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

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Preliminaries **3**

3 The Evolog Language **5**

3.1 Actions in Evolog 5

3.2 Syntax 7

3.3 Semantics 8

List of Figures **11**

List of Tables **13**

List of Algorithms **15**

Bibliography **17**

CHAPTER 1

Introduction

Intro here

CHAPTER 2

Preliminaries

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.

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 $fn(a_H), fn(l_1), \ldots, fn(l_m)$, with the expression $fn(l)$ for an atom or 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 $f_{act}(t_1, \dots, t_n)$, the corresponding input term is $input(t_1, \dots, t_n)$.
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*.

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 the (action-)rule R which writes a message into a file if $dom(X)$ holds:

$$write\_result(F, R) : @fileWrite(F, S) = R \leftarrow file\_handle(F), message(S), dom(X).$$

The expansion of R is

$$\begin{aligned} a_{res}(fileWrite, s(F, S, X), i(F, S), fileWrite(s(F, S, X), i(F, S))) \leftarrow& \\ file\_handle(F), message(S), dom(X).& \\ write\_result(R) \leftarrow a_{res}(fileWrite, s(F, S, X), i(F, S), R).& \end{aligned}$$

3.2 Syntax

Every valid ASP-Core2 program is a valid Evolog program. In addition, Evolog programs may contain *action rules* and *module literals*.

Definition 3.2.1 (Action Rule)**.** Action Rules are ASP rules that have a body as defined by the ASP-Core2 standard [CFG$^+$20] and an *action head*, where an action head is of the following form:

$$h : @a[i_1, \dots, i_n] = v_r$$

where

- the head atom h is an ASP atom of form $p(t_1, \dots, t_n)$ with p and $t_1 \dots, t_n$ being a predicate symbol and a list of terms, respectively.
- the function symbol a is the name of an action function, i.e. an identifier starting with a lower-case letter

- action input terms t_1 through t_n are a list of terms
- result variable r_v is a variable.

Action result variables must not occur in the rule body.

Definition 3.2.2 (Module Literals)**.** TBD

3.3 Semantics

3.3.1 Action Rules

Desiderata

For every Evolog Program P and answer set A, the following must be clearly defined:

- $D1$: Which actions were executed by the program?
- $D2$: For every individual action *act*, what led to the action being executed, i.e. of which rule body is *act* a consequence?

Combining $D1$ and $D2$ it follows that

- $D3$: for actions that depend on other actions, it is clearly visible in which sequence they were executed, i.e. the respective execution sequence can be unambiguously reconstructed using the answer set and program('s dependency graph).

Furthermore,

- $D4$: all state changes effected on the outside world by execution of P are reflected in each answer set (as results of actions).

This is an example, make into a proper definition

Definition 3.3.1 (Expansion of action rules)**.** Semantically, every action rule is equivalent to its *expansion*:

$$file1\_open(OP\_RES) : @fileInputStream[PATH] = OP\_RES : -file1(PATH).$$

The expansion of $r1$ is:

$$action\_result(r1, fileInputStream, PATH, fileInputStream(PATH)) : -file1(PATH).$$

$$file1\_open(OP\_RES) : -action\_result(r1, fileInputStream, PATH, OP\_RES).$$

Consequently, it is ensured that for each ground instance of an action rule R_a that fires, there is exactly one *action_result* instance in every answer set. We call this atom a *witness of action act*. Requirement $D1$ is fulfilled through the existence of action witnesses. Furthermore, inspection of a program (or its dependency graph) and all action witnesses in an answer set yields the information demanded in $D2$.

Definition 3.3.2 (Applicability of action rules)**.** In order to guarantee $D1$ and $D4$, for every (ground) action rule R_a that fires, it must hold that the corresponding *witness atom* is part of *every answer set*. Implementations may further restrict this in order to ensure static verifiability of the condition (e.g. by restricting action rules to the stratified part, i.e. common base program of a program).

Definition 3.3.3 (Rule Identifier)**.** Given a non-ground Evolog rule R, $id(R)$ denotes a (program-wide) unique identifier of R.

Definition 3.3.4 (Action function)**.** An action function f_{act} maps a rule id r, a tuple S, and a list of input terms $t_1, \ldots, t_n$ to a result term t_{res}.

- Identifier r references the rule (within a program) that is the *action source* (i.e. that fires in order to trigger the action)
- State S is a ground susbtitution for all body variables of the action source rule, i.e. it encodes the state of the world on which the action operates.

In accordance with Definition 3.3.4, an action witness $action\_result(r_1, fileInputStream, PATH, OP\_RES)$ then reads as "Function $fileInputStream$, with action source r_1, applied to input $PATH$, given world state $(PATH)$, gives result OP_RES".

Definition 3.3.5 (Frame)**.** A *Frame* F is an interpretation function

Definition 3.3.6 (Interpretations of Evolog programs)**.** An Evolog interpretation I of a program P is a tuple (F, H) consisting of a *Frame* F and a herbrand interpretation H. The frame F defines the action functions associated with rules in P.

Definition 3.3.7 (Evolog Model)**.** An evolog interpretation $I = (F, H)$ is a *model* of evolog program P iff H is a *stable model* of P, and all action witness atoms in H are consistent with the action function definitions in F.

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.