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

Intro here

CHAPTER 2

Preliminaries

2.1 Answer Set Programming

When speaking of *answer set programming*, 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.

abbreviations!

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* $t_1 \dots t_2$ is a term.
- For function symbol $f \in ID$ and argument terms $t_1, \dots, t_n$, the *functional expression* $f(t_1, \dots, t_n)$ is a term.

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

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

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

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

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

is called an *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 (Literal)**.** A literal in ASP is an atom a or ("default"-)negated atom *not* a.

Definition 2.1.8 (Rule, Program)**.** A *rule* is an expression of form

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

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

Definition 2.1.9 (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 snytactic sugar for

$$q \leftarrow b_1, \ldots, b_n, \mathit{not}\ q.$$

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

2.1.2 Semantics

Definition 2.1.10 (Herbrand Universe)**.** The Herbrand Universe HU_P of a Program P is the set of all valid terms that can be constructed with respect to Definitions 2.1.1, 2.1.2 and 2.1.4. Note that most papers use stricter definition 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 2.1.11 (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 2.1.12 (Herbrand Interpretation)**.** A Herbrand Interpretation is a special form of first order interpretation where the domain of the interpretation is a Herbrand Universe and terms are the interpretation of a term is the term itself, i.e. the corresponding element of HU_P. Intuitively, Herbrand Interpretations constitute listings of atoms that are true in a given program.

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 2.1.13 (Substitution, adapted from [Wei17])**.** A substitution $\sigma : \mathit{VAR} \mapsto (\mathit{ID} \cup \mathit{INT})$ 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 2.1.14 (Grounding)**.** Given a rule r, the *grounding* of r, $\mathit{grnd}(r)$, 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.

CHAPTER 3

The Evolog Language

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

3.1 Actions in Evolog

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

cite something here!

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

3.1.1 Syntax

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

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

where

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

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

3.1.2 Semantics

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

Action Rule Expansion

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

define (classic ASP) grounding and substitutions in preliminaries

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

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

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

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

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

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

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

Grounding of Action Rules

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

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

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

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

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

The expansion of R is:

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

according to which semantics? reference LFP here

Furthermore, consider following frame F:

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

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

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

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

The sole model of P with respect to frame F is

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

List of Figures

List of Tables

List of Algorithms

Bibliography

[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.

[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.

[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.