Reinforcement Learning for Golog Programs

Daniel Beck and Gerhard Lakemeyer

Department of Computer Science
RWTH Aachen University
Germany
{dbeck|gerhard}@cs.rwth-aachen.de

Abstract. A special feature of programs in the action language Golog are non-deterministic actions, which require an agent to make choices during program execution. In the presence of stochastic actions and rewards, Finzi and Lukasiewicz have shown how to arrive at optimal choices using reinforcement learning techniques applied to the first-order MDP representations induced by the program. In this paper we extend their ideas in two ways: we adopt a first-order SMDP representation, which allows Q-updates to be limited to the non-deterministic choice points within a program, and we give a completely declarative specification of a learning Golog interpreter.

1 Introduction

In classical reinforcement learning (RL) and Markov Decision Processes (MDPs), we are given a set of states, actions which stochastically take us from a given state into one of a number of states, and a reward function over states. The goal of learning is to find the optimal policy, which tells us for each state which action to select to maximize our expected reward. In principle this is well understood with methods such as Q-learning solving the problem. However, for most practical applications the huge state and action space is a concern, as explicit representations usually are not viable computationally. To address this problem, state abstraction mechanisms have been explored [1], including FOMDPs [2], which employ first-order logic to characterize a possibly infinite state space using a finite set of formulas.

In this paper, we take this idea further by also constraining the action space using programs written in the action language Golog. Roughly, instead of a state and a set of primitive actions to choose from, we are given a formula describing the current state and a program we need to follow. In the extreme case, when the program is completely deterministic, there is nothing to learn, as the program tells us exactly what the next action is. However, in general the program allows for non-deterministic choices, and here we again need to learn what choices are the best ones in terms of maximizing expected rewards. As we will see, the idea of Q-learning can be adapted to this setting.

More precisely, based on earlier work [2, 3], we start by presenting a method to compute, for a given reward function and Golog program, first-order state

formulas describing the possible states before the program is executed. Roughly, these formulas specify sets of states which are equivalent in the sense that the expected rewards are identical when following a policy which is compliant with the program. Moreover, only those properties of the states which are relevant to the expected reward are reflected in those state formulas.

In a way similar to [1], we then construct a joint semi-MDP (SMDP) over a state space which is made up of tuples consisting of a subprogram of the given program which starts off with a non-deterministic choice and a corresponding state formula.

Lastly, we give the semantics for our new Golog dialect QGolog which incorporates reinforcement learning techniques to learn the optimal decisions for the choice points of a program by means of executing it and observing the outcomes. In essence, we integrate a Q-learning algorithm for the SMDP described above. While [3] also considers a form of Q-learning, their approach is different in that they ignore the SMDP-nature of Golog programs. Perhaps more importantly, we give a completely declarative specification of learning, which we feel is more transparent and better lends itself to formal analysis.

We remark that, since the semantics of Golog requires to axiomatize the dynamics of actions, it is not possible to be completely model-free as in standard RL. Thus, we still assume that the effects of deterministic actions (and thus the successor states) are known; also, the possible outcomes of a stochastic action are known. But we do not assume that the probability distribution over these outcomes is known.

The rest of the paper is organized as follows. After giving a very brief introduction to the situation calculus and Golog, Sect. 3 considers the generation of state-partition formulas. In Sect. 4, we discuss the SMDP induced by a Golog program and specify how Q-learning works in this setting. We then present first experimental results, discuss related work and conclude.

2 Foundations

2.1 The Situation Calculus and Golog

The *situation calculus* is a sorted first-order[1] language with equality and sorts of type action and situation. A situation is a history of executed actions; the initial situation is denoted by S_0; the successor situation which results from executing action a in situation s is denoted as $do(a, s)$. Properties of the world that might change from situation to situation are described by means of (relational) *fluents,* which are ordinary predicate symbols which have a situation term as their last argument. Formulas which mention only a single situation term σ and which do not quantify over situations are called *uniform* in σ. We sometimes consider *situation-suppressed* formulas which are obtained by removing all situation arguments from the fluents. If ϕ is a situation-suppressed formula, then $\phi[\sigma]$ denotes the formula which restores the situation σ in all the fluents mentioned.

[1] There are also some second-order features, which do not concern us here.

The preconditions for each action A are given by $Poss(A(\boldsymbol{x}), s) \equiv \Pi_A(\boldsymbol{x}, s)$.[2] According to Reiter's solution of the frame problem [4] the effects of actions are encoded as so-called *successor-state axioms (SSAs)*,one for each fluent:

$$F(\boldsymbol{x}, do(a, s)) \equiv \Phi_F(\boldsymbol{x}, a, s).$$

A *basic action theory (BAT)* $\mathcal{D}$ consists of the foundational axioms Σ, which define the space of situations, the successor state axioms $\mathcal{D}_{ssa}$, the action preconditions D_{ap}, the unique name axioms for actions $\mathcal{D}_{una}$, and a set $\mathcal{D}_{S_0}$ of first-order sentences uniform in S_0 which describe the fluent values in the initial situation.

Example. Consider the blocks world domain. The fluent $on(b_1, b_2, s)$ expresses that block b_1 is on top of block b_2. The action $move(b_1, b_2)$ moves block b_1 on block b_2. It can only be performed iff there is, in the current situation, no other block on b_1 and on b_2 except if b_2 is the table. Furthermore, b_1 and b_2 have to be distinct.

$$\begin{aligned} Poss(move(b_1, b_2), s) \equiv \neg\exists z.on(z, b_1, s) \wedge \\ (b_2 \neq table \supset \neg\exists z.on(z, b_2, s)) \wedge b_1 \neq b_2 \end{aligned}$$

A block b_1 is on top of b_2 iff it has just been moved there or iff it has been there before and wasn't moved away with the last action.

$$on(b_1, b_2, do(a, s)) \equiv a = move(b_1, b_2) \vee on(b_1, b_2, s) \wedge \neg\exists z.a = move(b_1, z)$$

Besides deterministic primitive actions like *move* we also include stochastic actions. The idea is that, when a stochastic action is executed, nature chooses one of a finite number of deterministic actions [5]. Formally, for a stochastic action a_s the possible choices of primitive actions $n_1, \ldots, n_k$ are defined as

$$choice(a_s(\boldsymbol{x}), a) \equiv \bigvee_{i=1}^{k} a = n_i(\boldsymbol{x}).$$

We denote the probability with which $n_i(\boldsymbol{x})$ is chosen as the outcome of action $a_s(\boldsymbol{x})$ in situation s by $prob(n_i(\boldsymbol{x}), a_s(\boldsymbol{x}), s)$. Axioms of the form

$$\sum_{i=1}^{k} prob(n_i(\boldsymbol{x}), a_s(\boldsymbol{x}), s) = 1$$

ensure that we indeed obtain proper probability distributions.

If the probability distribution with which nature chooses is known, this can also be specified. In our setting, the distribution is generally not known. Moreover, the distributions may change from situation to situation, but not arbitrarily. We assume that there are situation suppressed formulas $\theta_1, \ldots, \theta_r$, which

[2] In formulas like these free variables are understood to be implicitly universally quantified.

partition the set of situations (see Definition 1 below for what that means) so that situations which satisfy the same θ_j agree on the distribution over the n_i. Formally, we include axioms of the form

$$\theta_j(s) \land \theta_j(s') \supset prob(n_i(\boldsymbol{x}), a_s(\boldsymbol{x}), s) = prob(n_i(\boldsymbol{x}), a_s(\boldsymbol{x}), s').$$

Note that in the simple case where the distribution does not change at all, there is only one $\theta = true$.

To ensure full observability it has to be possible to determine the actual outcome of a stochastic action. Therefore, sensing conditions $senseCond(n_i) \equiv \varphi_i$ are defined such that by means of the special action $senseEffect(a_s)$ the truth value of φ_i and thus the actual outcome can be determined.

The regression of a formula ϕ through an action a is a formula ϕ'. The idea is that, for a given BAT, ϕ holds after executing a just in case ϕ' held before the execution of a. Suppose that the SSA for fluent F is $F(\boldsymbol{x}, a, s) \equiv \Phi_F(\boldsymbol{x}, a, s)$. Then we inductively define the regression of a formula which is uniform in the situation $do(a, s)$ as:

$$\begin{aligned}
&Regr(F(\boldsymbol{x}, do(a, s)) = \Phi_F(\boldsymbol{x}, a, s)\\
&Regr(\neg\phi) = \neg Regr(\phi)\\
&Regr(\phi_1 \land \phi_2) = Regr(\phi_1) \land Regr(\phi_2)\\
&Regr(\exists x.\phi) = \exists x.Regr(\phi)
\end{aligned}$$

According to the regression theorem (Theorem 4.5.4 in [5]) two formulas $\phi(\boldsymbol{x}, s)$ and $\phi'(\boldsymbol{x}, do(a, s))$ where $\phi(\boldsymbol{x}, s) = Regr(\phi'(\boldsymbol{x}, do(a, s)))$ are logically equivalent wrt a given BAT, that is, $\mathcal{D} \models \phi \equiv \phi'$.

The high-level agent programming language Golog [6] is based on the situation calculus. Roughly, Golog allows us to write programs where the primitive actions are those defined by a basic action theory. Here we consider the following language constructs: primitive actions (a), test actions ($\varphi?$), sequences ($[\delta_1; \delta_2]$), conditionals (**if** φ **then** δ_1 **else** δ_2 **end**), loops (**while** φ **do** δ **end**), non-deterministic choice ($[\delta_1 \,|\, \delta_2]$), non-deterministic choice of arguments (**pick**$(x, \delta(x))$), non-deterministic iteration (δ^*), and procedures (**proc** $P(\boldsymbol{x})$ δ **end**).

The meaning of a Golog program can be defined with the help of two special predicates $Final(\delta, s)$ and $Trans(\delta, s, \delta', s')$, which can be read as "$\delta$ can legally terminate in situation s" and "executing the first action of program δ in situation s leads to situation s' with remaining program δ'." For example, if A is a primitive action, then $Trans([A; \rho], s, \delta', s')$ holds iff $Poss(A, s)$ holds, $s' = do(A, s)$, and $\delta' = \rho$. For lack of space, we will define $Trans$ only for the new constructs introduced in this paper and refer to [7] for the others. To start with, for a stochastic action a_s, $Trans$ is defined as $Trans(a_s, s, \delta, s') \equiv s = s' \land \delta = senseEffect(a_s)$, that is, a_s is simply replaced by the sensing action, which senses the actual outcome of the action. The idea is that, when $senseEffect(a_s)$ is executed by the Golog interpreter, a_s gets executed first before the sensing takes place. See the description of the interpreter (Section 4) for how this is done.

2.2 Markov Decision Processes

A *Markov Decision Process* (MDP) is a tuple $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R} \rangle$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ a set of actions, $\mathcal{T}(s, a, s')$ assigns probabilities to the transitions from state s to state s' with action a, and $\mathcal{R}(s, a, s')$ assigns a reward to getting from s to s' by performing action a. A solution of a MDP is represented by a policy π which maps states to actions; π^* is the optimal policy and achieves the maximal expected reward. A semi-MDP (SMDP) allows the actions to have different durations. Then, $\mathcal{T}$ defines a mapping $\mathcal{S} \times N \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ where N are the natural numbers. It specifies the probability of getting from a state s in n time steps to state s' by performing action a.

3 Generating Partitions

For a set of situation suppressed state formulas $S(\boldsymbol{x}) = \{\phi_1(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x})\}$, $S(\boldsymbol{x})[s]$ denotes the set of state formulas $S' = \{\phi_i(\boldsymbol{x})[s] \mid 1 \leq i \leq n\}$.

In the following we make the assumption that $\mathcal{D}_{S_0}$ is *completely specified*, i.e., for every state formula ϕ either $\mathcal{D}_{S_0} \models \phi[S_0]$ or $\mathcal{D}_{S_0} \models \neg\phi[S_0]$ holds.

Definition 1. *A set $P(\boldsymbol{x}) = \{\phi_1(\boldsymbol{x}), \ldots, \phi_n(\boldsymbol{x})\}$ of state formulas ϕ_i is a partition iff the following conditions hold for all completely specified $\mathcal{D}_{S_0}$ and an arbitrary ground situation term σ:*

1. $\mathcal{D} \models \forall \boldsymbol{x}. \bigvee_{i=1}^{n} \phi_i(\boldsymbol{x})[\sigma]$ *and*
2. $\mathcal{D} \models \forall \boldsymbol{x}. \phi_i(\boldsymbol{x})[\sigma] \supset \neg \left(\bigvee_{j \neq i} \phi_j(\boldsymbol{x})[\sigma] \right)$.

Definition 2. *Let $S_1(\boldsymbol{x}_1)$ and $S_2(\boldsymbol{x}_2)$ be sets of state formulas. Then, $S_1(\boldsymbol{x}_1) \otimes S_2(\boldsymbol{x}_2)$ is defined as:*

$$S_1(\boldsymbol{x}_1) \otimes S_2(\boldsymbol{x}_2) = \{\phi^1(\boldsymbol{x}_1) \wedge \phi^2(\boldsymbol{x}_2) \mid \phi^1 \in S_1, \phi^2 \in S_2\}$$

Lemma 1. *If $P_1(\boldsymbol{x}_1)$ and $P_2(\boldsymbol{x}_2)$ are state partitions then $P_1(\boldsymbol{x}_1) \otimes P_2(\boldsymbol{x}_2)$ is also a partition according to Def. 1. Also, $\{(\{\phi\} \otimes P_1), \neg\phi\}$ and $\{(\{\phi\} \otimes P_1), (\{\neg\phi\} \otimes P_2)\}$ are partitions.*

3.1 The Reward Partition

The reward function $rew(s)$ defines a mapping from the space of situations into the reals. We assume that the reward function can be written in the following form:

$$rew(s) = r \equiv \phi_1^{rew}[s] \wedge r = r_1 \vee \ldots \vee \phi_m^{rew}[s] \wedge r = r_m$$

where the r_i are distinct numeric constants. To ensure that $rew(s)$ is well-defined it is necessary to require that $P^{rew} = \{\phi_1^{rew}, \ldots, \phi_m^{rew}\}$ is a partition.

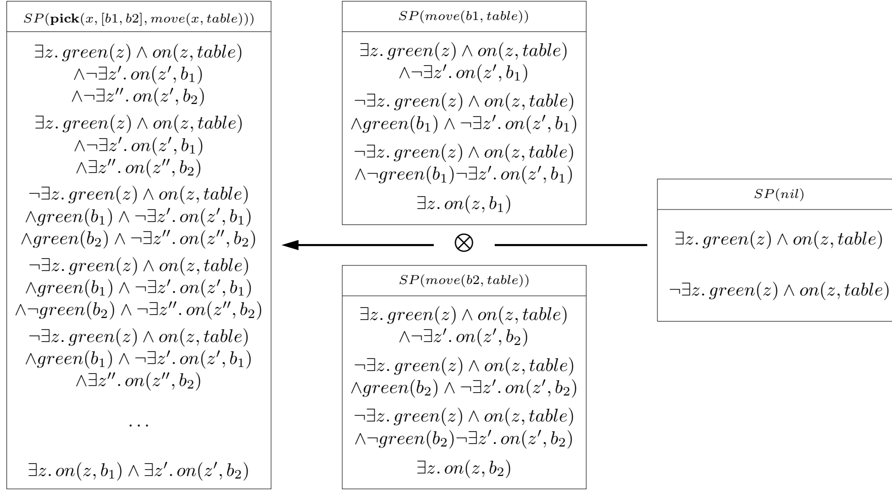

Fig. 1. The partition induced by the program $\mathbf{pick}(x,[b1,b2],move(x,table))$. In case there is a green block on the table a reward of 10 is received otherwise the reward is 0. Consequently, the reward partition is $P^{rew} = \{\exists z.green(z)\wedge on(z,table),\neg\exists z.green(z)\wedge on(z,table)\}$.

3.2 Partitions Induced by Golog Programs

Given a reward function $rew(s)$ a Golog program δ induces a partition which separates those states from each other that (potentially) have different expected rewards when executing program δ. The partition that is induced by the program δ given the reward function $rew(s)$ is denoted by $SP(\delta\,|\,rew(s))$ (often we omit to explicitly mention the reward function when it is clear what reward function is meant and just write $SP(\delta)$).

The partition $SP(\delta)$ is recursively defined over the structure of the remaining program. Consequently, $SP(\delta)$ is not well-defined for programs that have infinite execution traces. To avoid those we preprocess the programs and replace potentially dangerous constructs. In particular, these are:

– [**while** φ **do** δ' **end**; δ] is replaced by a finite number of nested conditionals:

$$[\mathbf{if}\ \varphi\ \mathbf{then}\ [\delta';\mathbf{if}\ \varphi\ \mathbf{then}\ [\delta';\ldots]\ \mathbf{else}\ nil\ \mathbf{end}]\ \mathbf{else}\ nil\ \mathbf{end};\delta]$$

– The star-operator is reduced to a non-deterministic choice between a finite number of repetitions: δ^* is replaced by $\mathbf{nondet}(nil,\delta,\delta^2,\ldots,\delta^n)$, where δ^i stands for the i-fold repetition of δ.

Further, we have to disallow recursive procedure calls in order to guarantee a finite execution trace. Moreover, we assume that the program is nil-terminated, i.e., it has the form $[\gamma;nil]$ where γ is an arbitrary (but finite) Golog program.

The partitions induced by this class of finite Golog programs can then be defined as follows.

The partition induced by the **empty program** is given by the reward partition, i.e., the (empty) program does not affect the expected reward.

$$SP(nil) = P^{rew}.$$

For programs that start with a **sequence** the first element of the sequence determines the induced partition.

$$SP([[\delta_1; \delta_2]; \delta_3]) = SP([\delta_1; [\delta_2; \delta_3]])$$

To determine the partition induced by a program that starts with a **primitive action** a regression is the key. Regression allows to compile state formulas which describe the state before executing a from the state formulas in the partition induced by the remaining program. Further, we make sure that the action's preconditions hold, split up the state formulas according to the reward partition, and complete the partition by adding a state formula for the case where the preconditions are not given.

$$\begin{aligned} SP([a(\boldsymbol{x}); \delta])[s] = \{\{&Regr(\phi_i[do(a(\boldsymbol{x}), s)]) \wedge Poss(a(\boldsymbol{x}), s) \mid \phi_i \in SP(\delta)\} \\ &\otimes P^{rew}[s]\} \cup \{\neg Poss(a(\boldsymbol{x}), s)\} \end{aligned}$$

A leading **test action** introduces a further distinguishing feature to the partition induced by the remaining program: either the test condition holds or it does not.

$$SP([\varphi?; \delta]) = \{\varphi\} \otimes SP(\delta) \cup \{\neg\varphi\}$$

Analogously, the partition induced by a program starting with a **conditional** is defined as:

$$SP([\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ end}; \delta]) = \{\varphi\} \otimes SP([\delta_1; \delta]) \cup \{\neg\varphi\} \otimes SP([\delta_2; \delta])$$

The model of a **stochastic action** is described by a number of possible outcomes and and a situation-dependent probability distribution over those. As already said above the ϑ which define the probability distribution over the outcomes of the stochastic actions a_s form a partition: $P^{pr}_{a_s} = \{\vartheta_1, \ldots, \vartheta_r\}$.

The partition induced by a program starting with a stochastic action can then be defined as:

$$SP([a_s(\boldsymbol{x}); \delta]) = P^{pr}_{a_s} \otimes \left(\bigotimes_{i=1}^{k} SP([n_i(\boldsymbol{x}); \delta]) \right)$$

In case of a **non-deterministic branching** the partition is made up of the combination of the partitions induced by each of the possible branches.

$$SP([[\delta_1 \mid \delta_2]; \delta]) = SP([\delta_1; \delta]) \otimes SP([\delta_2; \delta])$$

Since we restricted the **non-deterministic choice of argument** to a selection from a given set of arguments the partition induced by a program beginning with a non-deterministic choice of argument is the combination of the partitions induced by the remaining program with the different arguments:

$$SP([\mathbf{pick}(x,[v_1,\ldots,v_n],\gamma);\delta]) = \bigotimes_{i=1}^{n} SP([\gamma^x_{v_i};\delta])$$

For a program starting with a procedure call the partition it induces is computed as the partition induced by a program where the name of the procedure is replaced with its body. Assume a procedure P is defined as **proc** $P(\boldsymbol{x})$ δ_P **end**, then

$$SP([P(\boldsymbol{t});\delta]) = SP([\delta_P{}^{\boldsymbol{x}}_{\boldsymbol{t}};\delta]).$$

Theorem 1. *For a Golog program δ and a reward function $rew(s)$, $SP(\delta \mid P^{rew})$ describes a state partition according to Def. 1.*

Proof. by induction on the structure of the program.

1. $SP(nil)$ is a partition by definition.
2. For the proof that $SP([a;\delta])$ is a partition we assume a slightly different definition of $SP([a;\delta])$. The one given below is "finer grained" since it also partitions the cases where $Poss(a,s)$ does not hold.

$$\begin{aligned} SP(a(\boldsymbol{x});\delta)[s] = \{&Regr(\phi[do(a(\boldsymbol{x}),s)]) \mid \phi \in SP(\delta)\} \\ &\otimes \{Poss(a(\boldsymbol{x}),s), \neg Poss(a(\boldsymbol{x}),s)\} \otimes P^{rew}[s] \end{aligned}$$

   According to the regression theorem $\{Regr(\phi[do(a(\boldsymbol{x}),s)]) \mid \phi \in SP(\delta)\}$ is a partition iff $SP(\delta)$ is a partition. Then, according to Lemma 1, $SP([a;\delta])$ as given above is also a partition. This also holds for the original, "coarser" definition of $SP([a;\delta])$ since it combines all cases where the preconditions are not given into a single state formula and does not partition this case further as it is done by the alternative definition above.
3. According to Lemma 1 $SP([\varphi?;\delta])$, $SP([\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{end};\delta])$, $SP([a_s(\boldsymbol{x});\delta])$, and $SP([[\delta_1 \mid \delta_2];\delta])$ are partitions iff $SP(\delta)$ is a partition.

Theorem 2. *The space abstraction defined by the partitioning scheme is* safe*, i.e., for a program δ, ground situations σ_1 and σ_2 it holds that if $\mathcal{D} \models \phi[\sigma_1]$ and $\mathcal{D} \models \phi[\sigma_2]$ for a $\phi \in SP(\delta)$ then the expected reward for executing δ in σ_1 and in σ_2 is the same.*

An example for the incremental generation of the partition induced by a Golog program and a reward function is depicted in Fig. 1.

4 The QGolog Interpreter

In this section we describe how the joint SMDP alluded to in the introduction is defined exactly, how Q-learning works for such an SMDP, and we show how that is embedded in our QGolog interpreter by specifying its formal semantics.

The statespace $\mathcal{S}$ of the SMDP underlying the Golog program consists of tuples $\langle\phi,\delta\rangle$ where δ is a choicepoint in the program, i.e., a subprogram that begins with either a non-determinstic branching or a non-deterministic choice of argument, and $\phi \in SP(\delta)$. For all $\langle\phi,\delta\rangle \in \mathcal{S}$, $\mathcal{A}(\langle\phi,\delta\rangle)$ is the set of possible actions in the state $\langle\phi,\delta\rangle$. If $\delta = [[\delta_1 \,|\, \delta_2]; \delta']$ then $\mathcal{A}(\langle\phi,\delta\rangle) = \{\delta_1, \delta_2\}$; if $\delta = [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta']$ then $\mathcal{A}(\langle\phi,\delta\rangle) = \{\gamma^x_{v_1}, \ldots, \gamma^x_{v_n}\}$. By uniquely identifying the SMDP-actions by Golog programs which correspond to the resepective choices at the choicepoints it is possible to derive a legal execution trace of the program from a given policy for the SMDP. Particularly, the optimal choices at the choicepoints in the program can be derived from the optimal SMDP-policy.

The rewards received for performing a SMDP-action are computed as the sum of the rewards obtained for executing a sequence of primitive actions which leads the program to the next choicepoint in the program and which corresponds to the SMDP-action under consideration. Let k be the number of primitive actions in that sequence then

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{k-1} r_{t+k-1}$$

The update of the Q-value for performing action A_t in state $\langle\phi,\delta\rangle_t$ can then be formulated as

$$Q(\langle\phi,\delta\rangle_t, A_t) \leftarrow Q(\langle\phi,\delta\rangle_t, A_t) + \alpha \cdot \left(R_t + \gamma^k \cdot \max_{A \in \mathcal{A}(\langle\phi,\delta\rangle)} Q(\langle\phi,\delta\rangle_t, A) - Q(\langle\phi,\delta\rangle_t, A_t)\right)$$

where α is the learning rate and k the number of primitive actions executed while performing A_t.

The Q-table storing the Q-values for all state-action pairs is realized as a fluent $q(\phi, \delta, A, v, s)$ that is initialized in D_{S_0} in such a way that all state-action pairs in the SMDP from above are assigned a value. The value for a particular pair can be updated with the action $setQ(\phi, \delta, A, v)$:

$$q(\phi, \delta, p, v, do(a, s)) \equiv a = setQ(\phi, \delta, p, v) \lor q(\phi, \delta, p, v, s) \land \neg\exists v'. a = setQ(\phi, \delta, p, v')$$

Likewise, the fluent $\epsilon(s)$ denotes the probability to deviate from the current policy and to explore another action in situation s. The action $setEpsilon(p)$ changes the fluent's value. Moreover, we assume that the sensing action $senseRnd(r)$ returns a random number $r \in [0, 1]$.

The new construct $\mathbf{learn}(\delta)$ initiates the learning for the program δ. In a program configuration $\langle\mathbf{learn}(\delta), s\rangle$ the program may proceed to a configuration

where the situation is unchanged and the remaining program equals the policy computed by the predicate $QDo(\phi^*, \delta^*, A^*, \delta, s, k, r, \pi)$. The arguments of QDo are the SMDP state $\langle \phi^*, \delta^* \rangle$ corresponding to the last seen choicepoint, the action A^* taken at that state, the current program configuration $\langle \delta, s \rangle$, the number k of primitive actions encountered since the last choicepoint, and the cumulative, discounted reward r obtained since then. Formally:

$$Trans(\mathbf{learn}(\delta), s, \delta', s') \equiv \exists \pi.\, QDo(\phi_{start}, \delta, A_{start}, \delta, s, 0, 0, \pi) \wedge \delta' = \pi \wedge s' = s$$

where $\langle \phi_{start}, \delta \rangle$ is a distinguished start state that is added to $\mathcal{S}$; the only possible action in the state is the start action A_{start}.

The policy computed by QDo is a Golog program that describes a valid continuation of the remaining program. Embedded in the policy are *setQ* actions to update the Q-table, sensing actions to determine the actual outcome of a stochastic action, etc. Also, the policy handles the exploration of the state space, i.e., with a certain probability a non-optimal action (wrt. the current state-action values) is chosen.

The predicate QDo is defined in dependence on the beginning of the remaining program.

The remaining program is the empty program:

$$Do(\phi^*, \delta^*, A^*, nil, s, k, r, \pi) \stackrel{def.}{=} \pi = [setQ(\phi^*, \delta^*, A^*, r); setEpsilon(\eta \cdot \epsilon)]$$

where $\eta \in (0, 1]$.

The remaining program starts with a primitive action:

$$\begin{aligned} &QDo(\phi^*, \delta^*, A^*, [a; \delta], s, k, r, \pi) \stackrel{def.}{=} Poss(a, s) \wedge \exists r_t, \pi'.\, r_t = rew(s) \\ &\quad \wedge QDo(\phi^*, \delta^*, A^*, \delta, do(a, s), k + 1, r + \gamma^k \cdot r_t, \pi') \\ &\quad \wedge \pi = [a; \pi'] \\ &\quad \vee \neg Poss(a, s) \wedge \pi = setQ(\phi^*, \delta^*, A^*, r_{fail}) \end{aligned}$$

r_{fail} is a domain independent negative reward to punish the unsuccessful execution of the program.

The remaining program starts with a test action:

$$\begin{aligned} QDo(\phi^*, \delta^*, A^*, [\varphi?; \delta], s, k, r, \pi) \stackrel{def.}{=} \; &\varphi[s] \wedge QDo(\phi^*, \delta^*, A^*, \delta, s, k, r, \pi) \\ &\vee \neg\varphi[s] \wedge \pi = setQ(\phi^*, \delta^*, A^*, r_{fail}) \end{aligned}$$

The remaining program starts with a stochastic action:

$$\begin{aligned} &QDo(\phi^*, \delta^*, A^*, [a_s; \delta], s, k, r, \pi) \stackrel{def.}{=} \\ &\quad \pi = [a_s; senseEffect(a_s); \\ &\qquad \mathbf{if}\ \varphi_1\ \mathbf{then} \\ &\qquad\quad \mathbf{l}(\phi^*, \delta^*, A^*, \delta, k + 1, r + \gamma^k \cdot rew(do(n_1, s)) \\ &\qquad \mathbf{elsif}\ \varphi_2\ \mathbf{then}\ \ldots \\ &\qquad \mathbf{end}] \end{aligned}$$

where the φ_i are the sensing conditions for the outcomes n_i of the stochastic action a_s. Here, the computation of the policy is interrupted since it is necessary to determine the actual outcome of executing a_s first, before the policy for the remaining program δ is computed. The construct $\mathbf{l}(\ldots)$ mentioned by the policy is comparable to $\mathbf{learn}(\ldots)$ but it allows to memorize the last SMDP state, the action taken there, and the number of primitive actions performed since then and the reward obtained for those.

$$Trans(\mathbf{l}(\phi^*, \delta^*, A^*, \delta, k, r), s, \delta', s') \equiv \exists \pi.\, QDo(\phi^*, \delta^*, A^*, \delta, s, k, r, \pi) \\ \wedge\, \delta' = \pi \wedge s' = s$$

This way the computation of the policy can be continued after executing the stochastic action and observing its outcome.

The remaining program starts with a conditional:

$$QDo(\phi^*, \delta^*, A^*, [\mathbf{if}\ \varphi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{end}; \delta], s, k, r, \pi) \stackrel{def.}{=} \\ \varphi[s] \wedge QDo(\phi^*, \delta^*, A^*, [\delta_1; \delta], s, k, r, \pi) \\ \vee \neg\varphi[s] \wedge QDo(\phi^*, \delta^*, A^*, [\delta_2; \delta], s, k, r, \pi)$$

The remaining program starts with a procedure call $P(\boldsymbol{t})$ and the procedure is defined as $\mathbf{proc}\ P(\boldsymbol{v})\ \delta_P\ \mathbf{end}$:

$$QDo(\phi^*, \delta^*, A^*, [P(\boldsymbol{t}); \delta], s, k, r, \pi) \stackrel{def.}{=} QDo(\phi^*, \delta^*, A^*, [\delta_P{}^{\boldsymbol{v}}_{\boldsymbol{t}}; \delta], s, k, r, \pi)$$

The remaining program starts with a non-deterministic branching:

$$\begin{aligned} & QDo(\phi^*, \delta^*, A^*, [[\delta_1 \mid \delta_2]; \delta], s, k, r, \pi) \stackrel{def.}{=} \\ & \bigvee_{\phi \in SP([[\delta_1 \mid \delta_2]; \delta]} \phi[s] \wedge \exists q_t.\, q(\phi^*, \delta^*, A^*, q_t, s) \\ & \quad \wedge \exists q, A.\, QMax(\phi, [[\delta_1 \mid \delta_2]; \delta], A, q, s) \\ & \quad \wedge \exists q_{t+1}.\, q_{t+1} = q_t + \alpha \cdot \left(r + \gamma^k \cdot q - q_t\right) \\ & \quad \wedge \pi = [setQ(\phi^*, \delta^*, A^*, q_{t+1}); senseRnd(r); \\ & \qquad \mathbf{if}\ r > \epsilon\ \mathbf{then} \\ & \qquad\quad \mathbf{l}(\phi, [[\delta_1 \mid \delta_2]; \delta], A, [A; \delta], 0, 0) \\ & \qquad \mathbf{elsif}\ r \leq \frac{\epsilon}{2}\ \mathbf{then} \\ & \qquad\quad \mathbf{l}(\phi, [[\delta_1 \mid \delta_2]; \delta], \delta_1, [\delta_1; \delta], 0, 0) \\ & \qquad \mathbf{else} \\ & \qquad\quad \mathbf{l}(\phi, [[\delta_1 \mid \delta_2]; \delta], \delta_2, [\delta_2; \delta], 0, 0) \\ & \qquad \mathbf{end}] \end{aligned}$$

Again, the computation of the policy needs to be interrupted here since it needs to be decided randomly whether to explore or to exploit (after updating the Q-value of the last SMDP-state). The auxiliary predicate $QMax(\phi, \delta, A, q_{max}, s)$

determines the action A with the maximal Q-value q_{max} for the state $\langle \phi, \delta \rangle$ in situation s. It is defined as:

$$\begin{aligned}
&QMax(\phi, \delta, A, q_{max}, s) \stackrel{def.}{=} \\
&\bigvee_{A' \in \mathcal{A}(\langle \phi, \delta \rangle)} \exists q_{A'}.q(\phi, \delta, A', q_{A'}, s) \wedge \\
&\bigwedge_{\substack{B \in \mathcal{A}(\langle \phi, \delta \rangle), \\ B \neq A'}} \exists q_B.q(\phi, \delta, B, q_B, s) \wedge q_B \leq q_{A'} \wedge \\
&\qquad A = A' \wedge q_{max} = q_{A'}
\end{aligned}$$

The remaining program starts with a non-deterministic choice of argument:

$$\begin{aligned}
&QDo(\phi^*, \delta^*, A^*, [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta], s, k, r, \pi) \stackrel{def.}{=} \\
&\bigvee_{\phi \in SP([\mathbf{pick}(x,[v_1,\ldots,v_n],\gamma);\delta])} \phi[s] \wedge \exists q_t.\, q(\phi^*, \delta^*, A^*, q_t, s) \\
&\quad \wedge \exists A, q.\, QMax(\phi, [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta], A, q, s) \\
&\quad \wedge \exists q_{t+1}.\, q_{t+q} = q_t + \alpha \cdot \left(r + \gamma^k \cdot q - q_t\right) \\
&\quad \wedge \pi = [setQ(\phi^*, \delta^*, A^*, q_{t+1}); senseRnd(r); \\
&\qquad \mathbf{if}\ r > \epsilon\ \mathbf{then} \\
&\qquad\quad \mathbf{l}(\phi, [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta], A, [A; \delta], 0, 0) \\
&\qquad \mathbf{elsif}\ r \leq \frac{1}{n} \cdot \epsilon\ \mathbf{then} \\
&\qquad\quad \mathbf{l}(\phi, [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta], \gamma^x_{v_1}, [\gamma^x_{v_1}; \delta], 0, 0) \\
&\qquad \mathbf{elsif}\ \ldots \\
&\qquad \mathbf{elsif}\ \frac{n-1}{n} \cdot \epsilon < r \leq \epsilon\ \mathbf{then} \\
&\qquad\quad \mathbf{l}(\phi, [\mathbf{pick}(x, [v_1, \ldots, v_n], \gamma); \delta], \gamma^x_{v_n}, [\gamma^x_{v_n}; \delta], 0, 0) \\
&\qquad \mathbf{end}]
\end{aligned}$$

5 Evaluation

We implemented the QGolog interpreter in Prolog and tested it in the blocks world domain with the following program which either does nothing (the primitive action *noop*) or non-deterministically picks a block, moves that block on the table, and then moves block b_1 on block b_2.

$$[[\mathbf{pick}(x, [b_2, \ldots, b_5], move(x, table)); move(b1, b2)] | noop]$$

The reward function assigns the value ten to situations where block b_1 is on top of block b_2 and zero to all others. If the program reaches a configuration in which it cannot be executed further since a precondition is not fulfilled a

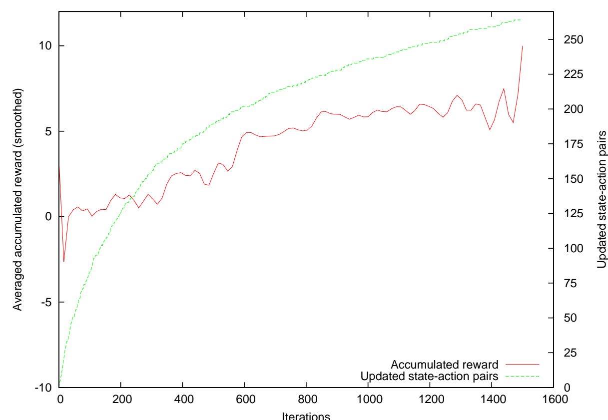

Fig. 2. The graph shows the accumulated reward obtained when executing the program in random, 5-block instances of the blocksworld averaged over 10 runs. For better readability the results are additionally smoothed. Additionally, the number of updated entries in the Q-table is depicted.

negative reward of -10 is given. The values in the Q-table are initialized with -1. Throughout the experiments the learning rate and the exploration probability were kept constant at 0.1.

For each iteration, that is, for each run of the program a new 5-block instance of the blocksworld domain was generated. The results of the experiments are averaged over ten runs and are shown in Fig. 2. For instances in which the program cannot achieve a situation in which block b_1 is on top of block b_2, i.e., the maximally achievable reward is 0 we set the reward to 10 in case an accumulated reward of 0 was achieved to honor the successful execution of the program. Additionally Fig. 2 shows the number of seen state-action combinations during the execution of the program over the course of the experiments.

Although even after the 1500 iterations new state-action combinations are encountered the average accumulated reward crosses the 50% mark after 600 iterations. That is, after 600 iterations the interpreter selects an execution trace that yields an accumulated reward of 5 and above (in average). Admittedly, the results do not look that impressive at first glance. But just reconsider that we have two choicepoints in the program with two and four choices, respectively, and the number of ground configurations of a 5-blocks blocksworld is already in the thousands. That implies that it would take much longer for "flat" Q-learning to explore the state space and find the "good" actions.

Furthermore, the experiments showed that the current way of generating the partition for a **pick** by basically grounding the state formula results in an exponential number of elements in the partitions (wrt the number of elments in the partition induced by the remaining program). Finding a way to characterize the choice of a domain element by a first-order formula would greatly improve the abstraction.

6 Related Work

Restricting the space of policies by means of (partial) programs has been proposed and implemented multiple times. The differences can be mainly found in the expressiveness of the proposed languages in which those partial program are formulated. The HAM-language [8] allows the programmer to define a hierarchical structure of machines whose states can either be action-states which trigger the execution of an action, be a choice state in which the next machine state is selected non-deterministically, or a call state which executes another machine. In [9] the language was extended by parametrization, aborts/interrupts, and memory variables. This raises the expressiveness which allows for more compact programs. These languages were superseded by the language ALisp which extends standard Lisp. In comparison to those languages Golog has a clearly defined semantics which allows to automatically generate abstract state descriptions as it was shown in this paper. State abstraction in ALisp requires the programmer to manually provide abstraction functions [1].

Logic-based representation languages are employed by several approaches for relational reinforcement learning to describe state and action in an abstract fashion. Though, the expressiveness is usually less than the expressiveness of full first-order languages (e.g., quantification is only incorporated implicitly). The approach for symbolic dynamic programming as it was proposed in [2] employs the full expressiveness of a first-order language but at the cost that full theorem proving is required to develop a first-order representation of the value functions. Though, in our approach we make use of the full first-order expressiveness, too, syntactic manipulation of the formulas is sufficient since the structure of the value functions is assumed to be given by the program. This might result in a separation of states which would be joined in the symbolic dynamic programming approach but this is only possible if the complete model is known.

The work which inspired our approach is described in [3]. We refine their approach in several ways. First, we do not employ a horizon in the generation of the partition induced by Golog programs. Only where it is necessary we rewrite the programs to ensures finite execution traces. A consequence thereof is that the horizon is not part of the state description. Secondly, we tightly integrate the reinforcement learning process in the language Golog and do not handle the learning externally. And lastly, we do the Q-update only for the choicepoints and not for every single primitive action. This seems to be reasonable since only at the choicepoints a decsision has to be made—if the program tells the interpreter to execute a primitive action it has no choice. This leads to a faster convergence of the Q-values.

7 Conclusion

In this paper we showed how reinforcment learning can be integrated into the Golog action language framework. We demonstrated how a Golog progam together with its underlying basic action theory gives rise to a first-order SMDP

and we gave a completely declarative specification of a learning Golog interpreter.

In ongoing work we are investigating a number of extensions such as these:

- We want to generalize **pick** to allow for an arbitrary choice of arguments instead of choosing from a finite list of given objects. The advantage would be that state-partition formulas can be generated which are independent of the actual objects in the domain. For example, it would not matter whether the blocks world contained 5 or 500 blocks.
- Perhaps more importantly, we are working on a form of hierarchical reinforcement learning along the lines of [10]. The idea is that procedure calls within programs form a natural hierarchy, and under certain conditions the choices within procedures can be learned independently from those in other procedures. For example, the task of building a tower of a certain color can be divided into collecting blocks of the specified color and then calling a subroutine to build the tower, where learning how to build a tower is completely independent of the color in question.

References

1. Andre, D., Russell, S.J.: State abstraction for programmable reinforcement learning agents. In: AAAI/IAAI. (2002) 119–125
2. Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order MDPs. In: Proceedings of the Seventeenth International. Joint Conference on Artificial Intelligence, IJCAI 2001. (August 2001) 690–700
3. Finzi, A., Lukasiewicz, T.: Adaptive multi-agent programming in GTGolog. In: KI 2006: Advances in Artificial Intelligence, 29th Annual German Conference on AI. Volume 4314., Springer (June 2007) 389–403
4. Reiter, R.: The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression. (1991) 359–380
5. Reiter, R.: Knowledge in Action. MIT Press (2001)
6. Levesque, H., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.: Golog: A logic programming language for dynamic domains. The Journal of Logic Programming **31**(1-3) (1997) 59–83 Reasoning about Action and Change.
7. Giacomo, G.D., Lesperance, Y., Levesque, H.: Congolog, a concurrent programming language based on the situation calculus. Artificial Intelligence **121**(1-2) (2000) 109–169
8. Parr, R., Russell, S.: Reinforcement learning with hierarchies of machines. In: Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10. (1997) 1043–1049
9. Andre, D., Russell, S.: Programmable reinforcement learning agents. In: Proceedings of the 2001 Conference on Advances in Neural Information Processing Systems 13. (2000) 1019–1025
10. Dietterich, T.: The MAXQ method for hierarchical reinforcement learning. In: Proceedings of the Fifteenth International Conference on Machine Learning. (July 1998) 118–126