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]. ..

add more content here, e.g. from "answer set programming at a glance"

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 $str\_x\_xs[\text{"}abc\text{"}](\text{"}a\text{"}, \text{"}bc\text{"})$, 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

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 5 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*. Given a literal l, the expression $pred(l)$ refers to the predicate of l (e.g $pred(p(a)) = p/1$).

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

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

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

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 answer-set solving using Alpha

The theoretical notion underlying lazy-grounding answer-set solving is that of a *Computation Sequence*, which is formalized in Definition 2.2.1. Intuitively, a computation sequence is a sequence of (ground) rules firing, such that, at the end, the atoms derived by the rules from the computation sequence (together with all facts of the program) constitute an answer set.

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.

It is easily observed that, in order to calculate computation sequences, one does not need a full grounding of the input program. Since facts are ground by definition, rules that only depend on facts can be grounded (and evaluated) based on facts alone. The same holds for rules depending on facts and rules from the previous step, etc. Example 2.2.1 demonstrates how a computation sequence for a very simple positive program can be caluclated using a naive algorithm based on repeated application of the T_P-operator.

Example 2.2.1 (Lazy-grounding a positive logic program)**.** In order to illustrate the use of computation sequences for lazy-grounding, we consider the program P in Listing 2.1. We denote the rule on line 2 as r_1 and the one on line 3 as r_2.

Listing 2.1: A positive, non-ground program

```
1 p(a). p(b). p(c). q(b). q(c). q(d).
2 r(X) :- p(X), q(X).
3 t(X, Y) :- r(X), r(Y), X != Y.
```

Starting from the set of facts on line 1, which we denote as F, we now look for rules that only depend on predicates in F (i.e. body literals only have predicates for which we already know ground instances). The only applicable rule is r_1 for which we can construct ground instances `r(b) :- p(b), q(b)` and `r(c) :- p(c), q(c).` Continuing this process, now based on the set $F \cup \{r(b), r(c)\}$, we find ground instances `t(b, c) :- r(b), r(c), b != c` and `t(c, b) :- r(c), r(b), c != b` of r_2. We thus arrive at the following computation sequence:

$$A_0 = \{p(a), p(b), p(c), q(b), q(c), q(d)\},$$
$$A_1 = \{p(a), p(b), p(c), q(b), q(c), q(d), r(b), r(c)\},$$
$$A_2 = \{p(a), p(b), p(c), q(b), q(c), q(d), r(b), r(c), t(b, c), t(c, b)\}$$

The last element of the sequence, A_2 is the sole answer set of P.

While the intuitive approach from Example 2.2.1 clearly works for positive programs, things get more complicated when negation is involved. Consider the following program:

```
1  p(a). p(b). p(c). q(c).
2  s(X) :- p(X), q(X).
3  t(X) :- p(X), not s(X).
```

In order to arrive a the correct answer set $A = \{p(a), p(b), p(c), q(c), s(c), t(a), t(b)\}$, one has to first evaluate all rules that could potentially derive instances of $s/1$, before starting to evaluate the last rule, in order to end up with a valid computation sequence. Evaluation orders for this kind of programs can be calculated using the notion of *stratification* [ABW88] which is described in detail in Section 2.2.1.

2.2.1 Structural Dependency Analysis and Stratified Evaluation

In a nutshell, a *stratifiable* logic program is a program, for which we can calculate a partition into sub-programs, such that, when evaluating the sub-programs sequentially, one always ends up with a correct computation sequence. Definition 2.2.2 formally characterizes stratifiable programs.

Definition 2.2.2 (Stratifiable answer set program, as stated in [Lan19], adapted from[ABW88], [EIK09])**.** Given a program $P = \{r_0, ..., r_n\}$, P is called *stratifiable* iff there is a partition $P = P_{S_0} \cup \ ... \cup \ P_{S_n}$ so that for each $0 \leq i \leq n$ the following holds:

- For every positive body literal l_b of every rule in P_{S_i}, every rule that derives instances of the predicate $pred(l_b)$, i.e. where the predicate of the head atom $pred(a_H)$ is $pred(l_b)$, is contained in some P_{S_j} with $j \leq i$.
- For every negative body literal *not* l_b of every rule in P_{S_i}, every rule that derivesinstances of the predicate $pred(l_b)$, i.e. where the predicate of the head atom $pred(a_H)$is $pred(l_b)$, is contained in some P_j with $j < i$.

The individual subprograms P_{S_i} are called strata. A *stratification* $S = \{P_{S_0}, \ldots, P_{S_n}\}$refers to the set of all strata making up a partition which satisfies the criteria above.

The least model, i.e. the sole answer set, of a stratifiable program can be computed by sequentially applying the immediate consequence operator to every stratum, see Definition 2.2.3.

Definition 2.2.3 (Least model of stratifiable programs, adapted from [EIK09])**.** Let P_{strat} be a stratifiable program for which a stratification $S = \{P_{S_0}, \ldots, P_{S_n}\}$ exists and let $T_{P_{S_i}}$ be the immediate consequence operator for the sub-program defined by stratum P_{S_i}. Furthermore, for a model M, let $pr(M)$ be the program consisting of the facts representing the atoms in M. Then the least model $LM(P_{strat})$ of P_{strat} is defined as follows. The sequence $\langle M_{S_i} \rangle$, $0 \leq i \leq n$ with $M_{S_0} = lfp(T_{P_{S_0}})$ and $M_{S_i} = lfp(T_{(P_{S_i} \cup \ pr(M_{S_{i-1}}))})$ for all $1 \leq i \leq n$ defines the (iterative) least model for each stratum. The least model

$LM(P_{strat})$ of program P_{strat} is then the iterative least model of the highest stratum M_{S_n}, i.e. the end of the sequence $\langle M_{S_i} \rangle$.

Since the evaluation mode described in Definition 2.2.3 does not require any backtracking, i.e. the result of each step is part of the final result, we observe that te intermediate results of the immediate consequence operator yield a computation sequence for P. Example 2.2.2 illustrates evaluation of a short stratifiable program.

Example 2.2.2 (Evaluating a stratified program)**.** Consider the program P from Listing 2.2.

Listing 2.2: A stratifiable program.

```
1 p(a). p(b). p(c). q(c). q(d).
2
3 s(X) :- p(X), q(X).
4 t(X) :- p(X), not s(X).
5 u(X) :- q(X), not p(X).
6
7 v(X, Y) :- t(X), u(Y), X != Y.
```

A possible stratification S of P could look as follows:

$$
\begin{aligned}
S_0 = \{&p(a).\ p(b).\ p(c).\ q(c).\ q(d). \\
&s(X) \leftarrow p(X),\ q(X).\} \\
S_1 = \{&t(X) \leftarrow p(X),\ not\ s(X). \\
&u(X) \leftarrow q(X),\ not\ p(X).\} \\
S_2 = \{&v(X,Y) \leftarrow t(X),\ u(Y),\ X\ !=\ Y.\}
\end{aligned}
$$

In this case, the least model of P can be calculated using the method from Definition 2.2.3.

The individual results of the T_P operator constitute a computation sequence.

$$T^1_{P_{S_0}} = T_{P_{S_0}}(\emptyset) = \{p(a), p(b), p(c), q(c), q(d)\}$$
$$T^2_{P_{S_0}} = T_{P_{S_0}}(T^1_{P_{S_0}}) = \{p(a), p(b), p(c), q(c), q(d), s(c)\}$$
$$T^3_{P_{S_0}} = T_{P_{S_0}}(T^2_{P_{S_0}}) = \{p(a), p(b), p(c), q(c), q(d), s(c)\} = lfp(T_{P_{S_0}})$$

$$T_{P_{E1}} = T_{P_{S_1 \cup pr(lfp(T_{P_{S_0}}))}}$$
$$T^1_{P_{E1}} = T_{P_{E1}}(\emptyset) = \{p(a), p(b), p(c), q(c), q(d), s(c)\}$$
$$T^2_{P_{E1}} = T_{P_{E1}}(T^1_{P_{E1}}) = \{p(a), p(b), p(c), q(c), q(d), s(c), t(a), t(b), u(d)\}$$
$$T^3_{P_{E1}} = T_{P_{E1}}(T^2_{P_{E1}}) = \{p(a), p(b), p(c), q(c), q(d), s(c), t(a), t(b), u(d)\} = lfp(T_{P_{E1}})$$

$$T_{P_{E2}} = T_{P_{S_2 \cup pr(lfp(T_{P_{E1}}))}}$$
$$T^1_{P_{E2}} = T_{P_{E2}}(\emptyset) = \{p(a), p(b), p(c), q(c), q(d), s(c), t(a), t(b), u(d)\}$$
$$T^2_{P_{E2}} = T_{P_{E2}}(T^1_{P_{E2}}) = \{p(a), p(b), p(c), q(c), q(d), s(c), t(a), t(b), u(d), v(a,d), v(b,d)\}$$
$$T^3_{P_{E2}} = T_{P_{E2}}(T^2_{P_{E2}}) = \{p(a), p(b), p(c), q(c), q(d), s(c), t(a), t(b), u(d), v(a,d), v(b,d)\} = lfp(T_{P_{E2}})$$

$$LM(P) = lfp(T_{P_{E2}})$$

However, in general, ASP programs are not stratifiable. Example 2.2.3 shows a typical non-stratifiable program.

Example 2.2.3 (A non-stratifiable program)**.** Consider a variant of the well-known 3-coloring problem for undirected graphs, where some vertices "stay blank", i.e. they are excluded from coloring. Essentially, prior to calculating actual colorings, we remove all ignored vertices and their corresponding edges from the graph and color only the resulting subgraph. Program P_{col} in Listing 2.3 shows a possible encoding of this problem.

Listing 2.3: Graph 3-coloring with excluded vertices

```
1  vertex(a). vertex(b). vertex(c). vertex(d).
2  edge(a, b). edge(a, c). edge(a, d).
3  edge(b, c). edge(b, d). edge(c, d).
4  edge(X, Y) :- edge(Y, X).
5
6  exclude_vertex(d).
7  exclude_edge(V1, V2) :- edge(V1, V2), exclude_vertex(V1).
8  exclude_edge(V1, V2) :- exclude_edge(V2, V1).
9
10 coloring_vertex(V) :- vertex(V), not exclude_vertex(V).
11 coloring_edge(V1, V2) :-
```

```
12          edge(V1, V2), not exclude_edge(V1, V2).
13
14  % Guess colors
15  red(V) :- coloring_vertex(V), not green(V), not blue(V).
16  green(V) :- coloring_vertex(V), not red(V), not blue(V).
17  blue(V) :- coloring_vertex(V), not red(V), not green(V).
18
19  % Filter invalid guesses
20  :- coloring_vertex(V1), coloring_vertex(V2),
21          coloring_edge(V1, V2), red(V1), red(V2).
22  :- coloring_vertex(V1), coloring_vertex(V2),
23          coloring_edge(V1, V2), green(V1), green(V2).
24  :- coloring_vertex(V1), coloring_vertex(V2),
25          coloring_edge(V1, V2), blue(V1), blue(V2).
```

Clearly, P_{col} is not stratifiable - the color assignment rules cyclically depend on each other through negated body literals. In other words, the rules on lines 15 to 17 are *mutually exclusive*, i.e. a solver has to *choose* which of the three should fire for a given ground instance of *coloring_vertex*/1.

The program from Example 2.2.3 can not be evaluated by iterative fixpoint-calculation as in Example 2.2.2 because it is not stratifiable. However, it is still possible to evaluate at least a part of it using this method, rather than a computationally more complex "trial-and-error"-approach. This partial evaluation approach uses *splitting sets*, which are characterized in Definition 2.2.4. Intuitively, a splitting set S is a set of atoms, such that if an atom a is in S, also all atoms it depends on (i.e. body atoms of rules deriving a) are in S. By the splitting set theorem, a splitting sets lets us partition a program P such that it can be evaluated incrementally, by first evaluating the bottom $B_U(P)$ and then the top $T_U(P)$. Since this section deals with non-ground programs specifically, we also give an adapted definition for splitting sets of nonground programs, see Definition 2.2.5. Example 2.2.4 applies the splitting set theorem to the program from Example 2.2.3.

Definition 2.2.4 (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 theatoms corresponding to all body literals of r are in U. The set of rules corresponding to U, i.e. the rules definingthe 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 $T_U(P)$.

Theorem 2.2.1 (Splitting Set Theorem, adapted from [LT94])**.** Given a program P and splitting set U which splits P into bottom $B_U(P)$ and top $T_U(P) = P \setminus B_U(P)$. Then aset of atoms A is an answer set if and only if $A = X \cup Y$ for sets X and Y where

- $X \in AS(B_U(P))$, i.e. X is an answer set of bottom $B_U(P)$ and

- $Y \in AS(T_U(P) \cup X)$, i.e. Y is an answer set of the program resulting from adding X as facts to $T_U(P)$

Definition 2.2.5 (Nonground splitting set)**.** Given a non-ground program P, we denote splitting sets as sets of predicates rather than atoms. For a given predicate p, we denote as $dep(p)$ the *dependencies* of p:

$$dep(p) = \{q \mid \exists r \in P : pred(h(r)) = p \wedge \exists l \in b(r) : pred(l) = q\}$$

Intuitively, the dependencies of a predicate p are all predicates of which all ground instances must be known in order to compute all ground instances of p. A set S of predicates $p_1, \ldots, p_n$ is a splitting set of P if the following holds.

$$\forall p : p \in S \Rightarrow dep(p) \in S$$

A non-ground splitting set S_{ng} is sematically equivalent to the ground splitting set S_{grnd} one gets by taking all ground instances of all predicates in S_{ng} from the ground program $grnd(P)$.

Example 2.2.4 (Partial evaluation using splitting sets)**.** Given program P_{col} from Listing 2.3 in Example 2.2.3, let S be the following (non-ground) splitting set:

$$S = \{coloring\_vertex/1,\ coloring\_edge/2,\ vertex/1,\ exclude\_vertex/1,\\ edge/2,\ exclude\_edge/2\}$$

Using S, we can split P_{col} into bottom $B_S(P_{col})$, which is shown in Listing 2.4, and top $T_S(P_{col})$ in Listing 2.5.

Listing 2.4: The bottom part of P_{col}

```
1  vertex(a). vertex(b). vertex(c). vertex(d).
2  edge(a, b). edge(a, c). edge(a, d).
3  edge(b, c). edge(b, d). edge(c, d).
4  edge(X, Y) :- edge(Y, X).
5
6  exclude_vertex(d).
7  exclude_edge(V1, V2) :- edge(V1, V2), exclude_vertex(V1).
8  exclude_edge(V1, V2) :- exclude_edge(V2, V1).
9
10 coloring_vertex(V) :- vertex(V), not exclude_vertex(V).
11 coloring_edge(V1, V2) :-
12         edge(V1, V2), not exclude_edge(V1, V2).
```

Looking at $B_S(P_{col})$, we can observe that this sub-program of P_{col} is now stratified, and can therefore be evaluated using the straight-forward computation employed in

Example 2.2.2. We arrive at the following model for $B_S(P_{col})$:

$$\begin{aligned} LM(B_S(P_{col})) = \{&coloring\_vertex(b),\ exclude\_vertex(d),\ edge(b,d),\\ &edge(a,b),\ edge(c,d),\ edge(c,b),\ coloring\_edge(b,c),\\ &coloring\_edge(a,c),\ coloring\_edge(b,a),\ edge(a,d),\\ &coloring\_edge(c,a),\ vertex(c),\ vertex(a),\ edge(d,b),\\ &exclude\_edge(d,b),\ exclude\_edge(c,d),\ exclude\_edge(a,d),\\ &coloring\_vertex(c),\ exclude\_edge(b,d),\ coloring\_vertex(a),\\ &edge(a,c),\ edge(b,c),\ edge(b,a),\ edge(d,c), coloring\_edge(a,b)\\ &coloring\_edge(c,b),\ vertex(d),\ exclude\_edge(d,a),\ vertex(b)\\ &edge(c,a),\ exclude\_edge(d,c),\ edge(d,a)\} \end{aligned}$$

Listing 2.5: The top part of P_{col}

```
1  % Guess colors
2  red(V) :- coloring_vertex(V), not green(V), not blue(V).
3  green(V) :- coloring_vertex(V), not red(V), not blue(V).
4  blue(V) :- coloring_vertex(V), not red(V), not green(V).
5
6  % Filter invalid guesses
7  :- coloring_vertex(V1), coloring_vertex(V2),
8         coloring_edge(V1, V2), red(V1), red(V2).
9  :- coloring_vertex(V1), coloring_vertex(V2),
10        coloring_edge(V1, V2), green(V1), green(V2).
11 :- coloring_vertex(V1), coloring_vertex(V2),
12        coloring_edge(V1, V2), blue(V1), blue(V2).
```

In order to compute the answer sets of the complete program, by the splitting set theorem, we only have to compute the answer sets of $T_S(P_{col}) \cup pr(LM(B_S(P_{col})))$. A simple (but computationally expensive) way to do this would be to perform an exhaustive search over all combinations of ground instances of the rules from lines 2 to 4 and drop all model candidates where any of the constraints fire. We finally get the following answer sets (filtered for predicates $red/1$, $green/1$ and $blue/1$):

$$\begin{aligned} A_1 &= \{blue(c),\ green(b),\ red(a)\}\\ A_2 &= \{blue(b),\ green(c),\ red(a)\}\\ A_3 &= \{blue(b),\ green(a),\ red(c)\}\\ A_4 &= \{blue(c),\ green(a),\ red(b)\}\\ A_5 &= \{blue(a),\ green(b),\ red(c)\}\\ A_6 &= \{blue(a),\ green(c),\ red(b)\} \end{aligned}$$

Expanding on Example 2.2.4, it is clear that a stratifiable bottom with respect to some splitting set always exists - every program has at least the empty set as a trivial splitting set. We call the maximal stratifiable sub-program that only depends on facts (i.e. is a bottom w.r.t some splitting set) Common Base Program (CBP), see Definition 2.2.6.

Definition 2.2.6 (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.

What remains is to solve the remaining, non-stratifiable part of a program. State-of-the-art two-phased solvers such as Clingo employ solving techniques based on Conflict-driven Nogood Learning (CDNL) for this [GKS12]. The basic idea of CDNL is to encode a ground program as a set of constraints called *nogoods*, i.e. sets of literals that must not be contained in any answer set. Based on these nogoods, truth assignments over $H_B(P)$ are "guessed" and conflicting guesses are used to potentially learn new nogoods using techniques originating in SAT-solving. However, these techniques normally require an input program to be fully ground, and therefore do not lend themselves to lazy-grounding approaches. Alpha uses a solving approach that is inspired by CDNL, but makes a number of modifications to apply the same principles in a lazy-grounding scenario. Section 2.2.2 gives a highlevel overview of how answer set search is realized in Alpha.

2.2.2 Lazy-grounding answer set search

Just like two-phased solvers, Alpha also uses *nogoods* to encode rules and constraints. However, Alpha does not have a full grounding of the program it solves, which necessitates a number of modifications. Without a full grounding, a solver must be able to distinguish between atoms in an assignment (i.e. answer set candidate) that are *true because some rule fired*, and atoms that *must be true because some constraint does not permit otherwise* (but haven't been derived by a rule). Lazy-grounding solvers introduce a third truth value, *must-be-true* (MBT), to reflect this. For an assignment to be an answer set, it must not contain any MBT values. Definitions 2.2.7 and 2.2.8 formally describe nogoods and their use to represent rules in Alpha.

Definition 2.2.7 (Alpha-Assignment, Nogood [Wei17])**.** An *Alpha-Assignment*, in the following just *assignment* is a sequence $A = \{sl1, \ldots, sl_n\}$ of literals $l_1 \ldots, l_n$ preceded by a *sign* $s \in \{T, M, F\}$. Intuitively, a signed literal sl_i assigns the truth value s to literal l. Possible truth values are *true* (T), *false* (F) and *must-be-true* (M). Must-be-true is used as truth value for literals that are deemed by the solver to necessarily be true (e.g. because of a constraint precluding any other truth value), but there is no known firing rule (yet) which would allow assigning T. The *boolean projection* $A^{\mathcal{B}}$ of an assignment A is defined as:

$$A^{\mathcal{B}} = \{Ta \mid Ta \in A \vee Ma \in A\} \cup \{Fa \mid Fa \in A\}$$

A *nogood*, in the variant used by Alpha, is a set of signed literals intuitively characterizing a set of literals that may not occur in an assignment with the given truth values:

$$ng = \{sl_1, \ldots, sl_n\}, s \in \{T, F\}$$

Different from how nogoods are defined in classic two-phased solvers, a nogood in Alpha may optionally have a specifically designated "head" literal. Given a nogood where sl_i is the head literal, we denote this as: $ng = \{sl_1, \ldots, sl_n\}_i$. For a signed literal sl, we denote the literal with inverse sign as $\overline{sl}$. An assignment A *satisfies* a nogood ng if $ng \nsubseteq A^{\mathcal{B}}$. Furthermore, an assignment A is a *solution* to a set Δ of nogoods if $\{a \mid Ta \in A^{\mathcal{B}}\} \cap \{a \mid Fa \in A^{\mathcal{B}}\} = \emptyset$, $\{a \mid Ta \in A^{\mathcal{B}}\} \cup \{a \mid Fa \in A^{\mathcal{B}}\} = A^{\mathcal{B}}$ and $\forall\, ng \in \Delta : ng \nsubseteq A^{\mathcal{B}}$.

Definition 2.2.8 (Nogood representation of rules [Wei17])**.** In order to represent a ground body of a rule, Alpha introduces new atoms, denoted $\beta(r, \sigma)$ for a rule r and variable substitution σ.
Given a rule r with head $h(r) = a_0$, positive body $b^+(r) = a_1, \ldots, a_k$ and negative body $b^-(r) = a_{k+1}, \ldots, a_n$, the *nogood representation* of r, $ng(r)$ is defined as follows:

$$\begin{aligned} ng(r) = \{&\{F\beta(r,\sigma), Ta_1, \ldots, Ta_k, Fa_{k+1}, \ldots, Fa_n\}_1, \{Fa_0, T\beta(r,\sigma)\}, \\ &\{T\beta(r,\sigma), Fa_1\}, \ldots, \{T\beta(r,\sigma), Fa_k\}, \\ &\{T\beta(r,\sigma), Ta_{k+1}\}, \ldots, \{T\beta(r,\sigma), Ta_n\}\} \end{aligned}$$

Lazy Grounding and solving in Alpha [LW17] Alpha's grounder always calculates ground substitutions for all rules that are *applicable* with respect to a (partial) assignment A. A rule r is said to be applicable w.r.t A, if, given a ground substitution σ $b^+(r\sigma) \subseteq A^{\mathcal{B}}$ and $b^-(r\sigma) \cap \{a \mid Ta \in A^{\mathcal{B}}\} = \emptyset$, i.e. a ground rule is applicable, if its positive body is satisifed and its negative body is not contradicted by an assignment. When solving a program, Alpha's grounder constructs an initial partial assignment from all facts and the model of the previously solved CBP. Based on this partial assignment, ground instances for applicable rules are computed and translated into nogoods. The partial assignment and generated nogoods are passed to the solver. The solver then first applies propagation, i.e. extends assignment A based on unit nogoods as described in Definition 2.2.9. Then, if a conflict between the (possibly extended) assignment and the current set of nogoods exists, the conflict is analyzed, a new nogood describing the conflict cause is added, and the solver *backtracks* (i.e. retracts enough of the assignment to get to a conflict-free state). If there are no conflicts, and applicable rules exist, the solver guesses (based on a heuristic), which rule to apply. If there are no applicable rules or unassigned atoms, and no atom is assigned *must-be-true*, the solver has found an answer set. In this case, the answer set is stored, a new nogood which prevents finding the same answer set again is added, and the solver backtracks. Algorithm 2.1 gives a highlevel summary of Alpha's answer set search procedure.

Definition 2.2.9 (Weak and strongly unit nogoods [Wei17])**.** Let ng be a nogood $ng = \{sl_1, \ldots, s_n\}$ and A an assignment. Then

- ng is *weakly unit* under A for sl if $ng \setminus A^{\mathcal{B}} = \{sl\}$ and $\overline{sl} \notin A^{\mathcal{B}}$
- ng is *strongly unit* under A for sl if sl is designated the "head" of ng, $ng \setminus A = \{sl\}$ and $\overline{sl} \notin A$

Intuitively, if a nogood is unit under an assignment A, it forces a truth value for the single literal sl which is not yet assigned (as otherwise, the nogood would be violated). In Alpha, the notions of weak and strong unitness are used to distinguish between nogoods forcing a literal to be set to *must-be-true* (if implied by a weakly unit nogood) and *true* (if implied by a strongly unit nogood)

Algorithm 2.1: Alpha's answer set search procedure [LW17]

Input : A non-ground program P
Output : The answer sets $AS(P)$ of P

Iniitialize $AS = \emptyset$, assignment A, nogood storage Δ.
Run lazy grounder, obtain initial nogoods Δ from facts.
while *search space not exhausted* **do**
    Propagate on Δ, extending A.
    if *conflicting nogood exists* **then**
        Analyze conflict, learn new nogood, backtrack.
    else if *propagation extended* A **then**
        Run lazy grounder w.r.t A to obtain new nogoods to extend Δ.
    else if *applicable rule exists* **then**
        Guess rule to fire according to heuristic.
    else if *unassigned atom exists* **then**
        Assign all unassigned atoms to false.
    else if *no atom assigned must-be-true in* A **then**
        $AS \leftarrow AS \cup \{A\}$ Answer set found, add enumeration nogood and backtrack.
    else
        Backtrack.
    end
end
return AS

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!

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*.

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 respective application rule $gr_{a_{app}}$ is contained in an answer set of P.

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

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

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 on LLM-based AI agents 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.

maybe move this section to preliminaries

cite alpha paper here

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:

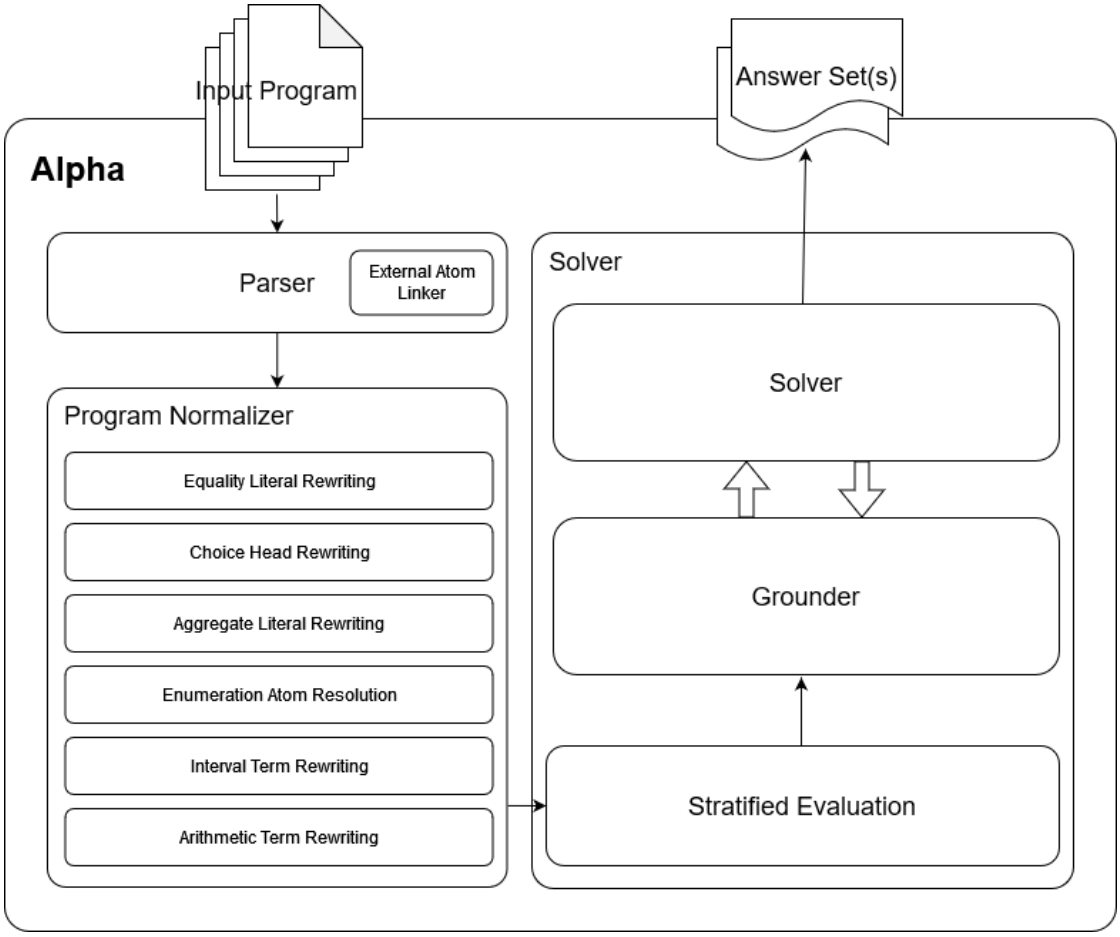

Figure 4.1: Alpha System Architecture

- *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 [Cona], 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

Once a program has been parsed and compiled using the steps described in 4.1.1, the compiled program can be evaluated (i.e. solved). Evaluation consists of two steps - first, the stratified bottom (*"common base program, CBP"*, see 2.2.6) is evaluated using a bottom-up evaluation algorithm based on iterative fixpoint calculation as described in Definition 2.2.3. The remainder of the program, i.e. the part which cannot be handlud using stratified evaluation is then solved using Alpha's CDNL-based answer set search described in Section 2.2.2. Since the stratified evaluation component is where Evolog Actions are handled, Section 4.1.2 gives an overview of how stratified evaluation is implemented in Alpha.

Stratified Evaluation in Alpha [Lan19]

Alpha's stratified evaluation component takes a normalized program as described in Section 4.1.1 as its input. A dependency graph for the program in question is calculated based on predicate dependencies based on Definition 2.2.5 where each dependency is labelled either "+" or "-" to distinguish dependencies through positive and negative body literals, respectively. Calculating the component graph (i.e. the graph resulting from condensing the dependency graph into its stronlgy connected components), one ends up with a directed acyclic graph, which is labelled such that all component nodes containing cycles through negative (i.e. labelled "-") dependencies get labelled as "unstratifiable". The CBP is then the program consisting of all rules from components that are not reachable by any path containing an unstratifiable component node. Based on the resulting subset of the component graph, a partition satisfying the criteria for a stratification according to Definition 2.2.2 is calculated.

The stratified part of the input program is then evaluated in order of ascending stratum using Algorithm 4.1. For each stratum, we first calculate applicable ground rules based

on all facts in the program as well as those derived when evaluating lower strata. As long as rule application yields new atoms, additional ground rules are computed and evaluated, until no new information can be derived, in which case a fixpoint has been reached on the current stratum. The result of the evaluation procedure is a program consisting of the rules of the non-stratified part of the input program, plus original facts as well as all new facts dervied during stratified evaluation.

Algorithm 4.1: Stratified up-front evaluation of CBP in Alpha

Input : Stratification $S = \{S_0, \ldots, S_n\}$
Input : Input Program $P = (F_{in}, R_{in})$ consisting of facts and rules F_{in}, R_{in}
Output : partially evaluated program P_{eval}

Facts $F_{out} = facts(F_{in})$
foreach *stratum* S_i **in** $S_0 \ldots S_n$ **do**
  initialize derived atoms in stratum $A = F_{out}$
  initialize derived atoms in individual run $A_{new} = A$
  do
    $A \leftarrow A \cup A_{new}$
    $A_{new} = \emptyset$
    calculate applicable ground rules R_{app} in P_{S_i} based on A
    foreach *rule* $r \in R_{app}$ **do**
      $A_{new} \leftarrow A_{new} \cup fireRule(r)$
    end
  while *new atoms derived in last run, i.e.* $A_{new} \neq A$;
  add newly derived atoms to output facts, $F_{out} \leftarrow F_{out} \cup A$
end
return $P_{eval} = (F_{out}, (P \setminus S))$

4.2 Implementing the Evolog extension in Alpha

After the technical overview of Alpha in Section 4.1, we now turn to how Alpha has been adapted to implement the Evolog language extension specified in Chapter 3. Specifically, additions were made in the following areas:

- *Parsing* - Alpha's program parser has been extended to be able to handle rules with action heads, module definitions, and references to modules from rule bodies.
- *Aggregate Rewriting* - In order to accomodate *list aggregation literals*, a concise construction syntax for list terms, additional logic to compile this new type of aggregate literal has been added.
- *Action Execution Service* - A new component responsible for executing actions has been added. It ensures idempotency of action rule application and serves as an

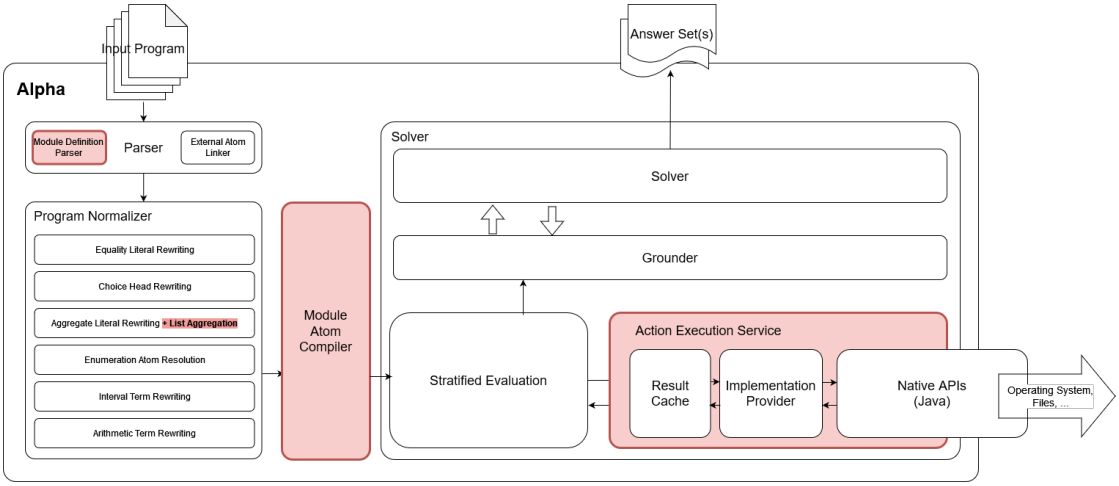

Figure 4.2: Alpha for Evolog System Architecture

abstraction layer between actual action implementations and Alpha's stratified evaluation component.

- *Module Atom Compilation* - A new program preprocessing component has been added in order to compile module atoms, i.e. link name-based references to modules with the actual implementation code.

Figure 4.2 shows an updated overview of Alpha's architecture, with new components highlighted.

4.2.1 Implementing Action Support

In this section, we give a detailed description of how Action support, conforming to definitions from Section 3.1 is implemented in Alpha.

Implemented Actions The actual actions provided in the reference implementation are intended to be sufficient for basic file (i.e. stream-)based input- and output operations. Specifically, Alpha offers actions to

- open a file handle for reading, i.e. obtaining an input stream,
- read from an input stream,
- close an input stream
- open a file handle for writing, i.e. obtaining an output stream,
- write to an output stream

- close an output stream

The idea of this selection of actions is that these can be used as the basic building blocks from which many more complex IO-Tasks, such as parsing specific file formats, waiting for user input, implementing command shells, etc. can be composed.

Supported Programs

As stated in Section 3.1.3, Evolog programs are required to be *transparent* in terms of action execution, i.e. for every action that gets executed, there must be a corresponding witnessing atom in the respective answer set. This requirement is formalized in Definition 3.1.8. While the formal definition only states that if an action rule fires (i.e. an action is executed), the corresponding head must be contained in an answer set - which, aside from the effect that is applied on the outside world, is not any different from the general meaning of a rule that, if a the body is true, the head is as well - this leads directly to the practical problem of how to enforce this in an implementation. Since actions taken in the "outside world" (such as sending data over a network interface, writing into a file, ending a process, etc.) can not be retracted, an Evolog interpreter must be able to establish action rule transparency *prior to actual execution*. In general, any answer set search algorithm based on guesses and propagation through nogoods as described in Section 2.2.2 does not know up-front whether an atom that gets assigned as true at some point in search, will be true in a resulting answer set - every later propagation step could lead to a violation of one or more nogoods and subsequent backtracking. With that in mind, we establish that it is not feasible to have any form of guesses depend on the result of an action, since whenever a solver makes a guess, there is a chance the program might be unsatisfiable, in which case the action would not be any transparent anymore. While it may certainly possible to identify classes of programs with guesses where action transparency can be assured through clever static analysis, our implementation takes a simple approach. In the reference implementation based on Alpha, an Evolog program P is supported, if and only if $P = cbp(e)$, using the notion of the common base program as given in Definition 2.2.6. Since the CBP of a program is by definition stratified, we can be sure there will be an answer set, and therefore have a guarantee of action transparency as well. At the same time, programs using the guess-and-check pattern can still be executed under this restriction, by encapsulating the guess-and-check part in a module. Consider the program from Example 3.2.2, which has a single answer set, in which individual atoms correspond to answer sets of the used modules. In a program structured like that, one can use modules to encapsulate arbitrarily complicated calculations with any number of guesses while maintaining stratifiability of the top-level program. Example 4.2.1 demonstrates an unsupported program as well as a supported version of it based on a short interactive application.

Example 4.2.1. The program in Listing 4.5 is intended to be a very short "greeter" application. The user is asked to enter their name, which is then prependen with "Hello" and printed to the terminal. However, the constraints `:- usr_input_res(success(line(""))).`

and `:- usr_input_res(error(_)).` render the program *unsupported*. In fact, the program is unsatisfiable with respect to any Frame (see Definition 3.1.3) where the user input is either empty or reading from `stdin` results in an error. Since, however, in order to evaluate the constraints in question, two actions with side-effects have to be performed first - namely one line needs to be written to `stdout` and one read from `stdin` - we would inevitably end up in a state where some side-effects have been applied to the outside world, but no corresponding action witness in an answer set exists.

Listing 4.5: An unsupported "greeter" application.

```
1  prompt_text("Hello user, enter your name: ").
2
3  write_prompt_res(R) : @streamWrite[STDOUT, PROMPT] = R :-
4      prompt_text(PROMPT), &stdout(STDOUT).
5  usr_input_res(INP) : @streamReadLine[STDIN] = INP :-
6      write_prompt_res(success(_)), &stdin(STDIN).
7
8  :- usr_input_res(success(line(""))).
9  :- usr_input_res(error(_)).
10
11 write_greeting_res(R) : @streamWrite[STDOUT, GREETING] = R
       :-
12     usr_input_res(success(line(TEXT)), _),
13     &stdlib_string_concat["Hello ", TEXT](GREETING).
14     &stdout(STDOUT).
```

Listing 4.6 shows a supported version of the program from Listing 4.5. Here, rather than constraints, regular rules are used for error checking, and in case of an error, a corresponding message is written to `stdout`. In this version, the program might not run in the "intended" fashion, but there will always be an answer set, i.e. effects the program applies on the outside world will always be made transparent.

This suggests a definition of transparency based on being satisfiable in all frames

Listing 4.6: A supported version of the "greeter" application.

```
1  prompt_text("Hello user, enter your name: ").
2
3  write_prompt_res(R) : @streamWrite[STDOUT, PROMPT] = R :-
4      prompt_text(PROMPT), &stdout(STDOUT).
5  usr_input_res(INP) : @streamReadLine[STDIN] = INP :-
6      write_prompt_res(success(_)), &stdin(STDIN).
7
8  error("Input String empty") :-
       usr_input_res(success(line(""))).
9  error(MSG):- usr_input_res(error(MSG)).
10 error_occurred :- error(_).
11
```

```
12      write_greeting_res(R) : @streamWrite[STDOUT, GREETING] = R
           :-
13          usr_input_res(success(line(TEXT)), _), not
               error_occurred.
14          &stdlib_string_concat["Hello ", TEXT](GREETING),
15          &stdout(STDOUT).
16
17      write_errmsg_res(R) : @streamWrite[STDOUT, ERRMSG] = R :-
18          error(ERR),
19          &stdlib_string_concat["An error occurred: ",
               ERR](ERRMSG),
20          &stdout(STDOUT).
```

Action Implementation

NOTE: this is just a basic write-up about how it works, not the final text

Action execution in Alpha is tied into the stratified evaluation component described in Section 4.1.2. Specifically, whenever a ground instance of an action rule is fired, the corresponding action function is evaluated. The expansion of action rules described in Definition 3.1.2 happens implicitly: Instead of actually splitting a rule with an action head into an application- and projection-rule as described in the formal definition, the implementation constructs an internal representation of the ground rule in question, which is equivalent to an *action application term* $f_{act}(S, I)$ (see Definition 3.1.2) in that it uniquely identifies a ground instance of an action rule within a program. After calculating the application for the ground rule to be fired, Alpha's action execution component queries an in-memory table called *action record* for existing entries where the key is equal to the calculated action application term. If such a mapping exists, the cached action result is returned, otherwise, the java method associated with the action function is executed, and a new mapping is added to the action record. The action record, i.e. action result cache, is used to make sure actions only get executed once, regardless of how often a certain ground instance of a rule is "fired" during program evaluation. Listing 4.7 shows an (abbreviated) version of Alpha's action execution component.

Listing 4.7: Alpha's action execution logic

```
public class ActionExecutionServiceImpl implements
   ActionExecutionService {

        private final ActionImplementationProvider
           actionProvider;
        private final Map<ActionInput, ActionWitness>
           actionRecord = new HashMap<>();
```

```
        // [Constructors and utility methods omitted for
           brevity]

        @Override
        public ActionWitness execute(String actionName, int
           sourceRuleId, Substitution sourceRuleInstance,
           List<Term> inputTerms) {
                ActionInput actInput = new
                   ActionInput(actionName, sourceRuleId,
                   sourceRuleInstance, inputTerms);
                return actionRecord.computeIfAbsent(actInput,
                   this::execute);
        }

        private ActionWitness execute(ActionInput input) {
                Action action =
                   actionProvider.getSupportedActions().get(input.name);
                ActionResultTerm<?> result =
                   action.execute(input.inputTerms);
                return new ActionWitness(input.sourceRule,
                   input.instance, input.name,
                   input.inputTerms, result);
        }

        private static class ActionInput {

                private final String name;
                private final int sourceRule;
                private final Substitution instance;
                private final List<Term> inputTerms;

        // [Constructors and utility methods omitted for
           brevity]

        }

}
```

Example 4.2.2 demonstrates a program where actions are used in recursive rules where ground instances potentially have to be evaluated multiple times and the aforementioned caching of action results comes into play.

Example 4.2.2. Listing 4.8 shows the "echo" program. It asks the user to enter text on the command line, and "echoes", i.e. prints, the previous input until the user enters

`EXIT`, at which point the program terminates. The "loop-like" behavior is realized through a positive recursive dependency cycle over predicates `write_prompt_res/2`, `usr_input_res/2` and `write_echo_res/2`.

Listing 4.8: An "echo" application which echoes user input written using Evolog actions.

```
1  prompt_text("Hello user, tell me something: ").
2  cancel_cmd("EXIT").
3
4  write_prompt_res(R, 0) : @streamWrite[STDOUT, PROMPT] = R :-
5      prompt_text(PROMPT), &stdout(STDOUT).
6  write_prompt_res(R, N) : @streamWrite[STDOUT, PROMPT] = R :-
7      prompt_text(PROMPT), write_echo_res(success(_), K),
8      N = K + 1, &stdout(STDOUT).
9  usr_input_res(INP, N) : @streamReadLine[STDIN] = INP :-
10     write_prompt_res(success(_), N), &stdin(STDIN).
11 write_echo_res(R, N) : @streamWrite[STDOUT, ECHO] = R :-
12     usr_input_res(success(line(TEXT)), N), TEXT != CANCEL_CMD,
13     cancel_cmd(CANCEL_CMD), &stdout(STDOUT),
14     &stdlib_string_concat["You said: ", TEXT](ECHO).
15
16 write_goodbye_res(R) : @streamWrite[STDOUT, MSG] = R :-
17     usr_input_res(success(line(TEXT)), _), cancel_cmd(CMD),
18     TEXT = CMD, MSG = "Goodbye, user! Sad to see you leave",
19     &stdout(STDOUT).
```

Action Result Terms The way action results are represented in Alpha takes its inspiration from the `Either` type present in many functional programming languages or libraries such as Vavr [Conb]. For example, an instance of type `Either<InputStream, Exception>` can either hold an instance of the "left" type, i.e. `InputStream` or an instance of the "right" type, i.e. `Exception`. Similarly, *action result terms* as defined in Definition 3.1.1 are function terms of form $r_t(r_v)$, where $r_t \in \{success, error\}$ is the *result type* and r_v the *result value*. Result type *success* indicates the action completed normally, and in this case, r_v holds the result of the successful action, e.g. a term representing an input stream. If the result type is *error*, for example when trying to obtain a read handle on a non-existing file, r_v typically holds an error message. Listing 4.9 shows the Java Interface specifying an Action result term in Alpha.

Listing 4.9: Java definition of an action result term.

```
public interface ActionResultTerm<T extends Term> extends
    FunctionTerm {
     public static final String SUCCESS_SYMBOL = "success";
     public static final String ERROR_SYMBOL = "error";
     /**
```

```
         * True if the action that generated this
         * result was successful (i.e. executed normally).
         */
        boolean isSuccess();
        /**
         * True if the action that generated this
         * result failed (i.e. threw an error in execution).
         */
        boolean isError();
        /**
         * Gets the actual value wrapped in this result.
         * Either a term representing the action return
         * value or a string term representing an error
         * message.
         */
        T getValue();
    }
```

Example 4.2.3. In this example, we take a detailed look at an evolog application intended to simulate possible real-world uses. We assume that graphs are stored as XML files, and construct an application that uses Evolog modules and actions to

- Prompt the user for a file path to read from using actions
- Read the content of the given file using actions
- Parse the resulting XML strings into a DOM-like representation using a generic module
- Use another generic module to calculate three-colorings of the graphs read from the input file
- Use actions to write the calculated colorings to an output file

I'm not sure where to best put this - maybe it should go to chapter 5.

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

Listing 4.10: 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 4.10, 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 4.10.

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 4.11 shows a much shorter version of the program from Listing 4.10, in which list aggregation is used instead of step-by-step construction of list terms.

Listing 4.11: 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 4.12 shows the rewritten version of the program form Listing 4.11 (Note that in the hand-crafted example from Listing 4.10, lists are constructed in descending element order, rather than ascending as is the case for the aggregate-based version).

Listing 4.12: Rewritten list aggregates from Listing 4.11

```
1  vertex_lst(LIST) :-
2      list_1_result(list_1_no_args, LIST).
3  edge_lst(LIST) :-
4      list_2_result(list_2_no_args, LIST).
5  list_1_element_greater(ARGS, N, K) :-
6      list_1_element(ARGS, N),
7      list_1_element(ARGS, K),
8      N > K.
9  list_1_element_not_successor(ARGS, N, K) :-
10     list_1_element_greater(ARGS, N, I),
11     list_1_element_greater(ARGS, I, K).
12 list_1_element_successor(ARGS, N, K) :-
13     list_1_element_greater(ARGS, N, K),
14     not list_1_element_not_successor(ARGS, N, K).
15 list_1_element_has_successor(ARGS, N) :-
16     list_1_element_successor(ARGS, _0, N).
17 list_1_lst_element(ARGS, IDX, lst(N, lst_empty)) :-
18     list_1_element(ARGS, N),
19     IDX = 0,
20     not list_1_element_has_successor(ARGS, N).
21 list_1_lst_element(ARGS, IDX, lst(N, lst(K, TAIL))) :-
22     list_1_element(ARGS, N),
23      list_1_element_successor(ARGS, K, N),
24      list_1_lst_element(ARGS, PREV_IDX, lst(K, TAIL)),
25      IDX = PREV_IDX + 1.
26 list_1_has_next_element(ARGS, IDX) :-
27     list_1_lst_element(ARGS, IDX, _1),
28     NEXT_IDX = IDX + 1,
29     list_1_lst_element(ARGS, NEXT_IDX, _2).
30 list_1_result(ARGS, LIST) :-
31     list_1_lst_element(ARGS, IDX, LIST),
32     not list_1_has_next_element(ARGS, IDX).
33 list_1_element(list_1_no_args, V) :-
34     vertex(V).
35 list_2_element_greater(ARGS, N, K) :-
36     list_2_element(ARGS, N),
```

```
37      list_2_element(ARGS, K),
38      N > K.
39  list_2_element_not_successor(ARGS, N, K) :-
40      list_2_element_greater(ARGS, N, I),
41      list_2_element_greater(ARGS, I, K).
42  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 4.13 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 4.13: Java Interface for Predicate Interpretations

```
@FunctionalInterface
public interface PredicateInterpretation {

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

}
```

Constructing interpretations for Module Atoms Listing 4.14 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 4.14: 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;
}
```

CHAPTER 5

Results

In this chapter, we apply Alpha's Evolog extension to a larger example. In order to showcase the kind of applications that can be written in pure Evolog (without the need to resort to any additional programming language), we implement an application that:

- asks the user to enter a path to a file containing a graph represented in XML format,
- reads and parses the XML content of the given files,
- calculates 3-colorings of the graph from the XML file,
- and finally writes the list of obtained colorings to a user-supplied file path.

We first describe the program itself and then move on to a discussion of observations that can be made from the example.

5.1 Interactive XML-based Graph Coloring

TODO: Reference full program in appendix. Maybe lose a few words about overall structure again.

5.1.1 Reading files from user-supplied paths

First, the user is asked to enter a file path. The content of the given file is then read line-by-line and aggregateed into a list term. Listing 5.1 shows this part of the program.

Listing 5.1: Reading a file based on user input.

```
1  enter_input_prompt("Please enter a path to read graphs from:
       ").
2
3  write_input_prompt_res(R) : @streamWrite[STDOUT, PROMPT] = R :-
4      enter_input_prompt(PROMPT), &stdout(STDOUT).
5  usr_input_res(INP) : @streamReadLine[STDIN] = INP :-
6      write_input_prompt_res(success(_)), &stdin(STDIN).
7
8  infile(PATH) :- usr_input_res(success(line(PATH))).
9  infile_open(PATH, HD) : @fileInputStream[PATH] = HD :-
10     infile(PATH).
11
12 % Handle file opening error
13 error(io, MSG) :- infile_open(_, error(MSG)).
14
15 % Read all lines from infile
16 readline_result(PATH, 0, RES) :
17     @streamReadLine[STREAM] = RES :-
18     infile(PATH), infile_open(PATH, success(stream(STREAM))).
19 readline_result(PATH, LINE_NO, RES) :
20     @streamReadLine[STREAM] = RES :-
21     infile(PATH),
22     infile_open(PATH, success(stream(STREAM))),
23     readline_result(PATH, PREV_LINE_NO, PREV_LINE_RES),
24     PREV_LINE_RES != success(line(eof)),
25     LINE_NO = PREV_LINE_NO + 1.
26
27 % close stream after getting eof
28 infile_closed(PATH, RES) :
29     @inputStreamClose[STREAM] = RES :-
30     infile(PATH),
31     infile_open(PATH, success(stream(STREAM))),
32     readline_result(PATH, _, ok(eof)).
33
34 % Now create a list of content lines
35 file_lines(PATH, LST) :-
36     LST = #list{
37         line(LINE_NO, LINE) :
38             readline_result(PATH, LINE_NO,
                    success(line(LINE))),
39             LINE != eof },
40     infile(PATH).
```

The rule on line 3 derives the predicate `write_input_prompt_res/1`, which wraps the result of a stream output operation - which in this case writes a prompt text to stdout. Note that the *standard output stream* `STDOUT` does not need to be opened using an Evolog action, but is always available to an application. A reference to `STDOUT` can be obtained through the external atom `stdout/1`.

If writing the prompt text succeeds, i.e. in case the result term of the write action has structure `success(_)`, user input is read from the *standard input stream* `STDIN`. The result of the read operation is derived on line 5 using predicate `usr_input_res/1`. Note that, as with stdout, `STDIN` does not have to be opened, but is obtained using the external atom `stdin/1`.

In case the read operation succeeded, we attempt to open an input stream on the file entered by the user. The rule on line 9 derives the result of this operation in the `infile_open/2` predicate. The first term is the path of the respective file, the second is an action result term which either takes the form `error(MSG)` or holds a reference to a file input stream `STREAM` wrapped in a function term `success(stream(STREAM))`.

If opening succeeds, the openend file is read line-wise, using the rules on lines 16 and 19. Lines are numbered in order to keep positional information of file content. Reading stops once the result from the last `@streamReadLine` action is `success(line(eof))` where `eof` denotes *end-of-file*.

After all lines of the input file have been read successfully an `@inputStreamClose` action is used to close the input file (see line 28). Finally, all lines from a file are aggregated into a single list term using a `#list` aggregate in the rule on line 35.

5.1.2 Parsing XML data

Based on the list of lines in an input file, the contained XML data is parsed using a separate Evolog Module `xml_dom`, which is described in more detail in Section 5.1.3. Input files are expected to conform to the Document Type Definition (DTD) in Listing5.2.

Listing 5.2: DTD for graph XML files.

```
1  <!ELEMENT graph (vertices, edges)>
2  <!ATTLIST graph directed (true | false) #REQUIRED>
3
4  <!ELEMENT vertices (vertex+)>
5  <!ELEMENT vertex (#PCDATA)>
6
7  <!ELEMENT edges (edge+)>
8  <!ELEMENT edge (source, target)>
9  <!ELEMENT source (#PCDATA)>
10 <!ELEMENT target (#PCDATA)>
```

Listing 5.3 shows an example input file conforming to the DTD which describes the complete graph K_3.

Listing 5.3: The complete graph K_3 represented according to the DTD from Listing 5.2.

```
1  <graph directed="false">
2      <vertices>
3          <vertex>a</vertex>
4          <vertex>b</vertex>
5          <vertex>c</vertex>
6      </vertices>
7      <edges>
8          <edge>
9              <source>a</source>
10             <target>b</target>
11         </edge>
12         <edge>
13             <source>b</source>
14             <target>c</target>
15         </edge>
16         <edge>
17             <source>c</source>
18             <target>a</target>
19         </edge>
20     </edges>
21 </graph>
```

The XML-parsing module has 4 output terms which represent lists of Document Object Model (DOM) elements, parent-child relationships between DOM elements, text content of elements as well as attributes, respectively. Listing 5.4 shows the Evolog code to parse the XML content of an input file. Since each of the four result terms of the `xml_dom` module is a list term, in order to have one atom per list entry, each list needs to be "unwrapped". List terms are unwrapped by first deriving one "entry"-atom for every element of the list which holds the element itself along with all successor elements in the list (i.e. the "tail list"). Finally, the tail list of each entry atom is projected away using an additional rule, such that tehre is one atom for each element of the original list. In Listing 5.4, this unwrapping is performed for each output list of the XML parsing module from line 8 onward.

Listing 5.4: Parsing and consuming XML data in an Evolog program.

```
1  % Parse the string content of individual lines into a
2  % DOM-like representation
3  file_dom(PATH, dom(ELEMENTS, CHILDREN, TEXTS, ATTRIBUTES)) :-
4      file_lines(PATH, LINES),
5      #xml_dom[LINES](ELEMENTS, CHILDREN, TEXTS, ATTRIBUTES).
```

```
6
7  % Unwrap list elements from module output
8  dom_element_entry(NAME, ID, TAIL) :-
9      file_dom(_, dom(lst(dom_element(NAME, ID), TAIL), _, _,
           _)).
10 dom_element_entry(NAME, ID, TAIL) :-
11     dom_element_entry(_, _, lst(dom_element(NAME, ID), TAIL)).
12 dom_element(NAME, ID) :-
13     dom_element_entry(NAME, ID, _).
14
15 dom_element_child_entry(NAME, ID, CHILD_ID, TAIL) :-
16     file_dom(_,dom(_, lst(
17         dom_element_child(NAME, ID, CHILD_ID), TAIL), _, _)).
18 dom_element_child_entry(NAME, ID, CHILD_ID, TAIL) :-
19     dom_element_child_entry(_, _, _, lst(
20         dom_element_child(NAME, ID,CHILD_ID), TAIL)).
21 dom_element_child(NAME, ID, CHILD_ID) :-
22     dom_element_child_entry(NAME, ID, CHILD_ID, _).
23
24 dom_element_text_entry(NAME, ID, TEXT, TAIL) :-
25     file_dom(_,dom(_, _, lst
26     (dom_element_text(NAME, ID, TEXT), TAIL), _)).
27 dom_element_text_entry(NAME, ID, TEXT, TAIL) :-
28     dom_element_text_entry(_, _, _, lst(
29         dom_element_text(NAME, ID, TEXT), TAIL)).
30 dom_element_text(NAME, ID, TEXT) :-
31     dom_element_text_entry(NAME, ID, TEXT, _).
32
33 dom_element_attribute_entry(
34     NAME, ID, ATTR_NAME, ATTR_VALUE, TAIL) :-
35     file_dom(_,dom(_, _, _, lst(
36         dom_element_attribute(
37             NAME,ID, ATTR_NAME, ATTR_VALUE), TAIL))).
38 dom_element_attribute_entry(
39     NAME, ID, ATTR_NAME, ATTR_VALUE, TAIL) :-
40     dom_element_attribute_entry(_, _, _, lst(
41         dom_element_attribute(
42             NAME, ID, ATTR_NAME, ATTR_VALUE), TAIL)).
43 dom_element_attribute(NAME, ID, ATTR_NAME, ATTR_VALUE) :-
44     dom_element_attribute_entry(
45         NAME, ID, ATTR_NAME, ATTR_VALUE, _).
```

5.1.3 An XML-DOM parser in ASP

The module `xml_dom` which is used in Listing 5.4 implements a generic XML-parser that generates a DOM-representation based on external atoms for regex evaluation.

Input is encoded as a list of strings, each one representing a line of input. At first, all input is tokenized in order to find opening and closing XML tags. Listing 5.5 shows the Evolog code used for tokenization in the `xml_dom` module.

Listing 5.5: Tokenizing XML input.

```
1  token_regex(tag_open, "(<(\w+)( (\w+=\\"\w+\\"))*>)").
2  token_regex(tag_close, "(</\w+>)").
3  opening_tag_name_regex("<(\w+)( (\w+=\\"\w+\\"))*>").
4  closing_tag_name_regex("</(\w+)>").
5  attribute_regex("(\w+=\\"\w+\\")").
6  attribute_name_regex("(\w+)=\\"\w+\\"").
7  attribute_value_regex("\w+=\\"(\w+)\\"").
8
9  % Unwrap lines
10 line_element(E, TAIL) :- input_lines(lst(E, TAIL)).
11 line_element(E, TAIL) :- line_element(_, lst(E, TAIL)).
12 line(LINE_NO, LINE) :- line_element(line(LINE_NO, LINE), _).
13
14 % Match basic tokens
15 token(t(TOK, VALUE), LINE_NO, FROM, TO) :-
16     line(LINE_NO, LINE), token_regex(TOK, REGEX),
17     &regex_matches[REGEX, LINE](VALUE,FROM, TO).
```

Regular expressions to match opening and closing XML tags are represented as facts the predicate `token_regex/2`. The external atom `regex_matches/5` is used to determine whether a line matches a token regex, as well as the respective parts of the string where the match occurs.

Once all tokens denoting opening and closing tags - up until that point ignoring actual tag names and relative positions of tags to each other - the next step is extracting actual tag names and pairs of opening and closing tags of the same name. This part of the parsing process is demonstrated in Listing 5.6. First, a relative ordering of tokens is established using the `token_before/8` predicate, which is derived by the rules on lines 2 through 11. Furthermore, we extract the name of each tag by using a variant of the tag open/close regexes with capture groups such that the first group is only the actual name of the tag, see lines 14 to 29. A matching pair of opening and closing tags in a well-formed XML document is characterized as an opening and a closing tag of the same name, with no opening tag of that name in between. In order to find these pairs, we first derive all matching pairs which do have another opening tag between them as instances of predicate `tag_opening_between/7`. The actual tag pairs making up well-formed

XML elements are the derived using the rule on line 54 as those pairs of matching tags for which no corresponding instance of `tag_opening_between/7` exists.

Listing 5.6: Finding pairs of matching opening and closing XML tags.

```
1  % Establish a token ordering
2  token_before(TOK1, TOK1_LINE_NO, TOK1_FROM, TOK1_TO,
3          TOK2, TOK2_LINE_NO, TOK2_FROM, TOK2_TO) :-
4          token(TOK1, TOK1_LINE_NO, TOK1_FROM, TOK1_TO),
5          token(TOK2, TOK2_LINE_NO, TOK2_FROM, TOK2_TO),
6          TOK1_LINE_NO = TOK2_LINE_NO, TOK1_TO < TOK2_FROM.
7  token_before(TOK1, TOK1_LINE_NO, TOK1_FROM, TOK1_TO,
8          TOK2, TOK2_LINE_NO, TOK2_FROM, TOK2_TO) :-
9          token(TOK1, TOK1_LINE_NO, TOK1_FROM, TOK1_TO),
10         token(TOK2, TOK2_LINE_NO, TOK2_FROM, TOK2_TO),
11         TOK1_LINE_NO < TOK2_LINE_NO.
12
13 % Extract tag names
14 tag_opening(TAG_NAME, t(tag_open, TOK_VALUE),
15         LINE_NO, TAG_FROM, TAG_TO) :-
16         token(
17             t(tag_open, TOK_VALUE),
18             LINE_NO, TAG_FROM, TAG_TO),
19         &regex_matches
20         [TAG_NAME_REGEX, TOK_VALUE](TAG_NAME, _, _),
21         opening_tag_name_regex(TAG_NAME_REGEX).
22 tag_closing(TAG_NAME, t(tag_close, TOK_VALUE),
23         LINE_NO, TAG_FROM, TAG_TO) :-
24         token(
25             t(tag_close, TOK_VALUE),
26             LINE_NO, TAG_FROM, TAG_TO),
27         &regex_matches
28         [TAG_NAME_REGEX, TOK_VALUE](TAG_NAME, _, _),
29         closing_tag_name_regex(TAG_NAME_REGEX).
30
31 % We have an opening/closing tag pair if tag name is the same,
32 % opening tag is before closing tag,
33 % and no opening tag of same name between.
34 tag_opening_between(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
35         CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
36         tag_opening(TAG_NAME, OPEN_TOK,
37         OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
38         tag_closing(TAG_NAME, CLOSE_TOK,
39         CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
40         token(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
```

```
41          token(CLOSE_TOK, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
42          token(INTM_OPEN_TOK, INTM_OPEN_LINE_NO,
43          INTM_OPEN_FROM, INTM_OPEN_TO),
44          tag_opening(TAG_NAME, INTM_OPEN_TOK,
45          INTM_OPEN_LINE_NO, INTM_OPEN_FROM, INTM_OPEN_TO),
46          token_before(
47          OPEN_TOK, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
48          INTM_OPEN_TOK, INTM_OPEN_LINE_NO,
49          INTM_OPEN_FROM, INTM_OPEN_TO),
50          token_before(
51          INTM_OPEN_TOK, INTM_OPEN_LINE_NO, INTM_OPEN_FROM,
52          INTM_OPEN_TO, CLOSE_TOK, CLOSE_LINE_NO,
53          CLOSE_FROM, CLOSE_TO).
54  element(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
55          CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
56          tag_opening(TAG_NAME, OPEN_TOK,
57          OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
58          tag_closing(TAG_NAME, CLOSE_TOK,
59          CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
60          token(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
61          token(CLOSE_TOK, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
62          token_before(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM,
               OPEN_TO,
63          CLOSE_TOK, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
64          not tag_opening_between(
65          TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
66          CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO).
```

Having parsed all XML elements, the next step is to determine which of these are terminal elements in the sense that they only contain text, as opposed to elements which consist of further XML elements. This is achieved using the code form Listing 5.7.

Listing 5.7: Parsing details of XML elements

```
1  % Any string between two tags with no other opening
2  % or closing tags between is content
3  any_tag_opening_between(TAG_NAME, OPEN_LINE_NO, OPEN_FROM,
      OPEN_TO, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
4           element(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
               CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
5           token(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
6           token(CLOSE_TOK, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
7           token_before(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM,
               OPEN_TO, t(tag_open, INTM_TOK), INTM_LINE_NO,
               INTM_FROM, INTM_TO),
```

```
8         token_before(t(tag_open, INTM_TOK), INTM_LINE_NO,
              INTM_FROM, INTM_TO, CLOSE_TOK, CLOSE_LINE_NO,
              CLOSE_FROM, CLOSE_TO).
9  any_tag_closing_between(TAG_NAME, OPEN_LINE_NO, OPEN_FROM,
       OPEN_TO, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
10        element(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
              CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
11        token(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM, OPEN_TO),
12        token(CLOSE_TOK, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
13        token_before(OPEN_TOK, OPEN_LINE_NO, OPEN_FROM,
              OPEN_TO, t(tag_close, INTM_TOK), INTM_LINE_NO,
              INTM_FROM, INTM_TO),
14        token_before(t(tag_close, INTM_TOK), INTM_LINE_NO,
              INTM_FROM, INTM_TO, CLOSE_TOK, CLOSE_LINE_NO,
              CLOSE_FROM, CLOSE_TO).
15 any_tag_between(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
       CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
16        any_tag_opening_between(TAG_NAME, OPEN_LINE_NO,
              OPEN_FROM, OPEN_TO, CLOSE_LINE_NO, CLOSE_FROM,
              CLOSE_TO),
17        any_tag_closing_between(TAG_NAME, OPEN_LINE_NO,
              OPEN_FROM, OPEN_TO, CLOSE_LINE_NO, CLOSE_FROM,
              CLOSE_TO).
18 terminal(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
       CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO) :-
19        element(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
              CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
20        not any_tag_between(TAG_NAME, OPEN_LINE_NO, OPEN_FROM,
              OPEN_TO, CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO).
21 % Extract content between tags (currently assumes start and
       end tags to be on the same line)
22 element_text(ELEMENT_NAME, FROM_POS, TO_POS, TEXT) :-
23        terminal(ELEMENT_NAME, LINE_NO, OPEN_FROM, OPEN_TO,
              LINE_NO, CLOSE_FROM, CLOSE_TO),
24        line(LINE_NO, FULL_LINE),
25        &string_substring[FULL_LINE, OPEN_TO,
              CLOSE_FROM](TEXT),
26        FROM_POS = pos(LINE_NO, OPEN_FROM),
27        TO_POS = pos(LINE_NO, CLOSE_TO).
```

In Listing 5.7, the rule on line 18 derives whether an XML element is a *terminal* element in that is does not enclose any additional elements other than plain text. In order to determine this, we check if there are any opening or closing tags between the opening and

closing tags of the element in question, see lines 3 through 15. For all terminal elements, we parse the text content of each individual element by taking a substring of the line containing the opening and closing tags ranging from after the last character of the opening tag to before the first character of the closing tag. (Note that this implementation assumes that content within a tag does not contain newline characters in order to keep the code a bit shorter.)

Every XML tag can also have any number of attributes. The parsing logic for attributes is shown in Listing 5.8. First, attributes are detected by checking every string that has been matched to be an opening XML tag against regular expressions matching attribute name an values, see the rule on line 2. Attributes derived using this approach are then associated with the corresponding `element/7` instance through the predicate `element_attribute/8`.

Listing 5.8: Parsing XML attributes.

```
1  %% Extract attributes
2  tag_opening_attribute(TAG_NAME, t(tag_open, TOK_VALUE),
       LINE_NO, FROM, TO, attribute(NAME, VALUE)) :-
3          tag_opening(TAG_NAME, t(tag_open, TOK_VALUE), LINE_NO,
               FROM, TO),
4          &regex_matches[ATTRIBUTE_REGEX,
               TOK_VALUE](ATTRIBUTE_STR, _, _),
5          attribute_regex(ATTRIBUTE_REGEX),
6          &regex_matches[ATTRIBUTE_VALUE_REGEX,
               ATTRIBUTE_STR](VALUE, _, _),
7          attribute_value_regex(ATTRIBUTE_VALUE_REGEX),
8          &regex_matches[ATTRIBUTE_NAME_REGEX,
               ATTRIBUTE_STR](NAME, _, _),
9          attribute_name_regex(ATTRIBUTE_NAME_REGEX).
10
11 element_attribute(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
       CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO, attribute(NAME,
       VALUE)) :-
12         element(TAG_NAME, OPEN_LINE_NO, OPEN_FROM, OPEN_TO,
               CLOSE_LINE_NO, CLOSE_FROM, CLOSE_TO),
13         tag_opening_attribute(TAG_NAME, t(tag_open, _),
               OPEN_LINE_NO, OPEN_FROM, OPEN_TO, attribute(NAME,
               VALUE)).
```

The final step in actual parsing of the XML content is to derive the complete hierarchy of all XML tags, i.e. determine which tags are direct children of another. This is done using the code from Listing 5.9. First, we establish elements that enclose other elements (i.e. opening and closing tag of the enclosed element between opening and closing tags of the enclosing element) using the rule on line 4. An element e_1 is not the parent element of an

element e_2 enclosed by e_1 if there is a third element e_i which is enclosed by e_1, but also encloses e_2. If, for any element encosing another element such an intermediate element exists, the rule from line 18 derives this as an instance of `element_not_parent/6`. An element e_1 *is* the parent element of e_2 if it encloses e_2 and is not known to be not the parent. Element pairs fulfilling this definition are derived by the rule on line 22 and expressed as instances of `element_parent/6`.

Listing 5.9: Parsing XML attributes.

```
1  % Derive parent elements
2  % An element encloses another if the opening and closing tag
       of the enclosed
3  % element lie fully between opening and cloising tags of the
       enclosing element
4  element_encloses(ENC, ENC_POS_FROM, ENC_POS_TO, LOWER,
       LOWER_POS_FROM, LOWER_POS_TO) :-
5          element(LOWER, LOWER_OPEN_LINE_NO, LOWER_OPEN_FROM,
               LOWER_OPEN_TO, LOWER_CLOSE_LINE_NO,
               LOWER_CLOSE_FROM, LOWER_CLOSE_TO),
6          element(ENC, ENC_OPEN_LINE_NO, ENC_OPEN_FROM,
               ENC_OPEN_TO, ENC_CLOSE_LINE_NO, ENC_CLOSE_FROM,
               ENC_CLOSE_TO),
7          token(LOWER_OPEN_TOK, LOWER_OPEN_LINE_NO,
               LOWER_OPEN_FROM, LOWER_OPEN_TO),
8          token(LOWER_CLOSE_TOK, LOWER_CLOSE_LINE_NO,
               LOWER_CLOSE_FROM, LOWER_CLOSE_TO),
9          token(ENC_OPEN_TOK, ENC_OPEN_LINE_NO, ENC_OPEN_FROM,
               ENC_OPEN_TO),
10         token(ENC_CLOSE_TOK, ENC_CLOSE_LINE_NO,
               ENC_CLOSE_FROM, ENC_CLOSE_TO),
11         token_before(ENC_OPEN_TOK, ENC_OPEN_LINE_NO,
               ENC_OPEN_FROM, ENC_OPEN_TO, LOWER_OPEN_TOK,
               LOWER_OPEN_LINE_NO, LOWER_OPEN_FROM, LOWER_OPEN_TO),
12         token_before(LOWER_CLOSE_TOK, LOWER_CLOSE_LINE_NO,
               LOWER_CLOSE_FROM, LOWER_CLOSE_TO, ENC_CLOSE_TOK,
               ENC_CLOSE_LINE_NO, ENC_CLOSE_FROM, ENC_CLOSE_TO),
13         ENC_POS_FROM = pos(ENC_OPEN_LINE_NO, ENC_OPEN_FROM),
14         ENC_POS_TO = pos(ENC_CLOSE_LINE_NO, ENC_CLOSE_TO),
15         LOWER_POS_FROM = pos(LOWER_OPEN_LINE_NO,
               LOWER_OPEN_FROM),
16         LOWER_POS_TO = pos(LOWER_CLOSE_LINE_NO,
               LOWER_CLOSE_TO).
17 % An element E1 is parent of element E2 if E1 encloses E2 and
       there is no other element that encloses E2 and is enclosed
       by E1.
```

```
18 element_not_parent(E, E_POS_FROM, E_POS_TO, P, P_POS_FROM,
      P_POS_TO) :-
19         element_encloses(P, P_POS_FROM, P_POS_TO, E,
              E_POS_FROM, E_POS_TO),
20         element_encloses(P, P_POS_FROM, P_POS_TO, I,
              I_POS_FROM, I_POS_TO),
21         element_encloses(I, I_POS_FROM, I_POS_TO, E,
              E_POS_FROM, E_POS_TO).
22 element_parent(E, E_POS_FROM, E_POS_TO, P, P_POS_FROM,
      P_POS_TO) :-
23         element_encloses(P, P_POS_FROM, P_POS_TO, E,
              E_POS_FROM, E_POS_TO),
24         not element_not_parent(E, E_POS_FROM, E_POS_TO, P,
              P_POS_FROM, P_POS_TO).
```

Last but not least, the `xml_dom` module has a number of rules to project away positional information of tags which are not deemed to be of interest to the calling component. Listing 5.10 shows all rules used to achieve this. First, each parsed XML element is assigned a unique identifier represented using the predicate `dom_element_id/2` by the rule on line 1. Identifier terms are obtained from the *enumeration literal* (see Section 4.1.1) `id_enum/3`. Based on the thereby established relation, versions of the predicates `element/7`, `element_parent/6`, `element_text/4`, `element_attribute/8` in which the individual terms making up the respective element's position within the document are replaced with the corresponding identifier.

Listing 5.10: Deriving simplified representations of parsed XML content.

```
 1 dom_element_id(e(NAME, POS_FROM, POS_TO), ID) :-
 2         element(NAME, START_LINE_NO, START_FROM, START_TO,
              END_LINE_NO, END_FROM, END_TO),
 3         POS_FROM = pos(START_LINE_NO, START_FROM),
 4         POS_TO = pos(END_LINE_NO, END_TO),
 5         id_enum(element_ids, e(NAME, POS_FROM, POS_TO), ID).
 6
 7 dom_element(NAME, ID) :-
 8         dom_element_id(e(NAME, _, _), ID).
 9
10 dom_element_child(NAME, ID, CHILD_ID) :-
11         dom_element(NAME, ID),
12         dom_element_id(e(NAME, POS_FROM, POS_TO), ID),
13         dom_element_id(e(CHILD_NAME, CHILD_POS_FROM,
              CHILD_POS_TO), CHILD_ID),
14         element_parent(CHILD_NAME, CHILD_POS_FROM,
              CHILD_POS_TO, NAME, POS_FROM, POS_TO).
15
```

```
16 dom_element_text(NAME, ID, TEXT) :-
17         dom_element(NAME, ID),
18         dom_element_id(e(NAME, POS_FROM, POS_TO), ID),
19         element_text(NAME, POS_FROM, POS_TO, TEXT).
20
21 dom_element_attribute(NAME, ID, ATTR_NAME, ATTR_VALUE) :-
22         dom_element(NAME, ID),
23         dom_element_id(e(NAME, pos(START_LINE_NO, START_FROM),
               pos(END_LINE_NO, END_TO)), ID),
24         element_attribute(NAME, START_LINE_NO, START_FROM, _,
               END_LINE_NO, _, END_TO, attribute(ATTR_NAME,
               ATTR_VALUE)).
```

5.1.4 Extracting a graph representation from parsed XML data

Based on the DOM-elements unwrapped in Listing 5.4, a representation of a graph using predicates `graph/1`, `graph_vertex/2`, `graph_directedness/2` and `graph_edge/2` is constructed using the code from Listing 5.11. DOM elements are translated into vertices and edges based on their names. Whether the graph in question is directed or undirected can be extracted from the corrseponding attribute of the `graph` element.

Listing 5.11: Constructing a graph representation from parsed XML data.

```
1  % Translate generic DOM-elements into graphs
2  graph(ID) :- dom_element("graph", ID).
3  graph_vertex(ID, VERTEX) :-
4      graph(ID), dom_element("graph", ID),
           dom_element_child("graph", ID, VERTEX_LIST_ID),
5      dom_element("vertices", VERTEX_LIST_ID),
           dom_element("vertex", VERTEX_ID),
           dom_element_child("vertices", VERTEX_LIST_ID,
           VERTEX_ID),
6      dom_element_text("vertex", VERTEX_ID, VERTEX).
7  graph_directedness(ID, DIRECTED) :-
8      graph(ID), dom_element("graph", ID),
           dom_element_attribute("graph", ID, "directed",
           DIRECTED).
9  graph_undirected(ID) :- graph_directedness(ID, "false").
10 graph_directed(ID) :- graph_directedness(ID, "true").
11 graph_edge(ID, EDGE) :-
12     graph(ID), graph_vertex(ID, SRC_ID), graph_vertex(ID,
           TARGET_ID), dom_element("graph", ID),
           dom_element_child("graph", ID, EDGE_LIST_ID),
```

```
13      dom_element("edges", EDGE_LIST_ID), dom_element("edge",
           EDGE_ID), dom_element_child("edges", EDGE_LIST_ID,
           EDGE_ID),
14      dom_element("source", SRC_ELEM_ID),
           dom_element_child("edge", EDGE_ID, SRC_ELEM_ID),
           dom_element_text("source", SRC_ELEM_ID, SRC_ID),
15      dom_element("target", TARGET_ELEM_ID),
           dom_element_child("edge", EDGE_ID, TARGET_ELEM_ID),
           dom_element_text("target", TARGET_ELEM_ID, TARGET_ID),
16      EDGE = e(SRC_ID, TARGET_ID).
17  graph_edge(ID, e(TARGET, SRC)) :- graph_edge(ID, e(SRC,
       TARGET)), graph_undirected(ID).
```

5.1.5 Calculating and storing graph colorings

Colorings for the graph parsed in the previous sections can be calculated using module `threecol` introduced in Example 3.2.1. For later processing (i.e. serialization to XML content), each calculated coloring is assigned a unique ID using the enumeration predicate `ids/3`.

Listing 5.12: Calculating graph colorings.

```
1  graph_threecol(ID, COL) :-
2          graph(ID),
3          VERTEX_LIST = #list{V : graph_vertex(ID, V)},
4          EDGE_LIST = #list{edge(V1, V2) : graph_edge(ID, e(V1,
               V2))},
5          #threecol[VERTEX_LIST, EDGE_LIST](COL).
6
7  graph_coloring(GRAPH, COLORING, ID) :- ids(coloring_ids,
       gcol(GRAPH, COLORING), ID), graph_threecol(GRAPH, COLORING).
```

In order to write calculated colorings to an XML file, each coloring is first converted to an XML string using module `threecol_to_xml`, which is shown in Listing 5.13. The module takes as input an atom of predicate `coloring/1` which is assumed to have one argument term, where the argument is a list of terms representing vertices along with an assigned color, e.g. `coloring(lst(col(1, red), lst(col(2, blue), lst_empty)))`. In the listing, the rules on lines 2 through 8 are used to first "unwrap" the input list into one `coloring_entry/2` atom per list entry, an then establish a total ordering of said atoms using predicates `coloring_entry_pred/2`, `coloring_entry_first/1` and `coloring_entry_last/1`. Based on the derived ordering, an XML string representing the input coloring is built recursively starting from the "lowest" element. Every step of the construction process yields an instance of `coloring_xmlstr_upto/2` where the second argument term is the identifier of the

"highest" vertex encoded in the CML string contituting the first argument. The output predicate of the module `coloring_xmlstr/1` is derived from the "maximal" instance of `coloring_xmlstr_upto/2`, i.e. the one corresponding to the last coloring entry according to `coloring_entry_last/1`.

Listing 5.13: Generating XML strings for graph colorings.

```
1  #module threecol_to_xml (coloring/1 => {coloring_xmlstr/1}) {
2      coloring_entry(vertex(ID, COL), TAIL) :-
           coloring(lst(col(ID, COL), TAIL)).
3      coloring_entry(vertex(ID, COL), TAIL) :- coloring_entry(_,
           lst(col(ID, COL), TAIL)).
4      coloring_entry_pred(ID, ID_PRED) :-
           coloring_entry(vertex(ID, _), lst(col(ID_PRED, _), _)).
5      coloring_entry_has_pred(ID) :- coloring_entry_pred(ID, _).
6      coloring_entry_is_pred(ID) :- coloring_entry_pred(_, ID).
7      coloring_entry_first(ID) :- coloring_entry(vertex(ID, _),
           _), not coloring_entry_has_pred(ID).
8      coloring_entry_last(ID) :- coloring_entry(vertex(ID, _),
           _), not coloring_entry_is_pred(ID).
9
10     coloring_tag_open("<coloring>").
11     coloring_tag_close("</coloring>").
12
13     coloring_part(ID, STR) :-
14         coloring_entry(vertex(ID, COL), _),
15         &stdlib_string_concat["<vertex><id>", ID](S1),
16         &stdlib_string_concat[S1, "</id><color>"](S2),
17         &stdlib_string_concat[S2, COL](S3),
18         &stdlib_string_concat[S3, "</color></vertex>"](STR).
19
20     coloring_xmlstr_upto(STR, ID) :-
21         coloring_entry_first(ID),
22         coloring_part(ID, FIRST_PARTSTR),
23         coloring_tag_open(OPENSTR),
24         &stdlib_string_concat[OPENSTR, FIRST_PARTSTR](STR).
25
26     coloring_xmlstr_upto(STR, ID) :-
27         coloring_entry_pred(ID, ID_PRED),
28         coloring_xmlstr_upto(PREV_STR, ID_PRED),
29         coloring_part(ID, PARTSTR),
30         &stdlib_string_concat[PREV_STR, PARTSTR](STR).
31
32     coloring_xmlstr(STR) :-
33         coloring_entry_last(ID),
```

```
34          coloring_xmlstr_upto(PREV_STR, ID),
35          coloring_tag_close(CLOSESTR),
36          &stdlib_string_concat[PREV_STR, CLOSESTR](STR).
37  }
```

Listing 5.14 shows how module `threecol_to_xml` is used in the application to obtain an XML string for each calculated graph coloring.

Listing 5.14: Generating XML strings for graph colorings.

```
1  % Translate colorings into strings
2  stringified_coloring_lst(GRAPH, COL_ID, COLSTR_LST) :-
3      graph_coloring(GRAPH, COLS, COL_ID),
4      #threecol_to_xml[COLS](COLSTR_LST).
5  stringified_coloring_entry(GRAPH, COL_ID, COL_STR, TAIL) :-
6      stringified_coloring_lst(GRAPH, COL_ID,
          lst(coloring_xmlstr(COL_STR), TAIL)).
7  stringified_coloring_entry(GRAPH, COL_ID, COL_STR, TAIL) :-
8      stringified_coloring_entry(GRAPH, COL_ID, _,
          lst(coloring_xmlstr(COL_STR), TAIL)).
9  stringified_coloring(GRAPH, COL_ID, COL_STR) :-
      stringified_coloring_entry(GRAPH, COL_ID, COL_STR, _).
```

Concatenating the strings for each coloring of a respective graph into one XML string encoding all colorings found for the graph in question is done in much the same way as the string representations of individual colorings are constructed. Listing 5.15 illustrates this.

Listing 5.15: Concatenating XML strings of individual colorings.

```
1  % Stitch individual coloring strings into one string
2  stringified_coloring_before(GRAPH, COL_ID, COL_ID_AFTER) :-
3      stringified_coloring(GRAPH, COL_ID, _),
4      stringified_coloring(GRAPH, COL_ID_AFTER, _),
5      COL_ID_AFTER > COL_ID.
6  stringified_coloring_not_predecessor(GRAPH, COL_ID,
      COL_ID_AFTER) :-
7      stringified_coloring_before(GRAPH, COL_ID, COL_ID_INTM),
8      stringified_coloring_before(GRAPH, COL_ID_INTM,
          COL_ID_AFTER).
9  stringified_coloring_predecessor(GRAPH, COL_ID, COL_ID_AFTER)
      :-
10     stringified_coloring_before(GRAPH, COL_ID, COL_ID_AFTER),
11     not stringified_coloring_not_predecessor(GRAPH, COL_ID,
          COL_ID_AFTER).
```

```
12 stringified_coloring_has_predecessor(GRAPH, COL_ID) :-
13     stringified_coloring(GRAPH, COL_ID, _),
14     stringified_coloring_predecessor(GRAPH, _, COL_ID).
15 stringified_coloring_first(GRAPH, COL_ID) :-
16     stringified_coloring(GRAPH, COL_ID, _),
17     not stringified_coloring_has_predecessor(GRAPH,
           COL_ID).
18 stringified_coloring_has_successor(GRAPH, COL_ID) :-
19     stringified_coloring(GRAPH, COL_ID, _),
20     stringified_coloring_before(GRAPH, COL_ID, _).
21 stringified_coloring_last(GRAPH, COL_ID) :-
22     stringified_coloring(GRAPH, COL_ID, _),
23     not stringified_coloring_has_successor(GRAPH, COL_ID).
24
25 xmlstring_upto(GRAPH, STR, COL_ID_FIRST) :-
26     &stdlib_string_concat["<colorings>", COLSTR](STR),
27     stringified_coloring_first(GRAPH, COL_ID_FIRST),
28     stringified_coloring(GRAPH, COL_ID_FIRST, COLSTR).
29
30 xmlstring_upto(GRAPH, STR, COL_ID) :-
31     stringified_coloring_predecessor(GRAPH, COL_ID_PRED,
           COL_ID),
32     xmlstring_upto(GRAPH, PREV_STR, COL_ID_PRED),
33     stringified_coloring(GRAPH, COL_ID, COLSTR),
34     &stdlib_string_concat[PREV_STR, COLSTR](STR).
35
36 colorings_xmlstring(GRAPH, RESULT_STR) :-
37     stringified_coloring_last(GRAPH, COL_ID),
38     xmlstring_upto(GRAPH, STR, COL_ID),
39     &stdlib_string_concat[STR, "</colorings>"](RESULT_STR).
```

5.1.6 Writing calculated graph colorings to XML files

Once all colorings for the graph(s) from the input file have been calculated and transformed into XML representation, the last remaining step is to actually write those XML strings to files. The code used for this is shown in Listing 5.16. The way the user interaction is implemented is similar to how the input file path is obtained in Listing 5.1.

Listing 5.16: Writing Graph colorings to XML files.

```
1 % Prompt user for a path to which to write the calculated
     colorings for each graph
2 write_output_prompt_res(G, R) : @streamWrite[STDOUT, P] = R :-
3         graph(G), enter_output_prompt(PROMPT),
```

```
4         &stdlib_string_concat[PROMPT, G](S),
5         &stdlib_string_concat[S, ": "](P),
6         &stdout(STDOUT).
7 usr_output_res(G, INP) : @streamReadLine[STDIN] = INP :-
     write_output_prompt_res(G, success(_)), &stdin(STDIN).
8
9 % Open the output file
10 outfile(G, PATH) :- usr_output_res(G, success(line(PATH))).
11 outfile_open(G, PATH, HD) : @fileOutputStream[PATH] = HD :-
     outfile(G, PATH).
12
13 % Write colorings to output file
14 write_colorings_result(G, R) : @streamWrite[HD, STR] = R :-
15         colorings_xmlstring(G, STR),
16         outfile(G, PATH),
17         outfile_open(G, PATH, success(stream(HD))).
18
19 % Close the output file
20 outfile_closed(G, RES) : @outputStreamClose[STREAM] = RES :-
21         outfile(G, PATH), write_colorings_result(G,
              success(_)),
22         outfile_open(G, PATH, success(stream(STREAM))).
```

5.2 Usage of the graph coloring application

In this Section, we demonstrate a few example inputs for the graph coloring application along with the resulting output files and answer sets.

rephrase, it's actually just one due to space constraints.

Example 5.2.1 (Trivial Graph)**.** The simplest case on which to test our graph coloring application is a trivial graph consisting of one vertex. Listing 5.17 shows the corresponding input XML file.

Listing 5.17: One-vertex graph in XML encoding.

```
1 <graph directed="false">
2     <vertices>
3         <vertex>a</vertex>
4     </vertices>
5 </graph>
```

Listing 5.18 shows the solver call to run the XML-Graph-Coloring application for the input as well as all inputs and outputs during runtime.

Listing 5.18: Running the application for a one-vertex graph.

```
1 C:\...\evolog-thesis\evolog-samples\xml-graphcol> alpha-solver
     -i .\file-based-3col.evl -i .\xml_dom.mod.evl -i
     .\3col-module-with-lists.evl -i .\3col_to_xml.mod.evl
2 Please enter a path to read graphs from:
3 inputs/trivial.xml
4 Please enter a path to write calculated colorings for graph 1:
5 outputs/trivial.out.xml
6 Answer set 1:
7 { ... }
```

The (single) answer set A of the program is rather lengthy in its unfiltered form and is therefore given as the union of sets $A_{input}, A_{dom}, A_{col}, A_{string}, A_{output}$ in which we group atoms by functional area they relate to. We denote as A_{input} the set of atoms in A which are derived by rules dealing with obtaining user input (i.e. the path from which to read graph specifications):

$$
\begin{aligned}
A_{input} = \{ & enter\_input\_prompt(\text{"}Please\ enter\ a\ path\ to\ read\ graphs\ from:\ \text{"}),\\
& write\_input\_prompt\_res(success(ok)),\\
& usr\_input\_res(success(line(\text{"}inputs/trivial.xml\text{"}))),\ infile(\text{"}inputs/trivial.xml\text{"}),\\
& infile\_open(\text{"}inputs/trivial.xml\text{"}, success(stream(inputStream\_2))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 0,\ success(line(\text{"} < graph\ directed = \text{"}false\text{"} > \text{"}))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 1,\ success(line(\text{"} < vertices > \text{"}))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 2\, success(line(\text{"} < vertex > a < /vertex > \text{"}))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 3,\ success(line(\text{"} < /vertices > \text{"}))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 4,\ success(line(\text{"} < /graph > \text{"}))),\\
& readline\_result(\text{"}inputs/trivial.xml\text{"},\ 5,\ success(line(eof))),\\
& infile\_closed(\text{"}inputs/trivial.xml\text{"},\ success(closeResult(ok))),\\
& file\_lines(\text{"}inputs/trivial.xml\text{"},\\
& \quad lst(line(0,\ \text{"} < graph\ directed = \text{"}false\text{"} > \text{"}),\\
& \quad lst(line(1,\ \text{"} < vertices > \text{"}),\ lst(line(2,\ \text{"} < vertex > a < /vertex > \text{"}),\\
& \quad lst(line(3,\ \text{"} < /vertices > \text{"}),\ lst(line(4,\ \text{"} < /graph > \text{"}),\\
& \quad lst\_empty))))))\}
\end{aligned}
$$

Similarly, A_{dom} contains all atoms in the answer set which model the DOM representation of the input XML file. For brevity, we do not list every individual list term of the $file\_dom$ instance (since the atoms resulting from unwrapping the list are part of A_{dom} anyway). Also, helper atoms for partial list entries needed for unwrapping (such as

dom_element_entry) are not included:

$$
\begin{aligned}
A_{dom} = \{&dom\_element(\text{"}graph\text{"},\ 1),\\
&dom\_element(\text{"}vertex\text{"},\ 3),\\
&dom\_element(\text{"}vertices\text{"},\ 2),\\
&dom\_element\_attribute(\text{"}graph\text{"},\ 1,\ \text{"}directed\text{"},\ \text{"}false\text{"}),\\
&dom\_element\_child(\text{"}graph\text{"},\ 1,\ 2),\\
&dom\_element\_child(\text{"}vertices\text{"},\ 2,\ 3),\\
&dom\_element\_text(\text{"}vertex\text{"},\ 3,\ \text{"}a\text{"})\}
\end{aligned}
$$

The subset A_{col} holds atoms relating to the actual model of the graph as well as calculated colorings:

$$
\begin{aligned}
A_{col} = \{&graph(1),\ graph\_vertex(1,\ \text{"}a\text{"}),\ graph\_undirected(1),\\
&graph\_vertices(1,\ lst(\text{"}a\text{"},\ lst\_empty)),\ graph\_edges(1,\ lst\_empty),\\
&graph\_compact(1,\ lst(\text{"}a\text{"},\ lst\_empty),\ lst\_empty),\\
&graph\_coloring(1,\ lst(col(\text{"}a\text{"},\ blue),\ lst\_empty),\ 2),\\
&graph\_coloring(1,\ lst(col(\text{"}a\text{"},\ green),\ lst\_empty),\ 3),\\
&graph\_coloring(1,\ lst(col(\text{"}a\text{"},\ red),\ lst\_empty),\ 1)\}
\end{aligned}
$$

A_{string} holds all atoms derived by rules dealing with serializing our calculated colorings into a single XML string. Since there is a relatively high number of pure "helper" atoms

that are only used to impose orderings etc., this set is given in a much abbreviated form:

$$
\begin{aligned}
A_{string} = \{&\mathit{stringified\_coloring}(1,\ 1,\\
&\text{"}< \mathit{coloring} >< \mathit{vertex} >\\
&\quad < \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{red} < /\mathit{color} >< /\mathit{vertex} >< /\mathit{coloring} >\text{"}),\\
&\mathit{stringified\_coloring}(1,\ 2,\\
&\text{"}< \mathit{coloring} >< \mathit{vertex} >\\
&\quad < \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{blue} < /\mathit{color} >< /\mathit{vertex} >< /\mathit{coloring} >\text{"}),\\
&\mathit{stringified\_coloring}(1,\ 3,\\
&\text{"}< \mathit{coloring} >< \mathit{vertex} >\\
&\quad < \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{green} < /\mathit{color} >< /\mathit{vertex} >< /\mathit{coloring} >\text{"}),\\
&\mathit{colorings\_xmlstring}(1,\\
&\text{"}< \mathit{colorings} >\\
&\quad < \mathit{coloring} >\\
&\quad\quad < \mathit{vertex} >< \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{red} < /\mathit{color} >< /\mathit{vertex} >\\
&\quad < /\mathit{coloring} >\\
&\quad < \mathit{coloring} >\\
&\quad\quad < \mathit{vertex} >< \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{blue} < /\mathit{color} >< /\mathit{vertex} >\\
&\quad < /\mathit{coloring} >\\
&\quad < \mathit{coloring} >\\
&\quad\quad < \mathit{vertex} >< \mathit{id} > a < /\mathit{id} >< \mathit{color} > \mathit{green} < /\mathit{color} >< /\mathit{vertex} >\\
&\quad < /\mathit{coloring} >\\
&< /\mathit{colorings} >\text{")}\}
\end{aligned}
$$

Last but not least, A_{output} groups together all atoms derived while prompting users for output paths to write colorings to, as well as actually writing the output file(s):

$$
\begin{aligned}
A_{output} = \{&\mathit{write\_output\_prompt\_res}(1,\ \mathit{success}(\mathit{ok})),\\
&\mathit{usr\_output\_res}(1,\ \mathit{success}(\mathit{line}(\text{"}\mathit{outputs/trivial.out.xml}\text{"})))),\\
&\mathit{outfile}(1,\ \text{"}\mathit{outputs/trivial.out.xml}\text{"}),\\
&\mathit{outfile\_open}(1,\ \text{"}\mathit{outputs/trivial.out.xml}\text{"},\ \mathit{success}(\mathit{stream}(\mathit{outputStream\_3}))),\\
&\mathit{write\_colorings\_result}(1,\ \mathit{success}(\mathit{ok})),\\
&\mathit{outfile\_closed}(1,\ \mathit{success}(\mathit{ok}))\}
\end{aligned}
$$

The output file written to `trivial.out.xml` is given in Listing 5.19.

Listing 5.19: XML encoding of caluclated colorings for the input file from Listing 5.17

```
1 <colorings>
```

```
2         <coloring>
3                 <vertex>
4                         <id>a</id>
5                         <color>red</color>
6                 </vertex>
7         </coloring>
8         <coloring>
9                 <vertex>
10                        <id>a</id>
11                        <color>blue</color>
12                </vertex>
13        </coloring>
14        <coloring>
15                <vertex>
16                        <id>a</id>
17                        <color>green</color>
18                </vertex>
19        </coloring>
20</colorings>
```

5.3 Performance of solving Evolog programs

In order to gain some idea of Alpha's solving performance for Evolog programs, a slightly modified version of the graph coloring program has been used where all rules that read user input interactively (and therefore are blocking) were removed and instead facts were added for the `infile/1` and `outfile/2` predicates.

Alpha's solving performance for this modified version of the XML-based-3-coloring application has been benchmarked against calculating colorings for the same graphs represented as ASP facts and without writing calculated colorings to XML files. Listing 5.20 shows the "classic ASP" representation used for the complete graph with 3 vertices K_3.

Listing 5.20: Pure-ASP version of a 3-coloring encoding for the graph K_3.

```
1 vertex(1).
2 vertex(2).
3 vertex(3).
4
5 edge(1,2).
6 edge(2,3).
7 edge(3,1).
8
9 % Make sure edges are symmetric
10 edge(V2, V1) :- edge(V1, V2).
```

```
11
12 % Guess colors
13 red(V) :- vertex(V), not green(V), not blue(V).
14 green(V) :- vertex(V), not red(V), not blue(V).
15 blue(V) :- vertex(V), not red(V), not green(V).
16
17 % Filter invalid guesses
18 :- vertex(V1), vertex(V2), edge(V1, V2), red(V1), red(V2).
19 :- vertex(V1), vertex(V2), edge(V1, V2), green(V1), green(V2).
20 :- vertex(V1), vertex(V2), edge(V1, V2), blue(V1), blue(V2).
21
22 col(V, red) :- red(V).
23 col(V, blue) :- blue(V).
24 col(V, green) :- green(V).
```

5.3.1 Performance Test Results

The results of running the above described performance tests are summarized in Table **??**.

Table 5.1: All execution times are given in seconds.

Instance	**Classic-3col**	**XML-3col**
K3	0.684	1.202
G8-05-4	0.731	1.520
G13-03-4	0.726	1.816
G13-05-4	0.694	1.871
G21-03-4	0.775	3.359
G34-02-3	0.805	10.318
G55-01-4	0.830	28.830

CHAPTER 6

Discussion

In this chapter, we discuss observations from the expirements conducted in Chapter 5. It is split into two main parts: In the first part, we discuss recurring patterns in code written in Evolog as observed on the model application written in Chapter 5. In the second part of this chapter, we take a look at the performance metrics gained from the testing described in Section 5.3.

6.1 Observations from writing Evolog code

Considering the example application discussed in Section 5.1, this section aims to highlight potential areas for improvement of the language.

6.1.1 Unwrapping list terms

As the examples in Section 5.1 clearly show, there is a certain amount of boilerplate code involved whenever one has to "unpack" a list term obtained as an output term of a module literal. Since in most cases where list terms come into play, one actually wants to work on individual values on the list, we end up with many iterations of similar-looking code that is needed to transform one atom with a single list term into a set of "element atoms", each referring to one element of the list. An example of this is the subset of the 3-Coloring module (see Example 3.2.1) where the module input (an atom referencing a list of vertices and a list of edges) is deconstructed in order to represent a graph using instances of predicates `vertex/1` and `edge/2`. The relevant part of the program is shown in Listing 6.2.

Listing 6.1: Unwrapping list terms representing a graph

```
1 vertex_element(V, TAIL) :- graph(lst(V, TAIL), _).
2 vertex_element(V, TAIL) :- vertex_element(_, lst(V,
     TAIL)).
```

```
3         vertex(V) :- vertex_element(V, _).
4         edge_element(E, TAIL) :- graph(_, lst(E, TAIL)).
5         edge_element(E, TAIL) :- edge_element(_, lst(E, TAIL)).
6         edge(V1, V2) :- edge_element(edge(V1, V2), _).
```

The 6 lines of code shown in Listing 6.2 deal with deconstructing two list terms, a vertex list and and edge list, respectively, into individual atoms such that each atom corresponds to one element of the list. The transformation always follows the same pattern:

- One rule derives an instance of an "element atom" from the head of the list, which is obtained by deconstructing the list term into the form `lst(V, TAIL)`, where `TAIL` is the rest of the list without the head value represented as variable `V`.
- A second rule recursively derives further list elements from element atoms by further deconstructing the `TAIL` terms of the individual element atoms. Repeated evaluation of this rule eventually leads to one element atom for every element of the original list term.
- Finally, a third rule derives the actual domain atoms we're interested in by projecting away the partial lists from the individual element atoms.

Example 6.1.1 demonstrates this method of unwrapping list terms based on a list representation of the complete graph with 3 vertices K_3.

Example 6.1.1 (List Unwrapping)**.** Listing **??** shows the "graph unwrapping" program from Listing 6.2 including a fact representing the complete graph K_3 encoded as two list terms.

Listing 6.2: Unwrapping the list representation of K_3.

```
1         graph(
2                 lst(a, lst(b, lst(c, lst_empty))),
3                 lst(edge(a, b), lst(edge(b, c), lst(edge(c,
                     a), lst_empty)))).
4
5         vertex_element(V, TAIL) :- graph(lst(V, TAIL), _).
6         vertex_element(V, TAIL) :- vertex_element(_, lst(V,
             TAIL)).
7         vertex(V) :- vertex_element(V, _).
8         edge_element(E, TAIL) :- graph(_, lst(E, TAIL)).
9         edge_element(E, TAIL) :- edge_element(_, lst(E, TAIL)).
10        edge(V1, V2) :- edge_element(edge(V1, V2), _).
```

Listing 6.3 shows the (single) answer set of the program given in Listing 6.2

Listing 6.3: The unwrapped list representation of K_3.

```
Answer set 1:
        { edge(a, b)
        edge(b, c)
        edge(c, a)
        edge_element(edge(a, b), lst(edge(b, c), lst(edge(c,
           a), lst_empty)))
        edge_element(edge(b, c), lst(edge(c, a), lst_empty))
        edge_element(edge(c, a), lst_empty)
        graph(
                lst(a, lst(b, lst(c, lst_empty))),
                lst(edge(a, b), lst(edge(b, c), lst(edge(c,
                   a), lst_empty))))
        vertex(a)
        vertex(b)
        vertex(c)
        vertex_element(a, lst(b, lst(c, lst_empty)))
        vertex_element(b, lst(c, lst_empty))
        vertex_element(c, lst_empty) }
```

Generalizing list-unrwapping As Example 6.1.1 demonstrates, unwrapping a list term always takes 3 rules that only differ in predicate names. Definition 6.1.1 introduces a potential "syntacic sugar" solution we call an *unwrap rule*.

Definition 6.1.1 (List-Unwrapping-Rule)**.** Given an Atom a with terms $t_1, \ldots, t_k, \ldots, t_n$, where t_k is a list term, we define the following rule to be the *unwrap rule for* t_k:

$$e(V) \leftarrow \#unwrap\{V \mid t_k\},\ a(t_1, \ldots, t_k, \ldots, t_n).$$

In the unwrap rule, $e(V)$ is called an *element atom*, where V refers to one value from the list t_k. We define the semantics of the above rule to be equivalent to the following program:

$$\begin{aligned} list\_element(V, TAIL) &\leftarrow a(t_1, \ldots, lst(V, TAIL), \ldots, t_n). \\ list\_element(V, TAIL) &\leftarrow list\_element(\_, lst(V, TAIL)). \\ e(V) &\leftarrow list\_element(V, \_). \end{aligned}$$

Listing 6.4 shows a potential implementation of the 3-Coloring Module from Example 3.2.1 using the shorthand notation for list unwrapping proposed in Definition 6.1.1.

Listing 6.4: 3-Coloring Module with shorthand list unwrapping rules.

```
#module threecol(graph/2 => {col/2})  {
        % Unwrap input
```

```
3           vertex(V) :- #unwrap{V | VLST}, graph(VLST, _).
4           edge(V1, V2) :- #unwrap{edge(V1, V2) | ELST}, graph(_,
               ELST).
5
6           % Make sure edges are symmetric
7           edge(V2, V1) :- edge(V1, V2).
8
9           % Guess colors
10          red(V) :- vertex(V), not green(V), not blue(V).
11          green(V) :- vertex(V), not red(V), not blue(V).
12          blue(V) :- vertex(V), not red(V), not green(V).
13
14          % Filter invalid guesses
15          :- vertex(V1), vertex(V2), edge(V1, V2), red(V1),
               red(V2).
16          :- vertex(V1), vertex(V2), edge(V1, V2), green(V1),
               green(V2).
17          :- vertex(V1), vertex(V2), edge(V1, V2), blue(V1),
               blue(V2).
18
19          col(V, red) :- red(V).
20          col(V, blue) :- blue(V).
21          col(V, green) :- green(V).
22  }
```

6.1.2 Repetitive File-IO Actions

As we can observe in Listings 5.1 and 5.16, respectively, reading the content of a file, as well as writing to a file require a set of rules that looks the same regardless of the specific file or content in question. Listing 6.5 shows the "standard program" to read all lines from a file:

- A file path (i.e. location in storage), represented using the fact `infile(PATH)` is opened.
- Assuming opening the file produced no error, the first line of content is read using the rule on line 11.
- Based on the first line read (line number 0), we recursively continue reading as long as reading the last line was successful and did not yield `eof` (i.e. a pseudo-term denoting the end of content) as a result.
- If `eof` has been read, the file is closed, and lines are represented as atoms of form `line(LINE_NO, LINE)`.

Listing 6.5: Reading all lines from a file.

```
1  infile_open(PATH, HD) : @fileInputStream[PATH] = HD :-
2          infile(PATH).
3
4  % Handle file opening error
5  error(io, MSG) :- infile_open(_, error(MSG)).
6
7  % Read all lines from infile
8  readline_result(PATH, 0, RES) :
9          @streamReadLine[STREAM] = RES :-
10         infile(PATH), infile_open(PATH,
               success(stream(STREAM))).
11 readline_result(PATH, LINE_NO, RES) :
12         @streamReadLine[STREAM] = RES :-
13         infile(PATH),
14         infile_open(PATH, success(stream(STREAM))),
15         readline_result(PATH, PREV_LINE_NO, PREV_LINE_RES),
16         PREV_LINE_RES != success(line(eof)),
17         LINE_NO = PREV_LINE_NO + 1.
18
19 % close stream after getting eof
20 infile_closed(PATH, RES) :
21         @inputStreamClose[STREAM] = RES :-
22         infile(PATH),
23         infile_open(PATH, success(stream(STREAM))),
24         readline_result(PATH, _, ok(eof)).
25
26 % Extract actual lines and numbers
27 line(LINE_NO, LINE) :-
28         readline_result(PATH, LINE_NO, success(line(LINE))),
29         LINE != eof.
```

The code to write a set of lines to a file looks similar to how a file is read. Listing 6.6 demonstrates the typical implementation. The following steps are needed:

- First, the file to which content is to be written is opened.
- Assuming lines to be written are consecutively numbered and available as atoms of form `line(LINE_NO, LINE)`, the first line (assumed to have an index of 0) is written.
- If the first line has been successfully written, all other lines are written using a recursive rule.

- We derive that all lines have been written when the line with maximum index has been written.
- One `all_lines_written` has been derived, the file is closed.
- The file is also closed in case an error occurs during writing.

Listing 6.6: Writing lines to a file.

```
1  % Open the output file
2  open_result(PATH, RES) : @fileOutputStream[PATH] = RES :-
       outfile(PATH).
3
4  % Write lines in order of ascending line number
5  write_result(0, RES) : @streamWrite[STREAM, LINE] = RES :-
       open_result(PATH, success(stream(STREAM))), outfile(PATH),
       line(0,LINE).
6  write_result(LINE_NO, RES) : @streamWrite[STREAM, LINE] = RES
       :-
7            open_result(PATH, success(stream(STREAM))),
8            write_result(LINE_NO - 1, success(ok)),
9            outfile(PATH), line(LINE_NO, LINE).
10 all_lines_written :- write_result(LINE_NO, success(ok)),
       LINE_NO = #max{NUM : line(NUM, _)}.
11
12 % The file should be closed once all lines were successfully
       written
13 should_close(PATH, STREAM) :- all_lines_written,
       open_result(PATH, success(stream(STREAM))), outfile(PATH).
14
15 % The file should also be closed when an IO error occurs
       during writing
16 should_close(PATH, STREAM) :-
17           write_result(_, error(_)),
18           open_result(PATH, success(stream(STREAM))),
                outfile(PATH).
19
20 % Close the file
21 close_result(PATH, RES) : @outputStreamClose[STREAM] = RES :-
22           should_close(PATH, STREAM),
23           open_result(PATH, success(stream(STREAM))),
24           outfile(PATH).
```

Module-based abstraction of actions As we've demonstrated in Listings 6.5 and 6.6, frequently used file operations like reading all lines from and writing a set of lines to a file, form repetitive code patterns that would greatly benefit from some shortened abstraction. The obvious solutions seems to be to simply permit actions in Evolog Modules. However, this approach presents some challenges with regards to transparent semantics.

Recalling the demand for *action transparency* (see Definition 3.1.8), i.e. that the result of every firing (ground) action rule must be part of an answer set, modules containing actions can not simple be used in module literals as they're described in Section 3.2. Instead, we'd have to apply the definitions of action functions in rule heads to an Evolog Module and have a module function as an action interpretation function. Example 6.1.2 demonstrates this approach.

Example 6.1.2 (Action Modules)**.** In this example, we show a potential implementation for a module that takes a list of numbered strings and a string describing a file location as input, and writes all lines to the referenced file. Listing 6.7 gives an example of how a module definition for a module containing actions might look.

Listing 6.7: Prototypical definition of a module containing Evolog actions.

```
1  #action write_lines(lines/1, path/1 => (success:
       lines_written/1 | error: io_error/1)) {
2          % Unwrap line list
3          line(IDX, LINE) :- #unwrap{line(IDX, LINE) | LST},
               lines(LST).
4          % Unwrap 1-element-list path/1
5          outfile(PATH) :- #unwrap{P | PLST}, path(PLST).
6
7          % Open the output file
8          open_result(PATH, RES) : @fileOutputStream[PATH] = RES
               :- outfile(PATH).
9
10         % Write lines in order of ascending line number
11         write_result(0, RES) : @streamWrite[STREAM, LINE] =
               RES :- open_result(PATH, success(stream(STREAM))),
               outfile(PATH), line(0, LINE).
12         write_result(LINE_NO, RES) : @streamWrite[STREAM,
               LINE] = RES :-
13                 open_result(PATH, success(stream(STREAM))),
14                 write_result(LINE_NO - 1, success(ok)),
15                 outfile(PATH), line(LINE_NO, LINE).
16         all_lines_written :- write_result(LINE_NO,
               success(ok)), LINE_NO = #max{NUM : line(NUM, _)}.
17
18         % The file should be closed once all lines were
               successfully written
```

```
19          should_close(PATH, STREAM) :- all_lines_written,
               open_result(PATH, success(stream(STREAM))),
               outfile(PATH).
20
21          % The file should also be closed when an IO error
               occurs during writing
22          should_close(PATH, STREAM) :-
23                  write_result(_, error(_)),
24                  open_result(PATH, success(stream(STREAM))),
                       outfile(PATH).
25
26          % Close the file
27          close_result(PATH, RES) : @outputStreamClose[STREAM] =
               RES :-
28                  should_close(PATH, STREAM),
29                  open_result(PATH, success(stream(STREAM))),
30                  outfile(PATH).
31
32          io_error("Failed opening file: " + MSG) :-
33                  open_result(_, error(MSG)).
34          io_error("Failed writing line " + IDX + ": " + MSG) :-
35                  write_result(IDX, error(MSG)).
36          io_error("Failed closing file: " + MSG) :-
37                  close_result(_, error(MSG)).
38
39          lines_written(CNT) :- CNT = #count{N, R :
               write_result(N, R)}, not io_error(_).
40  }
```

The action module proposed in Listing 6.7 differs from a "regular" module in the following ways:

- The module output definition `(success: lines_written/1 | error: io_error/1)` distinguishes between a success and an error case.
- Since we assume action result terms to always be unary function terms with symbol either *success* or *error*, the atoms of the given predicates would then be arguments of the respective (success or error) term in their termified form according to Definition 3.2.5.
- The above items imply that an action module result can only be success *or* error, but not both. In an actual implementation of this concept, ways of enforcing this rule would need to be adressed.

Listing ?? shows a prototypical example of how an action module could be used.

```
line(0, "lorem ipsum dolor").
line(1, "sit amet consectetur").
line(2, "adipiscing  elit")

write_result(R) : @write_lines[LINES, "/tmp/out.txt"] = R
        :- LINES = #list{line(I, L) : line(I, L)}.
```

While the approach to writing composite actions using a special form of modules sketched in Example 6.1.2 looks promising at first glance, we note that it also, potentially, projects away information about side-effects: Consider two distinct Frames F_1 and F_2, where in F_1 the `stream_write` action in module `write_lines` gives an error for line 1, and in F_2, the same action yields an error for line 2. In both frames, the result of the action rule in the top-level program is going to be an error, specifically `write_result(io_error("Failed writing line 1:  error"))` for F_1, and `write_result(io_error("Failed writing line 2:  error"))` (note that wrapping list terms mandated by module output translation were ignored since there is only one answer set holding one instance of `io_error` anyway). While in this case the line on which writing failed is made transparent through the error message, in general, it would be possible to project away the information how many lines were written, i.e. two frames might yield different outcomes for the actions inside the module while yielding the exact same result for the overall program. Potential implementations of this concept would have to make allowances for this.

6.2 Potential improvements to solving performance

The performance test results from Section 5.3.1 show a strong increase in runtime of the XML-3-Coloring application with increasing vertex count of the input graph. Intuitively, this seems easily explained - the XML-parsing module described in Section 5.1.3 has a number of predicates whose (ground) instance count in an answer set is $O(n^2)$ with regards to the number of instances of the predicates they depend on. A typical example are rules like in Listing 5.6 where, in order to find pairs of corresponding opening and closing XML tags, we construct a partial ordering of tokens, resulting in $\frac{n^2-n}{2}$ atoms for the relation predicate. Similar constructs occur multiple times in the XML parsing module. While information about the position of tokens in the input text relative to each other is necessary for the program, it would not have to be kept in memory at all times - the program in the XML parsing module is stratified, and the output predicates which are actually reported back to the calling program all reside in the uppermost stratum.

Based on the observations above, we could reasonably adapt Alpha's stratified evaluation algorithm (see Section 4.1.2) to discard predicates from working memory that are no longer needed for evaluation (i.e. all instances of all directly dependent predicates have been caluclated) and are not part of a module's output predicates, thereby reducing

memory footprint and in turn reduce the amount of time spent on (fruitless) garbage collection cycles in the JVM running Alpha.

CHAPTER 7

Conclusion

The idea of this work was to extend traditional answer set programming with a means to apply "side-effects", i.e. influence the outside world, from a program, while still preserving fully declarative semantics.

To that end, we introduced the language extension *Evolog* in Chapter 3, which is a conservative extension to the ASP language under stood by the Alpha ASP solver. Evolog adds a formal representation of actions, which is defined in Section 3.1 to ASP that is inspired by the Monad concept from funtional programming in that it treats the state of the world side-effects are applied on as input and output of an action function. In Evolog, this is captured in the notion of a *Frame*, which describes the interactions of a program with the outside world by means of an interpretation function for action rules. In order to enable a practical implementation of the action semantics that actually enables programs with actions that can be run on a physical computer, we define necessary restrictions on the structure those programs are allowed to have. The most important such restriction is that, whenever an action rule fires, the resulting atom must be part of an answer set, thereby ensuring that for every effect applied to the outside world, there is a witnessing atom in an answer set. In our reference implementation of said action semantics, in order to get to a condition that can actually be statically verified by an interpreter, we narrow this to only allowing actions in stratified programs. While this may seem like a rather large limitation, it neatly lines up with the fact that - even though ASP has multiple-model-semantics - we only run a given program on one computer at a time, so attempting to apply contradicting actions from multpile models would not be practical.

The second part of the Evolog extension is the program modularization concept defined in Section 3.2. Modularization in Evolog treats modules which encapsulate sub-programs as a special kind of external atoms. By defining each answer set of a module program to constitute one ground substitution of the corresponding module atom, Evolog modules

also provide a means to use program parts with multiple models (and guesses through e.g. the guess-and-check pattern) alongside Actions.

In Chapter 4 we describe our implementation of the Evolog extension, which is based on the lazy-grounding ASP solver Alpha, which we then test by implementing a "real world" application, combining user interaction, file in- and output, parsing, as well as solving an NP-complete decision problem, in Chapter 5. We describe a pure-Evolog implementation of an application that parses XML files representing graphs, on which we the search for three-colorings, and write the found colorings to an XML file. Chapter 5 also contains some short performance test results from a variation of the XML-based 3-coloring application.

Last but not least, Chapter 6 attempts to draw lessons for further improvement of the Evolog language extension as well as implementation performance from the code described in Chapter 5 - we identify code constructs that are often repeated and discuss potential optimizations. Furthermore, we also discuss solving performance for Evolog programs, and sketch a possible approach to reducing the memory footprint of an Evolog program.

7.1 Outlook

While the work described here delivers a possible start on action semantics and a compatible modularization concept for (lazy-grounding) ASP solvers, it does not go beyond a first step, and there are a number of topics worth exploring further.

An intuitive first step for furter improving the Evolog extension would be actually implementing the language constructs sketched in Section 6.1, namely syntactic shorthands for working with lists as well as composite actions. Furthermore, the current implementation does not support nested module calls, i.e. a module program cannot use another module. Since there is no semantic reason prohibiting this, it would suggest itself as a major improvement in the area of developer comfort.

Apart from developing language features, implementing additional applications using actions could lead to further insights. The actions provided for the reference implementation shown here are very centered around file input and output. While means to write to and read from file descriptors already enable a vast number of different programs, further actions, for example to interact with network sockets, or even build graphical user interfaces, would greatly increase the usability of Evolog as a potential general purpose programming language.

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, 30, 38, 39

CBP Common Base Program. 21, 22, 45, 46, 48, 107

CDNL Conflict-driven Nogood Learning. 21, 39, 45

DOM Document Object Model. 66, 81

DTD Document Type Definition. 65, 66

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.

[Cona] Alpha Contributors. Aggregate support in alpha. `https://github.com/alpha-asp/Alpha/pull/268`. Accessed: 2024-09-29.

[Conb] Vavr Contributors. Vavr. `https://github.com/vavr-io/vavr`. Accessed: 2024-08-10.

[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.

[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. Evolog reference implementation. `https://github.com/alpha-asp/Alpha/pull/338`. Accessed: 2024-08-01.

[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.

[LT94] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *ICLP*, volume 94, pages 23–37, 1994.

[LW17] Lorenz Leutgeb and Antonius Weinzierl. Techniques for efficient lazy-grounding asp solving. In *International Workshop on Functional and Constraint Logic Programming*, pages 132–148. Springer, 2017.

[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.

[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.