Logic Programming
Introducing GOLOG and FLUX

Michael Ruster

University Koblenz-Landau

1 Introduction

An important part of artificial intelligence research deals with autonomous agents acting in environments which are not or only partially known to the agents in the beginning. Allowing agents to reason about their knowledge and interacting with the environment is tackled by different programming methods. This technical report introduces the two logic programming languages GOLOG and FLUX. The basic structure of each language will be shown together with examples to ease understanding. In the first section the situation calculus will be presented. It is a logic formalism GOLOG builds upon. The two subsequent sections 3 and 4 cover GOLOG and FLUX respectively. This report closes with a conclusion drawn from the features of GOLOG and FLUX regarding a multi-agent scenario with incomplete knowledge.

2 Situation Calculus

The situation calculus was introduced by McCarthy and Hayes [3]. It is mainly a fist-order logic but also encodes a dynamic world through second order logic [2]. The situation calculus consists of the three first-order terms *fluents*, *actions* and *situations*. Fluents model properties of the world. Actions may change fluents and hence may modify the world. Every action is "logged" in the situation. Therefore, a situation is a history of actions up to a certain point in time starting from the initial situation s_0. Due to the initial situation modelling the situation before any action has been executed, there can only be one initial situation.

Fluents can be evaluated to return a result. As they are situation dependent, the evaluation result may change over time. Fluents are distinguished in *relational fluents* and *functional fluents*. Relational fluents can hold in situations. Their evaluation hence may return either true or false. An example is given in (1) with p being an agent and s a situation.

$$hasCoffee(p, s) \tag{1}$$

Functional fluents return values instead. A fluent $location(p, s)$ may return the coordinates (x, y) as an example.

Actions also depend on situations. The reason for this is that actions might not always be executable. Instead, it is possible that certain actions need specific

fluents to hold which are modified by actions. Describing when an action is executable is done with *action precondition axioms*. This is expressed by the predicate $Poss(a, s)$ with a being an action. As a recurring example, let us think of the ability to pour coffee to an agent p. This must only be possible when p does not already have coffee. Equation (2) illustrates how this can be formalised.

$$Poss(pourCoffee(p), s) \Leftrightarrow \neg hasCoffee(p, s) \tag{2}$$

As mentioned before, the execution of action must always alter the situation: $do(a, s) \rightarrow s'$. Its effects on the world say fluents are described with *action effect axioms*. Equation (3) shows how pouring a coffee to p will result in p having coffee afterwards.

$$Poss(pourCoffee(p), s) \rightarrow hasCoffee\big(p, do(pourCoffee(p), s)\big) \tag{3}$$

In (3), it is unclear whether other fluents stay unaffected. For example, reasoning about $location(p, s')$ would not be possible, with $do(pourCoffee(p, s)) \rightarrow s'$. With this arises the so called called *frame problem*. Defining for every fluent how every action may or may not affect is only a theoretical solution. The reason for that is that the resulting complexity of $\mathcal{O}(A * F)$ would be too high even in most small worlds. A feasible solution to this problem was proposed by Reiter [4]. His approach was to define every effect of allactions only once. Thus Reiter reduced the complexity to $\mathcal{O}(A * E)$. This solution is known as the *successor state axiom* shown in (4).

$$Poss(a, s) \rightarrow \big[F(do(a, s)) \Leftrightarrow \gamma_F^+(a, s) \vee F(s) \wedge \neg\gamma_F^-(a, s)\big] \tag{4}$$

$F(do(a, s))$ means that the fluent F will be true after exectuing the action a. The first part of the disjunction is $\gamma_F^+(a, s)$ and expresses that the action made the fluent true. $F(s) \wedge \neg\gamma_F^-(a, s)$ as the second part expresses that the fluent had been true before and the action had no influence on it. For a reasonable example, there needs to be a second action which does not influence (1). Therefore, the $sing(s)$ action will be introduced which has no effect on any fluents and can be executed anytime as shown in (5).

$$Poss(sing) \Leftrightarrow \top \tag{5}$$

Given (1), (2) and (5) an example can be compiled like in (6):

$$\begin{aligned} \text{Poss}(a, s) \rightarrow \big[&\text{hasCoffee}(p, \text{do}(a, s)) \\ &\Leftrightarrow [a = \text{pourCoffee}(p)] \\ &\vee\ [\text{hasCoffee}(p, s) \wedge a \neq \text{pourCoffee}(p)]\big] \end{aligned} \tag{6}$$

Equation (6) then formalises that an agent p may only have coffee if it was poured coffee or if it already had coffee and the action was not to pour p a coffee.

3 GOLOG

GOLOG is a language for logic programming introduced by Levesque et al. [2]. It builds on the situation calculus. To allow high-level programming, GOLOG adds complex actions like loops, conditions, tests and non-deterministic elements. As an example, a GOLOG program should have a robot pouring other agents coffee until everybody does have coffee. After that, the robot should sing and terminate. Such a program would reuse the fluent in (1), the action precondition axioms in (2), (5), the successor state axiom in (6) and extend them with the two procedures given in (7) and (8):

$$\begin{aligned} &\textbf{proc}\ \texttt{main}\ [\textbf{while}\ (\exists p)\neg \mathit{hasCoffee}(p) \\ &\qquad\qquad \textbf{do}\ \texttt{pourSOCoffee}(p)\ \textbf{endWhile}]; \\ &\qquad \mathit{sing}\ \textbf{endProc}. \end{aligned} \tag{7}$$

$$\begin{aligned} &\textbf{proc}\ \texttt{pourSOCoffee}\ (\boldsymbol{\pi} p)\ [\neg \mathit{hasCoffee}(p)\textbf{?}; \\ &\qquad\qquad\qquad\qquad \mathit{pourCoffee}(p)]\ \textbf{endProc}. \end{aligned} \tag{8}$$

Equation (7) shows the procedure which can be seen as the main method. It loops as long as there exist agents without coffee and tells the robot to pour coffee to some agent lacking coffee. In the end, the robot sings. Equation (8) allows the robot to non-deterministically choose an agent p to pour coffee to by using the π-operator. The ?-operator is similar to the `if`-operator in other programming languages like Java. Due to the non-determinsmic operator, there can be two different resulting situations like shown in (10) with the initial configuration given in (9):

$$\neg \mathit{hasCoffee}(p, s_0) \Leftrightarrow p = \text{Miriam} \vee p = \text{Sergey}. \tag{9}$$

$$\begin{aligned} s &= do\Big(\mathit{sing}, do\big(\mathit{pourCoffee}(\text{Miriam}), do(\mathit{pourCoffee}(\text{Sergey}), s_0)\big)\Big), \\ s &= do\Big(\mathit{sing}, do\big(\mathit{pourCoffee}(\text{Sergey}), do(\mathit{pourCoffee}(\text{Miriam}), s_0)\big)\Big) \end{aligned} \tag{10}$$

Levesque et al. [2] highlight some problems with GOLOG. These make it unsuitable for a multiple agent-based scenario like the Mars-scenario of the MAPC[1] without considerable modifications and extensions. One problem is that complete knowledge is assumed in the initial situation. This is obviously not the case for scenarios with unknown worlds that get explored by agents. The second problem is that GOLOG does neither offer a solution for internal nor external reactions of agents on sensed actions. A third problem is that exogenous actions say actions out of the agent's control cannot be handled. These could e.g. be actions in control of nature like sudden rain, which are assumed not to be caused by an agent. A fourth problem is highlighted by Thielscher [7] and arises from GOLOG being *regression-based*. This means that for deciding whether an action is executable is only possible after looking at all previous actions and how they might have affected the world. As a result, reasoning takes exponentially longer over time and hence GOLOG does not scale.

[1] `https://multiagentcontest.org/`, online – last accessed 01.05.2014, 16:00.

4 FLUX

FLUX was introduced by Thielscher [7] and offers solutions for the problems of GOLOG presented before. This is done by using the *fluent calculus* instead of the situation calculus. Both are similar but the fluent calculus adds *states*. A state z is a set of fluents $f_1, \cdots, f_n$. In FLUX, it is denoted as $z = f_1 \circ \ldots \circ f_n$. In every situation there always exists only one state with which the current properties of the world are described. Yet, the world can be in the same state in multiple situations. For representing agent knowledge which can be incomplete, FLUX uses *knowledge state*. These are denoted through $KState(s, z)$ meaning that an agent knows that z holds in s.

The frame problem is solved through *state update axioms* [6]. They define the effects of an action as the difference between the state before and after the action. This is modelled with ϑ^- for negative and ϑ^+ for positive effects. Both are simply macros for finite states. Due to using states, reasoning is linear in the size of the state representation. This is called being *regression-based* and therefore FLUX can outperform GOLOG [7].

Disjunctive and negative state knowledge is modelled through constraints. FLUX uses a constraint solver To simplify these constraints until they are solvable. This is done by using *constraint handling rules* introduced by Frühwirth [1]. Their general form is shown in (11). It consists of one or multiple heads H_m, zero or more guards G_k and one or multiple bodies B_n.

$$H_1, \ldots, H_m \Leftrightarrow G_1, \ldots, G_k \mid B_1, \ldots, B_n \tag{11}$$

The general mechanism is that if the guard can be derived, parts of the constraint matching the head will be replaced by the body and hence get simplified. This constraint solver builds the kernel say the foundation of FLUX programs. The domain encodings are built on top of this. Included are the initial knowledge state(s), domain constraints, as well as the action precondition and state update axioms. The final part of a FLUX program is the programmer defined intended agent behaviour called strategy. As a trivial example program, the previous example implemented in GOLOG will be transferred to FLUX in Prolog:

```
1 perform(sing, []).
2 poss(sing, Z) :- all_holds(hasCoffee(_), Z).
3 state_update(Z, sing, Z, []).
```

Listing 1.1. Defintion of the `sing`-action.

Listing 1.1 shows the definition of the `sing`-action. Empty arrays could be replaced by sensed information that could then effect the outcome of the methods. As this is a trivial example, no sensed information is assumed. Line 2 is the precondition that singing is only possible in a state where every agent has coffee. As singing should not alter any fluents, the state `Z` in line 3 is not modified and returned as `Z`.

```
4 perform(pourCoffee(P), []).
5 poss(pourCoffee(P), Z) :-
```

```
6         member(P,[miriam,sergey]),
7         not_holds(hasCoffee(P), Z).
8   state_update(Z1, pourCoffee(P), Z2, []) :-
9         update(Z1, [hasCoffee(P)], [], Z2).
```

Listing 1.2. Definition of the `pourCoffee`-action

In Listing 1.2 the `pourCoffee`-action is defined similarly. Line 6 ensures that Prolog will only look for agents that actually exist instead of iterating over memory addresses. The action must modify the state by adding `hasCoffee(P)` to the state as it is done in line 9. The empty array after it corresponds to ϑ^-, which is empty in this case.

```
10  main_loop(Z) :-
11    poss(sing, Z)
12      -> execute(sing, Z, Z);
13    poss(pourCoffee(P), Z)
14      -> execute(pourCoffee(P), Z, Z1),
15         main_loop(Z1);
16    false.
```

Listing 1.3. Main method which either tells the robot to sing or to pour coffee.

Listing 1.3 models the main method and thus is similar to (7). When singing is possible, the robot will do so and terminate. Else, it will pour someone a coffee and call the main loop again. Line 16 ensures that Prolog will return `No` if neither of the both actions get triggered at some point.

```
17  init(Z0) :-
18         not_holds(hasCoffee(miriam), Z0),
19         not_holds(hasCoffee(sergey), Z0).
```

Listing 1.4. Initial configuration.

The initial configuration in listing 1.4 is comparable to (9) but due to Prolog interpreting from top to bottom, the result will only be `Z = [hasCoffee(sergey), hasCoffee(miriam)]`.

5 Conclusion

Judging mainly from the papers by Levesque et al. [2] and Thielscher [7] it can be said that GOLOG does not satisfy the demands that arise from scenarios with multiple agents exploring unknown worlds. FLUX on the other hand is applicable to such scenarios as has been shown by Schiffel and Thielscher [5]. There, the authors succesfully applied FLUX to the gold mining domain.

References

1. Frühwirth, T.: Theory and practice of constraint handling rules. The Journal of Logic Programming 37(1-3), 95–138 (1998), http://www.sciencedirect.com/science/article/pii/S0743106698100055
2. Levesque, H.J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.B.: GOLOG: a logic programming language for dynamic domains. The Journal of Logic Programming 31(1), 59–83 (1997), http://www.sciencedirect.com/science/article/pii/S0743106696001215
3. McCarthy, J., Hayes, P.: Some philosophical problems from the standpoint of artificial intelligence. Stanford University USA (1969), http://works.bepress.com/cgi/viewcontent.cgi?filename=0&article=1002&context=jozsef_toth&type=additional
4. Reiter, R.: The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy 27, 359–380 (1991), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.2995&rep=rep1&type=pdf
5. Schiffel, S., Thielscher, M.: Multi-agent FLUX for the gold mining domain (system description). In: Computational Logic in Multi-Agent Systems, p. 294–303. Springer (2007), http://link.springer.com/chapter/10.1007/978-3-540-69619-3_17
6. Thielscher, M.: From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. Artificial intelligence 111(1), 277–299 (1999), http://www.sciencedirect.com/science/article/pii/S0004370299000338
7. Thielscher, M.: FLUX: a logic programming method for reasoning agents. Theory and Practice of Logic Programming 5(4-5), 533–565 (2005), http://journals.cambridge.org/production/action/cjoGetFulltext?fulltextid=316562