not_successor(ARGS, N, K) :-
    list_2_element_greater(ARGS, N, I),
    list_2_element_greater(ARGS, I, K).
list_2_element_successor(ARGS, N, K) :-
```

```
43      list_2_element_greater(ARGS, N, K),
44      not list_2_element_not_successor(ARGS, N, K).
45  list_2_element_has_successor(ARGS, N) :-
46      list_2_element_successor(ARGS, _3, N).
47  list_2_lst_element(ARGS, IDX, lst(N, lst_empty)) :-
48      list_2_element(ARGS, N),
49      IDX = 0,
50      not list_2_element_has_successor(ARGS, N).
51  list_2_lst_element(ARGS, IDX, lst(N, lst(K, TAIL))) :-
52      list_2_element(ARGS, N),
53      list_2_element_successor(ARGS, K, N),
54      list_2_lst_element(ARGS, PREV_IDX, lst(K, TAIL)),
55      IDX = PREV_IDX + 1.
56  list_2_has_next_element(ARGS, IDX) :-
57      list_2_lst_element(ARGS, IDX, _4),
58      NEXT_IDX = IDX + 1,
59      list_2_lst_element(ARGS, NEXT_IDX, _5).
60  list_2_result(ARGS, LIST) :-
61      list_2_lst_element(ARGS, IDX, LIST),
62      not list_2_has_next_element(ARGS, IDX).
63  list_2_element(list_2_no_args, edge(V1, V2)) :-
64      edge(V1, V2).
```

Parsing Module Definitions

Module definitions are parsed together with regular ASP code. Given a file containing a set of facts and rules and an arbitrary number of module definitions, Alpha's parser will group all rules and facts outside any module definition into one ASP program (i.e. the "main program"), and emit a set containing all encountered module definitions alongside with the parsed main program. Definition 4.2.2 details the technical representation of a module definition (which is basically a data structure storing all elements specified in the syntactical definition of a module, see 3.2.1). The actual code used in Alpha for parsing and storing module definitions can be found in the Appendix, see Example A.3.2.

Definition 4.2.2 (Internal module representation)**.** A parsed module definition gets stored as a tuple $M = (name, p_{in}, OUT, PROG)$ where

- *name* is a string representing the name of the module. This must be unique, i.e. Alpha currently has no concept of namespaces or similar means of distinguishing equally named modules,
- p_{in} is a predicate (represented as a string of form *name*/*arity*) called *input specification*,

- OUT denotes a (possibly empty) set of predicates called *output specification*,
- and $PROG$ is the ASP program holding the actual implementation code for the module.

Module Literal Compilation

Once a program and all contained module definitions have been parsed, every module atom (i.e. call to a module from a rule body) needs to be linked to its implementation. In Alpha, external atoms are evaluated during grounding. Definition 4.2.3 formally defines the notion of an *external predicate interpretation*, i.e. an interpretation function for ground instances of external atoms.

Definition 4.2.3 (External Predicate Interpretation)**.** Given an external atom $\&ext[i_1, \ldots, i_n](o_1, \ldots, o_m)$ with input terms $i_1, \ldots, i_n$ and output terms $o_1, \ldots, o_m$, and a substitution σ assigning ground values to at least all input terms of ex, the *predicate interpretation function* $f : HU^n \mapsto 2^{HU^m}$ is a function which, for a given n-tuple over the Herbrand Universe HU, returns all m-tuples over HU for which the ground atom $\&ext[\sigma(i_1), \ldots, \sigma(i_n)](r_1, \ldots, r_m)$ where $(r_1 \ldots, r_m) \in f(\sigma(i_1), \ldots, \sigma(i_n))$ is true.

Implementation-wise, predicate interpretations are Java methods with a specific signature. Listing 4.8 shows the corrseponding interface definition. Note that, for an implementation of `PredicateInterpretation` to be valid, it must be pure functional, i.e. not have any side-effects.

Listing 4.8: Java Interface for Predicate Interpretations

```
@FunctionalInterface
public interface PredicateInterpretation {

    Set<List<Term>> evaluate(List<Term> terms);

}
```

Constructing interpretations for Module Atoms Listing 4.9 shows the (abbreviated) Java code used in Alpha to construct a `PredicateInterpretation` for a module atom. Boundary checks and similar validations as well as auxiliary method calls are left out for easier readability. The interpretation itself is constructed using the Lambda-expression `PredicateInterpretation interpretation = terms -> { ... }`, where `terms` is the list of input terms, which gets wrapped into an atom of the predicate defined in `inputSpec` (i.e. the module input specification). Answer sets get filtered based on predicate names according to the module's output specification. Finally, each answer set is converted to a list of list terms using function `answerSetToTerms`, and the resulting set of term lists is returned.

Listing 4.9: Constructing module interpretations

```
private ExternalAtom translateModuleAtom(ModuleAtom atom,
   Map<String, Module> moduleTable) {
    ...
    Predicate inputSpec = definition.getInputSpec();
    ...
    Set<Predicate> outputSpec = definition.getOutputSpec();
    Set<Predicate> expectedOutputPredicates;
    if (outputSpec.isEmpty()) {
        expectedOutputPredicates =
           calculateOutputPredicates(normalizedImplementation);
    } else {
        expectedOutputPredicates = outputSpec;
    }
    ...
    PredicateInterpretation interpretation = terms -> {
        BasicAtom inputAtom = Atoms.newBasicAtom(inputSpec,
           terms);
        NormalProgram program = Programs.newNormalProgram(
            normalizedImplementation.getRules(),
                ListUtils.union(List.of(inputAtom),
            normalizedImplementation.getFacts()),
            normalizedImplementation.getInlineDirectives(),
            Collections.emptyList());
        java.util.function.Predicate<Predicate> filter =
           outputSpec.isEmpty() ? p -> true :
           outputSpec::contains;
        Stream<AnswerSet> answerSets =
           moduleRunner.solve(program, filter);
        if (atom.getInstantiationMode()
            .requestedAnswerSets().isPresent()) {
                answerSets = answerSets.limit(
                atom.getInstantiationMode()
                    .requestedAnswerSets().get());
        }
        return answerSets.map(as -> answerSetToTerms(as,
           expectedOutputPredicates))
            .collect(Collectors.toSet());
    };
    return Atoms.newExternalAtom(atom.getPredicate(),
       interpretation, atom.getInput(), atom.getOutput());
}
```

```
private static List<Term> answerSetToTerms(AnswerSet
   answerSet, Set<Predicate> moduleOutputSpec) {
    List<Term> terms = new ArrayList<>();
    for (Predicate predicate : moduleOutputSpec) {
        if (!answerSet.getPredicates().contains(predicate)) {
            terms.add(Terms.EMPTY_LIST);
        } else {
            terms.add(Terms.asListTerm(
                    answerSet.getPredicateInstances(predicate).stream()
                        .map(Atoms::toFunctionTerm)
                        .collect(Collectors.toList())));
        }
    }
    return terms;
}
```

need to decide whether module recursion is a thing! (probably should be)

4.3 Evolog applications

TODO: A few programs using both actions and modules to achieve something cool!

CHAPTER 5

Results

TODO: Observations from running programs, edge cases, bugs(?)

CHAPTER 6

Conclusion

TODO: related work
TODO: discussion of contribuitions and results,
TODO: turning into future work, what Evolog would need to get from prototype to product.

APPENDIX A

Additional Material

Some more samples that would be too much inline. do proper text

A.1 Installing Alpha

TODO: Brief how-to to install an Alpha build made for this thesis and run the examples (Windoze/Linux).

A.2 Running ASP code with Alpha

TODO: Debugging features of CLI application, basic API usage.

A.3 Examples

Example A.3.1 (Fibonacci-Numbers using external atoms)**.** The following code snippet demonstrates how to run a program which includes user-supplied external atoms using Alpha. Since the Alpha Commandline-App currently does not support loading Atom Definitions from jar files, the solver is directly called from Java code in this example.

In this example, we use an external predicate definition `fibonacci_number/2` to efficiently calculate Fibonacci numbers. The actual ASP program generates all Fibonacci numbers up to index 40 that are even. The ASP program is shown in Listing A.1, while Listing A.2 shows the Java implementation of the external predicate.

Listing A.1: ASP program to find even Fibonacci numbers.

```
1 %% Find even fibonacci numbers up to F(40).
2 fib(N, FN) :- &fibonacci_number[N](FN), N = 0..40.
```

```
3 even_fib(N, FN) :- fib(N, FN), FN \ 2 = 0.
```

The Java implementation of the `fibonacci_number/2` predicate makes use of *Binet's Formula* to efficiently compute Fibonacci numbers using a closed form expression of the sequence.

can we cite something here?

Listing A.2: Fibonacci number computation in Java.

```
public class CustomExternals {

        private static final double GOLDEN_RATIO =
           1.618033988749894;
        private static final double SQRT_5 = Math.sqrt(5.0);

        /**
         * Calculates the n-th Fibonacci number using a
            variant of Binet's Formula
         */
        @Predicate(name="fibonacci_number")
        public static Set<List<ConstantTerm<Integer>>>
           fibonacciNumber(int n) {
                return
                   Set.of(List.of(Terms.newConstant(binetRounding(n))));
        }

        public static int binetRounding(int n) {
                return (int) Math.round(Math.pow(GOLDEN_RATIO,
                   n) / SQRT_5);
        }

}
```

The Alpha solver is invoked from a simple Java method, which uses Alpha's API to first compile the external atom definition, and then run the solving process for the parsed ASP program. This is illustrated in Listing A.3.1

```
public class CustomExternalsApp {

        public static void main(String[] args) throws
           IOException {
                Alpha alpha = AlphaFactory.newAlpha();
                Map<String, PredicateInterpretation>
                   customExternals =
                        Externals.scan(CustomExternals.class);
                String aspCode =
```

```
                        Files.readString(
                                Paths.get("src/main/resources/customExternals.as
                                InputProgram program =
                                        alpha.readProgramString(aspCode,
                                customExternals);
                alpha.solve(program).forEach(as -> {
                        System.out.println("Answer set:\n" +
                           as);
                });
        }

}
```

Example A.3.2 (Module Parsing)**.** Listing A.3 shows the actual ANTLR-grammar used in Alpha to parse module definitions. Parsed Module Definitions are stored in instances of the `Module` type. Listing A.4 shows the corresponding Java interface definition.

Listing A.3: ANTLR grammar for module definitions

```
1  directive_module:
2          SHARP DIRECTIVE_MODULE id
3                  PAREN_OPEN module_signature PAREN_CLOSE
4                          CURLY_OPEN statements CURLY_CLOSE;
5
6  module_signature :
7          predicate_spec ARROW CURLY_OPEN
8                  ('*' | predicate_specs) CURLY_CLOSE;
9
10 predicate_specs:
11         predicate_spec (COMMA predicate_specs)?;
12
13 predicate_spec: id '/' NUMBER;
```

Note how the `module_signature` rule permits the shorthand `*` instead of an output predicate list. This is used as a shorthand to express the list of all predicates derived by any rule in the module (i.e. emit full, unfiltered answer sets).

Listing A.4: Java interface specifying Alpha's internal representation of a module definition.

```
public interface Module {

        String getName();

        Predicate getInputSpec();
```

```
            Set<Predicate> getOutputSpec();

            InputProgram getImplementation();

    }
```

List of Figures

List of Tables

List of Algorithms

Acronyms

ASP Answer Set Programming. 1–4, 7, 9, 11, 13–16, 24, 32, 33

CDNL Conflict-driven Nogood Learning. 14, 16, 33

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.

[AJO+21] Dirk Abels, Julian Jordi, Max Ostrowski, Torsten Schaub, Ambra Toletti, and Philipp Wanko. Train scheduling with hybrid answer set programming. *Theory and Practice of Logic Programming*, 21(3):317–347, 2021.

[BDTE18] Harald Beck, Minh Dao-Tran, and Thomas Eiter. Lars: A logic-based framework for analytic reasoning over streams. *Artificial Intelligence*, 261:16–70, 2018.

[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.

[Con] Alpha Contributors.

[DPDPR09] Alessandro Dal Palu, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. Gasp: Answer set programming with lazy grounding. *Fundamenta Informaticae*, 96(3):297–322, 2009.

[DTEFK09] Minh Dao-Tran, Thomas Eiter, Michael Fink, and Thomas Krennwallner. Modular nonmonotonic logic programming revisited. In *International Conference on Logic Programming*, pages 145–159. Springer, 2009.

[EGPASB20] Flavio Everardo, Gabriel Rodrigo Gil, OB Pérez Alcalá, and G Silva Ballesteros. Using answer set programming to detect and compose music in the structure of twelve bar blues. In *Proceedings of the Thirteenth Latin American Workshop on Logic/Languages, Algorithms and New Methods*

of Reasoning. Virtual conference at Universidad Nacional Autónoma de México (UNAM), pages 10–11, 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.

[GGKS11] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 54–66. Springer, 2011.

[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.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo= asp+ control. *arXiv preprint arXiv:1405.3694*, 2014.

[GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[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.

[IIP+04] Giovambattista Ianni, Giuseppe Ielpa, Adriana Pietramala, Maria Carmela Santoro, and Francesco Calimeri. Enhancing answer set programming with templates. In *NMR*, pages 233–239, 2004.

[Kre18] Thomas Krennwallner. *Modular nonmonotonic logic programs*. PhD thesis, Technical University of Vienna, 2018.

[Lan] Michael Langowski.

[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.

[NBG+01] Monica Nogueira, Marcello Balduccini, Michael Gelfond, Richard Watson, and Matthew Barry. An a-prolog decision support system for the space shuttle. In *Practical Aspects of Declarative Languages: Third International Symposium, PADL 2001 Las Vegas, Nevada, March 11–12, 2001 Proceedings 3*, pages 169–183. Springer, 2001.

[OJ08] Emilia Oikarinen and Tomi Janhunen. Achieving compositionality of the stable model semantics for smodels programs. *arXiv preprint arXiv:0809.4582*, 2008.

[RGA+12] Francesco Ricca, Giovanni Grasso, Mario Alviano, Marco Manna, Vincenzino Lio, Salvatore Iiritano, and Nicola Leone. Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming*, 12(3):361–381, 2012.

[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.