Symbolic Dynamic Programming

Olga Skvortsova
Department of Computer Science,
Dresden University of Technology,
01062 Dresden (Germany)
e-mail:os447478@inf.tu-dresden.de

Abstract. A symbolic dynamic programming approach for solving first-order Markov decision processes within the situation calculus is presented. As an alternative specification language for dynamic worlds the fluent calculus is chosen and the fluent calculus formalization of the symbolic dynamic programming approach is provided. The major constructs of Markov decision processes such as the optimal value function and the policy are logically represented. The technique produces a set of first-order formulae that minimally partitions the state space. Consequently, the symbolic dynamic programming algorithm presented here does not require neither state nor action space enumeration, thereby solving the drawback of classical dynamic programming methods.

Keywords. Dynamic programming, Markov decision processes, Situation calculus, Fluent calculus

1 Introduction

Planning under uncertainty is a major issue in the study of sequential decision problems and has been addressed by scientists in many different fields, including AI planning, decision analysis, operations research, control theory, and economics [13, 14, 15, 16]. A very wide class of planning problems in these areas can be modeled as Markov decision processes (MDPs) and treated by using algorithms of decision theory.

The term *dynamic programming* (DP) refers to a collection of algorithms that can be used to compute optimal policies given a model of the environment as an MDP. The key idea of the dynamic programming is the use of value functions to organize and structure the search for optimal policies [10].

An advantage of DP methods is their capability to deal with stochastic actions and incomplete world knowledge. However, classical dynamic programming algorithms do not scale up to large state and action spaces. They suffer from what Bellman called "the curse of dimensionality". That is to say, they rely on explicit state enumeration which dramatically limits their use for solving complex tasks with very large state spaces. This implies that their computational requirements grow exponentially with the number of state variables.

In this paper, we address the scalability problem mentioned above by proposing the fluent calculus formalization of the dynamic programming paradigm. We develop a symbolic description of the value iteration algorithm [2] by representing stochastic actions, value functions, and the optimality criterion for choosing an optimal value function within the fluent calculus. Our approach performs a minimal partition of the state space and associates to each obtained partition, or "abstract state", an utility value. Consequently, our symbolic dynamic programming algorithm avoids the explicit enumeration of states and actions. Moreover, by using the logical description for states, we do not need to enumerate domain individuals, i.e., we avoid domain grounding. The latter is very impractical because the number of propositions grows dramatically with the number of domain objects. If, for example, the domain comprises d objects and we have k n_i-ary relations then the number of atoms is $\sum_{i=1}^{k} d^{n_i}$. Taking a very primitive example with 2-ary function f and 3-ary function g and 4 domain objects, we already end up with 80 propositions.

The paper is organized as follows. In Section 2, we describe the classical value iteration algorithm, an instance of a DP method. In Section 3, we present the symbolic dynamic programming approach developed at the University of Toronto. Section 4 is devoted to the fluent calculus formalization of the classical value iteration algorithm. Finally, we draw conclusions and give the directions of future work in Section 5.

2 Classical Dynamic Programming

2.1 Markov Decision Processes

Let us first start with the definition of a Markov Decision Process(MDP) [2].

An MDP is a 4-tuple $(\mathcal{Z}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where

- $\mathcal{Z}$ is a finite set of system states
- $\mathcal{A}$ is a finite set of agent actions
- $\mathcal{P} : \mathcal{Z} \times \mathcal{A} \times \mathcal{Z} \to [0..1]$ is a state transition function. $\mathcal{P}(z, a, z')$ is the probability with which the state z' is reached by performing an action a at a state z.
- $\mathcal{R} : \mathcal{Z} \to \Re$ is a real-valued reward function associating with each state $z \in \mathcal{Z}$ the immediate utility $\mathcal{R}(z)$.

Throughout the paper, we make the following three assumptions. Firstly, only fully-observable MDPs are considered, that is to say, the decision maker (the robot in our case) observes the state it is in. The outcome of each decision is not predictable with certainty but can be observed once it is reached. Secondly, discrete state and action spaces are discussed though general MDPs may work with continuous spaces [3, 4]. Thirdly, rewards are given for every state $z \in \mathcal{Z}$.

Please note that the model is *Markovian* if state transitions and rewards are independent of previous states and actions [2, 3].

2.2 Finding a Policy Given a Model

Next we will explore techniques for determining the optimal policy given a correct model represented as MDP. Because the decision problem faced by an agent is to find an optimal policy (or plan) that maximizes the expected total discounted accumulated reward received over the whole course. We restrict our attention mainly to finding optimal policies for the infinite-horizon discounted model, but one should note that most of these algorithms have analogues for the finite-horizon and average-case models as well [4]. A *stationary policy* π is a mapping $\pi : \mathcal{Z} \to \mathcal{A}$. The value of a policy π, or value function, denoted $V_\pi(z)$, is given by the expected total discounted accumulated reward

$$V_\pi(z) = E\left(\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(z_t)\right)$$

where E is an expectation taken with respect to the probability $\mathcal{P}$[1]; $0 \leq \gamma < 1$ is a discount factor; $\mathcal{R}(z_t)$ is a value of the reward function in a state z_t.

[1]The value you expect to get in a statistical experiment is the mean. If you toss a coin 10 times, you expect to get 5 heads and 5 tails. The mean is often called the "expected value", the "expectation value", or simply the

The discount factor γ can be interpreted in different ways. One may consider it as an interest rate, namely how much is an agent interested to get rewards in the future. Alternatively, γ is regarded as a probability of living another step, or one might prefer to view it as a mathematical means to bound the infinite sum.

$V_\pi(z)$ denotes the value of the state z – it is the expected discounted sum of reward that the agent will get if it starts at state z and then executes the policy π. $V_\pi(z)$ can be defined recursively as follows

$$V_\pi(z) = \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, \pi(z), z') V_\pi(z')$$

In effect, finding an optimal policy is finding the optimal value function which asserts that the value of a state z is the expected instantaneous reward plus the expected discounted value of the next state using the best available action. The optimal value is denoted as $V^*(z)$ and defined as

$$V^*(z) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V^*(z') \right\}$$

for all $z \in \mathcal{Z}$.

2.2.1 Value Iteration Algorithm

A value iteration algorithm [4] can be used to construct optimal value functions for all $z \in \mathcal{Z}$.

The n-stage-to-go Q-function $Q_n(a, z)$ denotes the expected value of performing action a at a state z with n stages to go and acting optimally thereafter.

Please note that the Equation 1 given below will be called in the sequel *the optimality constraint, or the optimality criterion.*

$$V_n(z) := \max_{a \in \mathcal{A}} Q_n(a, z) \tag{1}$$

The crucial result concerning the value iteration algorithm is that the sequence of value functions $V_n(z)$ converges to the optimal value function $V^*(z)$.[2]

"expectation". To formalize the concept a bit more, if for an experiment with the discrete outcomes x_i for which the probability is $P(x_i)$, then the mean denoted as a is given by $a = \sum_i x_i P(x_i)$. For details about expectations and their usage please refer to any textbook on statistics, on estimators in particular.

[2]Since this topic is beyond the scope of our presentation the formal proof will be omitted here. For details see e.g. [3] which serves as an excellent introduction for discrete stochastic dynamic programming. For discussion of stopping criteria please refer to the same book.

$n := 0$.
Specify $\varepsilon > 0$.
Initialise $V_0(z)$ arbitrarily, e.g. $V_0(z) = \mathcal{R}(z)$, for all $z \in \mathcal{Z}$.
`loop`
    $n := n + 1$.
    `loop` for all $z \in \mathcal{Z}$.
        `loop` for all $a \in \mathcal{A}$.
            $Q_n(a, z) := \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V_{n-1}(z')$.
        `end loop`.
        $V_n(z) := \max_{a \in \mathcal{A}} Q_n(a, z)$.
    `end loop`.
`until` $|V_n(z) - V_{n-1}(z)| < \varepsilon$ for all $z \in \mathcal{Z}$.

Figure 1: Finding the optimal value function through value iteration.

Given the optimal value function, the optimal policy $\pi^*(z)$ executed at a state z can be specified as

$$\pi^*(z) = \arg \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z, a, z') V^*(z') \right\}$$

Closing this section, we would like to summarize the results obtained so far. Now we know how to formalize the task as MDP, and then given a formalization to compute the optimal value functions from which one can extract the optimal policy.

3 Symbolic Dynamic Programming within the Situation Calculus

3.1 A Short Introduction to the Situation Calculus

The situation calculus [5] is a first-order language (with some second order features) designed for specifying dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant s_0 is used to denote the initial situation, namely the empty history. Non-empty histories are constructed using a distinguished binary function symbol *do; do(α, S)* denotes the successor situation to *S* resulting from performing the action α. Actions may be parameterized. For example, $put(X, Y)$ might stand for the action of putting

object X on object Y, in which case *do(put(a, b), s)* denotes the situation resulting from placing a on b when the history is S. In the situation calculus, actions are denoted by first order terms, situations (world histories) are also first order terms. For example, *do(putdown(a), do(walk(l), do(pickup(a), s_0)))* is the situation denoting the world history consisting of the sequence of actions *[pickup(a),walk(l),putdown(a)]*. Note that the sequence of actions is obtained from a situation term given above by reading off the actions from right to left.

A particular domain of application will be specified by the union of the following sets of axioms.

▷ Precondition axioms that are actionwise

$$Poss(a(\overline{X}), S) \equiv \Pi_a(\overline{X}, S),$$

where $\Pi_a(\overline{X}, S)$ is a first order formula describing all the conditions under which the action a is possible to execute in the situation S.

▷ Successor state axioms that are fluentwise

$$Poss(a, S) \supset [f(\overline{X}, do(a, S)) \equiv \gamma_f^+(\overline{X}, a, S) \vee (f(\overline{X}, S) \wedge \neg\gamma_f^-(\overline{X}, a, S))],$$

where $\gamma_f^+(\overline{X}, a, S)$ describes all the conditions under which performing action a in S results in f becoming true in the successor situation; $\gamma_f^-(\overline{X}, a, S)$ describes all the conditions under which performing action a in S results in f becoming false in the successor situation.The successor state axioms embody the solution to the frame problem [6].

▷ UNA for the primitive actions which are domain dependent.
For distinct action names a and a',

$$a(X) \neq a'(Y).$$

Identical actions should have identical arguments:

$$a(X_1, \ldots, X_n) = a(Y_1, \ldots, Y_n) \supset X_1 = Y_1 \wedge \ldots \wedge X_n = Y_n$$

▷ Axioms describing s_0

3.2 Regression Operator

Regression operator is defined recursively as follows.

1. If ψ is non-fluent atom then

$$Regr(\psi) = \psi$$

2. If $f(\overline{X}, do(a, S))$ is a fluent whose successor state axiom is

   $Poss(a, S) \supset f(\overline{X}, do(a, S)) \equiv \Phi_f(\overline{X}, a, S)$

   then

   $Regr(f(\overline{T}, do(a', S'))) = \Phi_f(\overline{X}/\overline{T}, a/a', S/S')$

3. Whenever ψ is a formula,

   $Regr(\neg\psi) = \neg Regr(\psi)$
   $Regr((\exists X)\psi) = (\exists X)Regr(\psi)$

4. Whenever ψ_1 and ψ_2 are formulae,

   $Regr(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_2)$,

and likewise for $\forall$, $\supset$, $\equiv$ and $\vee$.

3.3 Stochastic Actions

Now let us enhance the classical version of the situation calculus presented in the previous section by the notion of stochastic actions. We need to do so because stochastic actions have different outcomes that can be described using probabilities. The trick that is used to introduce stochastic action within the situation calculus is to decompose a stochastic action into deterministic primitives under nature's control, referred to as nature's choices. In other words, we give to the nature a prerogative to expand stochastic actions.

Let us consider a logistics example with trucks, blocks, and cities. Trucks are driven between cities, blocks are loaded onto and unloaded from trucks. This scenario can be described using the fluents and actions depicted on the Figure 2.

To illustrate the decomposition of stochastic actions into deterministic primitives consider the action $unload(B, T)$ that is stochastic and corresponds to unloading a block B from truck T.

$$choice(unload(B, T), a) \equiv a = unloadS(B, T) \vee a = unloadF(B, T)$$

where $unloadS(B, T)$ and $unloadF(B, T)$ define two nature's choices of action $unload(B, T)$ and should be thought of successful and poor unloading, respectively.

For each of the nature's choices $n_j(\overline{X})$ associated with action $a(\overline{X})$, we define the probability $prob(n_j(\overline{X}), a(\overline{X}), S)$ with which it is chosen in situation S.

$$prob(unloadS(B, T), unload(B, T), S) = p \equiv rain(S) \wedge p = 0.7 \vee \neg rain(S) \wedge p = 0.9$$
$$prob(unloadF(B, T), unload(B, T), S) = p \equiv p = 1 - prob(unloadS(B, T), unload(B, T), S)$$

Fluents

$bin(B, C, S)$	block B is in the city C in the situation S
$tin(T, C, S)$	truck T is in the city C in situation S
$on(B, T, S)$	block B is on truck T in situation S
$rain(S)$	it is raining in situation S

Actions

$unload(B, T)$	block B is unloaded from truck T
$load(B, T)$	block B is loaded onto truck T
$drive(T, C)$	truck T is driven in city C

Figure 2: Fluents and Actions from Logistics Example.

Precondition Axioms

$Poss(loadS(B, T), S) \equiv (\exists C).bin(B, C, S) \wedge tin(T, C, S)$
$Poss(loadF(B, T), S) \equiv (\exists C).bin(B, C, S) \wedge tin(T, C, S)$
$Poss(unloadS(B, T), S) \equiv on(B, T, S)$
$Poss(unloadF(B, T), S) \equiv on(B, T, S)$
$Poss(driveS(T, C), S) \equiv true$
$Poss(driveF(T, C), S) \equiv true$

Figure 3: Precondition Axioms.

Notice that the unloading is successful with the probability 0.7 if it is raining and 0.9 if it is not raining. Unloading fails with probability of 0.3 if it rains and 0.1, otherwise.

Taking into account decomposition of stochastic actions, precondition and successor state axioms will evolve as shown on the Figure 3 and Figure 4.

3.4 Case Notation

In this section, we introduce an additional notation. This notation is just one convenient way to represent logical formulae, which is introduced to simplify further calculations. A *case statement*

$$t = case[\phi_1, t_1; \ldots ; \phi_n, t_n]$$

Successor State Axioms

$$
\begin{aligned}
bin(B,C,do(a,S)) \equiv\ & (\exists T)[tin\,(T,C,S) \wedge a = unloadS(B,T)] \vee \\
& bin(B,C,S) \wedge \neg(\exists T) a = loadS\,(B,T) \\
tin\,(T,C,do(a,S)) \equiv\ & a = driveS(T,C) \vee \\
& tin\,(T,C,S) \wedge \neg(\exists C') a = driveS(T,C') \\
on\,(B,T,do(a,S)) \equiv\ & a = loadS\,(B,T) \vee \\
& on\,(B,T,S) \wedge a \neq unloadS(B,T) \\
rain\,(do(a,S)) \equiv\ & rain\,(S)
\end{aligned}
$$

Figure 4: Successor State Axioms.

is an abbreviation for the formula

$$\bigvee_{i \leq n} \phi_i \wedge t = t_i,$$

where ϕ_i are logical formulae that partition the state space and t_i are values associated with each partition. Case statements are defined together with operators on them.

$$
\begin{aligned}
& case[\phi_i, t_i : i \leq n] \otimes case[\psi_j, v_j : j \leq m] = \\
& \qquad case[\phi_i \wedge \psi_j, t_i \cdot v_j : i \leq n, j \leq m] \\
& case[\phi_i, t_i : i \leq n] \oplus case[\psi_j, v_j : j \leq m] = \\
& \qquad case[\phi_i \wedge \psi_j, t_i + v_j : i \leq n, j \leq m] \\
& case[\phi_i, t_i : i \leq n] \ominus case[\psi_j, v_j : j \leq m] = \\
& \qquad case[\phi_i \wedge \psi_j, t_i - v_j : i \leq n, j \leq m] \\
& case[\phi_i, t_i : i \leq n] \cup case[\psi_j, v_j : j \leq m] = \\
& \qquad case[\phi_1, t_1; \ldots; \phi_n, t_n; \psi_1, v_1; \ldots; \psi_m, v_m]
\end{aligned}
$$

Now let us reformulate the probabilities within the case notation and address the question whether it is possible to specify the reward and value functions symbolically, namely whether one can give correct case statements for them.

First, let $a(\overline{X})$ be a stochastic action type with possible nature choices $n_1(\overline{X})$,...,$n_k(\overline{X})$. We refer to $a(\overline{X})$ as an action type rather than a specific action because we do not bind it to specific individuals X_1,...,X_n which can be infinitely many. This implies that our solution can be applied for problems with arbitrary domains of individuals. In this way, we perform an abstraction of the action space as well.

The general case statement for probabilities of nature's choices is given by the following equation:

$$prob(n_j(\overline{X}), a(\overline{X}), S) = case[\phi_1^j(\overline{X}, S), p_1^j; \ldots; \phi_n^j(\overline{X}, S), p_n^j]$$

where $\phi_1^j(\overline{X}, S)$,...,$\phi_n^j(\overline{X}, S)$ are logical formulae which represent the conditions satisfied by the corresponding elements of the state space partition and $p_1^j, \ldots, p_n^j$ are the probabilities of nature's choice $n_j(\overline{X})$. The refinement of the probabilities with case notation for action type $unload(B, T)$ from Section 3.3 will have the following form.

$$prob(unloadS(B,T), unload(B,T), S) = p \equiv p = case[rain(S), 0.7; \neg rain(S), 0.9]$$

$$prob(unloadF(B,T), unload(B,T), S) = p \equiv p = case[rain(S), 0.3; \neg rain(S), 0.1]$$

In the following, we assume that only relational fluents determine the reward. The reward function is defined as case statement in the following way:

$$reward(S) = case[\xi_1(S), r_1; \ldots; \xi_m(S), r_m],$$

where parameters $\xi_i(S)$ partition the state space. For example, the reward function

$$reward(S) = case[(\exists B) bin(B, paris, S), 10; \neg(\exists B) bin(B, paris, S), 0]$$

partitions the state space into two parts: one which satisfies the condition that a block B is in $paris$ and another one in which this is not the case. If the first condition is met then the robot is given a reward of 10 points and 0, otherwise.

Value functions can be defined using the case notation in a similar fashion

$$v(S) = case[\beta_1(S), v_1; \ldots; \beta_n(S), v_n],$$

where the states satisfying conditions β_i can be treated as an *abstract* state i. In this way, we specify value functions without explicit state enumeration exploiting the logical flavor of the function. This is the crucial point of the approach presented by C. Boutilier, R. Reiter, and B. Price in [1]. This technique offers the second wind to classical dynamic programming algorithms that require the explicit state formulation hence, loosing efficiency while solving complex tasks in complex environments.

3.5 First-Order Decision-Theoretic Regression

Section 3.4 has shown that main constructs of MDPs, namely probabilities, rewards, and value functions, can be represented in very concise and natural way by using logical descriptions. In this section, we will generalize decision-theoretic regression(DTR) to the first-order case and find logical description for the n-stage-to-go Q-function $Q_n(a, S)$ which is defined here classically as in Section 2.2.1 but with an arbitrary value function v which replaces V_{n-1}. The function $q_v(a, S)$ denotes the value of performing action a in situation S, then acting in a way to obtain value v thereafter.

Let $a(\overline{X})$ be a stochastic action with nature's choices $n_j(\overline{X})$. Then

$$Q_n(a(\overline{X}), S) = \mathcal{R}(S) + \gamma \sum_{S' \in \mathcal{Z}} \mathcal{P}(S, a(\overline{X}), S') \cdot V_{n-1}(S')$$

The situation calculus analogue for $Q_n(a(\overline{X}), S)$ is

$$q_v(a(\overline{X}), S) = reward(S) + \gamma \sum_j prob(n_j(\overline{X}), a(\overline{X}), S) \cdot v(do(n_j(\overline{X}), S))$$

because the only one way when action $a(\overline{X})$ has different outcomes is through different nature's choices. Hence, state S' is a state reached by performing, for instance, nature's choice $n_5(\overline{X})$ at state S.

Denoting functions $reward(S)$, $prob(n_j(\overline{X}), a(\overline{X}), S)$ and $v(do(n_j(\overline{X}), S)$ by $rCase(S)$, $pCase(n_j(\overline{X}), S)$ and $\nu Case(do(n_j(\overline{X}), S))$ as case statements, we obtain

$$q_v(a(\overline{X}), S) = rCase(S) \oplus \gamma \cdot [\oplus_j \{pCase(n_j(\overline{X}), S) \otimes Regr(\nu Case(do(n_j(\overline{X}), S)))\}]$$

where we apply the regression operator to the last component $\nu Case(do(n_j(\overline{X}), S))$ because it refers not to the current situation S, but to the next situation $do(n_j(\overline{X}), S)$. This is the crucial point where regression comes into the play to reflect the "backward nature" of value iteration algorithm.

Since sums and products of case statements are also case statements we obtain a case statement for $q_v(a(\overline{X}), S)$, e.g., $case[\alpha_i(\overline{X}, S), v_i]$, where the parameters $\alpha_i(\overline{X}, S)$ partition the state space into abstract states and v_i are values for each of the abstract states. Handling preconditions, a function $q_v(a(\overline{X}), S)$ becomes a relation $q_v(a(\overline{X}), V, S)$ of the form:

$$q_v(a(\overline{X}), V, S) \equiv \left[\bigvee_i Poss(n_i(\overline{X}), S) \right] \wedge V = case[\alpha_i(\overline{X}, S), v_i]$$

An important outcome of this section is that DTR performs partitioning not only of the state space but of action space as well. Moreover, the presented technique helps to obviate difficulties with explicit state and action enumeration.

3.6 First-Order Value Iteration

Having computed the case (or logical) representation of the n-stage-to-go Q-function $Q_n(a, S)$, let us move to the main issue of the approach in [1], namely to the refinement upon value iteration algorithm by adding symbolic feature.

Coming back to the Section 2.2.1, to classical value iteration algorithm, the only ingredient left to define logically is the condition that value function has maximum value of Q-function $Q_n(a, S)$ over all actions out of $\mathcal{A}$. And the corresponding action is considered as optimal. Now let us move to technical details describing how to express this criterion.

First of all, all values occurring in $q(a(\overline{X}), V, S)$ are assumed to be numerical constants[3].

Suppose we have computed n-stage-to-go relations $q(a(\overline{X}), V, S)$ for each action type $a(\overline{X})$ and let $v(S)$ denote the n-stage-to-go value function. Then the equation for the optimality constraint can be written as

$$v(S) = V \equiv (\exists a).q(a, V, S) \wedge (\forall b).q(b, V', S) \supset V' \leq V \tag{2}$$

Assuming that all actions a are instances of some action type $a(\overline{X})$, we have

$$v(S) = V \equiv [\bigvee_a (\exists \overline{X}).\ q(a(\overline{X}), V, S)] \wedge \bigwedge_b (\forall \overline{Y}, V').\ q(b(\overline{Y}), V', S) \supset V' \leq V \tag{3}$$

We have computed $q(a(\overline{X}), V, S)$ for each action type $a(\overline{X})$ as a case statement:

$$q(a(\overline{X}), V, S) \equiv V = case[\alpha_i^a(\overline{X}, S), v_i^a] \tag{4}$$

Substituting Equation 4 for $q(a(\overline{X}), V, S)$ in Equation 3, we obtain

$$v(S) = V \equiv \left[\bigvee_a (\exists \overline{X}).V = case[\alpha_i^a(\overline{X}, S), v_i^a]\right] \wedge \bigwedge_b (\forall \overline{Y}, V').q(b(\overline{Y}), V', S) \supset V' \leq V \tag{5}$$

Since v_i^a are constants, we can move the existential quantifier into the case statement.

$$v(S) = V \equiv \left[\bigvee_a V = case[(\exists \overline{X}).\alpha_i^a(\overline{X}, S), v_i^a]\right] \wedge \bigwedge_b (\forall \overline{Y}, V').q(b(\overline{Y}), V', S) \supset V' \leq V \tag{6}$$

Defining $(\exists \overline{X}).\alpha_i^a(\overline{X}, S)$ as $\gamma_i^a(S)$, and recalling the definition of the case statement for the $\cup$ operator from Section 3.4, we have

$$v(S) = V \equiv v = \left[\bigcup_a case[\gamma_i^a(S), v_i^a]\right] \wedge \bigwedge_b (\forall \overline{Y}, V').q(b(\overline{Y}), V', S) \supset V' \leq V. \tag{7}$$

Suppose $\bigcup_a case[\gamma_i^a(S), v_i^a]$ has the form $case[\gamma_i(S), v^i : i \leq k]$. Therefore, recalling the definition of the case statement, we have

$$v(S) = V \equiv \left[\bigvee_{i=1}^{k} \gamma_i(S)\ \wedge\ V = v^i\right] \wedge \bigwedge_b (\forall \overline{Y}, V').q(b(\overline{Y}), V', S) \supset V' \leq V. \tag{8}$$

[3]Please note that it does not mean that the approach works only on this assumption. Symbolic descriptions of values can be treated in a same way [1].

Simplifying further, one gets

$$v(S) = V \equiv \bigvee_{i=1}^{k} \left(\gamma_i(S) \ \wedge \ \bigwedge_b (\forall \overline{Y}, V').[q(b(\overline{Y}), V', S) \supset V' \leq V] \right) \wedge V = v^i. \tag{9}$$

Using the case notation, the Equation 9 is transformed into the Equation 10.

$$v(S) = V \equiv V = case[\gamma_i(S) \ \wedge \ \bigwedge_b (\forall \overline{Y}, V').q(b(\overline{Y}), V', S) \supset V' \leq v^i, v^i : i \leq k] \tag{10}$$

Simplifying the expressions $q(b(\overline{Y}), V', S) \supset V' \leq v^i$ in the same way as for the action type a, we get

$$v(S) = V \equiv V = case[\gamma_i(S) \ \wedge \ \bigwedge_j \left[\gamma_j(S) \supset v^j \leq v^i], v^i : i \leq k \right]. \tag{11}$$

Since v^i are numerical constants, they can be compared, and therefore Equation 11 is rewritten in more readable way.

$$v(S) = case \begin{bmatrix} \gamma_1(S) \wedge \bigwedge\limits_{\{i|v^i > v^1\}} \neg\gamma_i(S) & v^1 \\ \vdots & \vdots \\ \gamma_k(S) \wedge \bigwedge\limits_{\{i|v^i > v^k\}} \neg\gamma_i(S) & v^k \end{bmatrix} \tag{12}$$

$$v(S) = case \begin{bmatrix} \gamma_1(S) & v^1 \\ \gamma_2(S) \wedge \neg\gamma_1(S) & v^2 \\ \vdots & \vdots \\ \gamma_k(S) \wedge \neg\gamma_1(S) \wedge \neg\gamma_2(S) \wedge \ldots \wedge \neg\gamma_{k-1}(S) & v^k \end{bmatrix} \tag{13}$$

where all $v^i | i \leq k$ preserve the descending order, namely v^1 is the biggest value and v^k the smallest one. In the first line of the Equation 13 the expression $\bigwedge\limits_{\{i|v^i > v^1\}} \neg\gamma_i(S)$ is absent because the value v^1 is the biggest one, in the second line the expression $\bigwedge\limits_{\{i|v^i > v^2\}} \neg\gamma_i(S)$ has only one conjunct because there exists the only one value v^i, namely v^1, which is bigger than v^2. In other words, the formulae $\gamma_i(S)$ are conditions that partition state space into abstract states.

Equation 13 describes a value function in a purely symbolic manner, obviating the need of the explicit state and action space formulation in the value iteration algorithm.

The termination criterion can also easily be represented in case notation. We just take the current and previous (from the previous iteration) values of value function and subtract previous value from current one, i.e., $C^n \ominus C^{n-1}$, where C^n and C^{n-1} are corresponding case statements[4]. The optimal policy (optimal action for every abstract state) is then extracted from the corresponding optimal value function.

MDPs and dynamic programming algorithms are popular approaches to treat probabilistic uncertainty that more realistic robotics applications involve. However, these algorithms face the difficulty of explicit state enumeration; their complexity is exponential in the number of state variables. Symbolic approaches propose a concise and natural way out from this difficulty. Purely symbolic reasoning about value functions and policies leads to no need in explicit state and action formulation, and moreover, to more effective policy (or plan) search. Another crucial advantage of this technique is the ability to work in domains represented by first-order terms without grounding them, hence, avoiding the explosion of number of propositions even in very simple environments.

Keeping these ideas in mind, we would like to illustrate the advantages of the presented approach by an example in the next section.

3.7 An Example

We will develop a coffee delivery example, where a robot delivers a cup of coffee to persons. The uncertainty is captured by the fact that the person can be absent at her office [8].

Let us describe the world as MDP. First, we introduce the actions and fluents stated on Figure 5 and precondition axioms together with the successor state axioms on Figure 6.

For the sake of slight simplification, we want to have two assumptions. First is that $\mathit{startGo}(L_1, L_2)$ and $\mathit{endGo}(L_1, L_2)$ are "incorporated" into an action $\mathit{giveCoffee}(P)$. Secondly, an attempt to serve a person a cup of coffee is made only once in order to prevent the robot from repeatedly trying to serve a person who is out of her office.

Probabilities of nature's choices of stochastic action types $\mathit{endGo}(L_1, L_2)$ and $\mathit{giveCoffee}(P)$ are given below.

$$\mathit{prob}(\mathit{endGoS}(L_1, L_2), \mathit{endGo}(L_1, L_2), S) = \mathit{case}[\mathit{true}, 0.7]$$
$$\mathit{prob}(\mathit{endGoF}(L_1, L_2), \mathit{endGo}(L_1, L_2), S) = \mathit{case}[\mathit{true}, 0.3]$$
$$\mathit{prob}(\mathit{giveCoffeeS}(P), \mathit{giveCoffee}(P), S) =$$
$$\mathit{case}[\mathit{ispresent}(P, S), 0.95; \neg \mathit{ispresent}(P, S), 0.1]$$
$$\mathit{prob}(\mathit{giveCoffeeF}(P), \mathit{giveCoffee}(P), S) =$$
$$\mathit{case}[\mathit{ispresent}(P, S), 0.05; \neg \mathit{ispresent}(P, S), 0.9]$$

[4]Please note that in general it is necessary to take the modulus of the expression $C^n \ominus C^{n-1}$ and then compare the resulting case statement with ε. If each value is less then ε the algorithm terminates, yielding an optimal value function for every abstract state from the state space.

Actions

$startGo(L_1, L_2)$	robot starts to go from location L_1 to location L_2
$endGo(L_1, L_2)$	robot ends at location L_2 starting from location L_1
$giveCoffee(P)$	robot serves a coffee to a person P

Fluents

$going(L_1, L_2, S)$	robot is going from location L_1 to location L_2 in the situation S
$robotloc(L, S)$	robot is in location L in situation S
$hascoffee(P, S)$	person P has a coffee in situation S
$ispresent(P, S)$	person P is present at his/her office in situation S

Figure 5: Actions and Fluents.

Action $endGo(L_1, L_2)$ is successful with probability 0.7 and fails with 0.3. Please note, that in either case no conditions are met. Action $giveCoffee(P)$ is successful, meaning that a coffee delivery was successful, with probability 0.95 if a person is present and with 0.1, otherwise. Accordingly, $giveCoffee(P)$ is poor with likelihood 0.05 if a person is present and with 0.9, otherwise.

To be ready to perform value iteration we should initialize the reward function first[5]. Then consider reward function $reward(S)$ as

$$reward(S) = case[hascoffee(bob, S), 10; hascoffee(ann, S), 20]$$

which means rewarding of the fact that *bob* has coffee. In this case robot will get the immediate utility of 10 points and 20 points if *ann* was served with a coffee.

For the sake of simplicity, initialize $v^0(S)$ in the same way as reward function. Following the first-order value iteration algorithm in the Sections 3.5, 3.6 we compute for each action type $a(\overline{X})$ the case representation of $Q_1(a(\overline{X}), V, S)$ using $v_0(S)$. We first will show DTR of $v_0(S)$ through action $giveCoffee(bob)$ which results from a simplification in $q_1(giveCoffee(bob), V, S)$ with following elements

$\alpha_1(S) \equiv hascoffee(bob, S) \wedge robotloc(office(bob), S)$
$\alpha_2(S) \equiv \neg ispresent(bob, S) \wedge hascoffee(ann, S) \wedge robotloc(office(ann), S)$
$\alpha_3(S) \equiv ispresent(bob, S) \wedge hascoffee(bob, S) \wedge robotloc(office(bob), S)$
$\alpha_4(S) \equiv hascoffee(ann, S) \wedge robotloc(office(ann), S) \wedge \ hascoffee(bob, S)$

which represent the conditions that partition state space into 4 regions (abstract states). The corresponding values of n stage-to-go q-function are

[5] c.f. first-order value iteration algorithm in section 4.3.

Action Precondition Axioms

$Poss(startGo(L_1, L_2), S) \equiv \neg(\exists L, L'). \; going(L, L', S) \wedge robotloc(L_1, S)$
$Poss(endGoS(L_1, L_2), S) \equiv going(L_1, L_2, S)$
$Poss(endGoF(L_1, L_2), S) \equiv (\exists L'). \; going(L_1, L', S) \wedge L' \neq L_2$
$Poss(giveCoffeeS(P), S) \equiv robotloc(office(P), S)$
$Poss(giveCoffeeF(P), S) \equiv robotloc(office(P), S)$

Successor State Axioms

$going(L, L', do(a, S)) \equiv a = startGo(L, L') \vee \; going(L, L', S) \wedge$
$\quad \neg a = endGoS(L, L') \vee \; going(L, L', S) \wedge \neg(\exists L''). \; a = endGoF(L, L'')$
$robotloc(L, do(a, S)) \equiv (\exists P). \; L = office(P) \vee L = hall \wedge$
$\quad (\exists L_1).(a = endGoS(L_1, L) \vee a = endGoF(L_1, L)) \; \vee robotloc(L, S) \wedge$
$\quad \neg(\exists L_2, L_3).(a = endGoS(L_2, L_3) \wedge a = endGoF(L_2, L_3))$
$hascoffee(P, do(a, S)) \equiv a = giveCoffeeS(P) \; \vee \; hascoffee(P, S)$
$ispresent(P, do(a, S)) \equiv ispresent(P, S)$

Figure 6: Precondition and Successor State Axioms.

$v_1 = 29 \qquad v_2 = 27.1 \qquad v_3 = 19.45 \qquad v_4 = 19$

Second, the regression of $v_0(S)$ through action $giveCoffee(ann)$ results in a case statement for $q_1(giveCoffee(ann), V, S)$ with following elements (after simplification):

$\alpha_1(S) \equiv hascoffee(ann, S) \wedge robotloc(office(ann), S)$
$\alpha_2(S) \equiv \neg ispresent(ann, S) \wedge hascoffee(bob, S) \wedge robotloc(office(bob), S)$
$\alpha_3(S) \equiv ispresent(ann, S) \wedge hascoffee(ann, S) \wedge robotloc(office(ann), S)$
$\alpha_4(S) \equiv hascoffee(ann, S)$

and the values are

$v_1 = 38 \qquad v_2 = 37.1 \qquad v_3 = 29.45 \qquad v_4 = 28$

In the next step of the algorithm, the value function is computed using $q_1(a(\overline{X}), V, S)$. Suppose that one iteration already gave us the optimal value function, namely that the termination criterion was met, hence there is no need to increment n. Therefore, value function $v^1(S)$ can be considered to be optimal and it has the following elements:

$\gamma_1(S) \equiv hascoffee(ann, S) \wedge robotloc(office(ann), S)$

$\gamma_2(S) \equiv \neg ispresent(ann, S) \wedge hascoffee(bob, S) \wedge robotloc(office(bob), S)$
$\gamma_3(S) \equiv ispresent(ann, S) \wedge hascoffee(ann, S) \wedge robotloc(office(ann), S)$
$\gamma_4(S) \equiv hascoffee(ann, S) \wedge \neg(hascoffee(bob, S) \wedge robotloc(office(bob), S))$

and the values

$v_1 = 38 \qquad v_2 = 37.1 \qquad v_3 = 29.45 \qquad v_4 = 28$

Action $giveCoffee(ann)$ turns out to be optimal because the maximum of the value function is obtained when this action is chosen.

This simple example shows how in an application which involves probabilistic uncertainty one can find an optimal value function and extract from it an optimal policy (or plan) by purely symbolic reasoning, avoiding explicit state and action enumeration.

4 The Fluent Calculus View

4.1 Alternatives: Reflections and Suggestions

The fluent calculus, much like the situation calculus, provides a methodology for specifying and reasoning about states, actions and causality. It was originally set up as a first-order logic program with equality using SLDE- or SLDENF-resolution as sole inference rule [11]. In the meantime, the fluent calculus has been revised as a predicate logic specification language using constraint handling rules for reasoning [12]. Whereas the original version allowed for backward as well as forward reasoning, the revised version is a strictly forward reasoning approach.

The crucial difference between the situation calculus and the revised version of the fluent calculus is in search strategy. The situation calculus uses backward reasoning (regression) and the fluent calculus forward reasoning (progression) instead.

In forward search, the initial state is the root of the search tree. Each action applied extends the plan by one step and transfers the system to a new successor state, which is a new leaf node in the tree. If the corresponding state is already in the tree, then this node can be deleted. The search ends when a state is identified as a member of the goal set (in which case a plan can be extracted from the tree), or when all branches have been pruned (in which case no plan exists). Forward search attempts to build a plan progressively, i.e., from the beginning to the end, adding actions to the end of the current sequence of actions. Forward search never considers states that cannot be reached from the initial state.

Backward search can be viewed in the following way. The search starts from the goal state which is taken as the root of a tree. We take an action and consider all predecessor states from which we can reach the current state performing this action. The search terminates when the initial state is added to the tree, and a solution plan can again be extracted from the tree. This search is similar to dynamic programming based algorithms for finding an optimal policy (or plan, or path in the tree).

The important thing to observe about these two strategies is that they restrict their attention to the reachable states. That is to say, in forward search, only those states that can be reached from initial state are considered. This can provide benefit over dynamic programming methods if few states are reachable, since unreachable states cannot play a role in constructing a successful plan. In backward approaches only states on some path to the goal state are considered, and this can have significant advantages over dynamic programming if only a part of the state space is connected to the goal state.

Both methods do not make use of all information available, namely forward search does not exploit knowledge of the goal set, nor does backward search exploit knowledge of the initial state.

One should note that there exists an alternative to search strategies mentioned above, namely bidirectional search which uses backward reasoning as well as forward reasoning. The main idea is to simultaneously search forward from the initial state, and backward from the goal state, until the two search frontiers meet. The path from the initial state is then concatenated with the inverse of the path from the goal state to form the complete solution path [9].

This leads us to the conclusion that both forward and backward search techniques offer benefits to dynamic programming algorithms helping to shrink the state space. Moreover, these techniques should not be seen as mutually exclusive, and therefore, the common opinion that regression suits more effectively for dynamic programming algorithms than progression can be disputed [7].

Thus, in order to combine the fluent calculus and dynamic programming we have three possibilities: (1) use the revised version of the fluent calculus and revise dynamic programming such that it reasons forwardly, (2) use the revised version of the fluent calculus and specify a regression operator in this version, or (3) use the original version of the fluent calculus.

To our knowledge, there is no efficient forward dynamic programming algorithm. There always has to be a back up at some point to do any form of prediction. Checking the revised version of the fluent calculus, it is not immediately obvious how a regression operator can be formalized without a major change in the calculus. We would need to introduce a regression operator which performs regression over fluents rather than over actions. This requires the crucial change in the fluent calculus axiomatization, namely the change of the state update axioms (which are initially *actionwise*) to a $fluentwise$ form. In other words, the fluent calculus becomes the counterpart of the situation calculus.

Because this paper is mainly a case study of whether it is beneficial to combine dynamic programming and reasoning in the fluent calculus, we opt for the last possibility. In this case we do not have to change existing systems but can rather concentrate on finding a fluent calculus formalization of the value iteration algorithm presented in the previous sections.

Before doing so, however, the original version of the fluent calculus is briefly sketched in the next subsection.

4.2 Fluent Calculus Revisited

The fluent calculus is a first-order axiomatization technique for solving main Cognitive Robotics problems [12]. Basic notions are *fluents*, *actions*, and *states*.

Fluents are first-order terms starting with function symbol like $on(a, b)$, i.e., block a is on block b, or just constants like *empty*, i.e., arm of the gripper is empty.

Actions are elementary actions of the robot such as $load(B, T)$ denoting the fact of loading block B onto truck T.

States are characterized by multisets of fluents which may or may not be present in certain states.

Formally, we have an alphabet $\mathcal{A}$ with variables $\mathcal{A}_V$ and function symbols $\mathcal{A}_F$ together with predefined function symbols $\{\circ, 1\}$, where $\circ$ is 2-ary function symbol and 1 is a constant.

A *fluent term*, which clearly represents a state of the system can be defined inductively:

- 1 is a fluent term;
- each fluent is a fluent term;
- if S and T are fluent terms, then $S \circ T$ is a fluent term.

For example, $on(a, b) \circ on(b, c) \circ ontable(c) \circ clear(a)$ defines a state in which the block a is on b, the block b is on c, the block c is on the table, and the block a is clear.

Foundational axioms which are used in the fluent calculus are $AC1$, namely associativity(A), commutativity(C), and unit element(1).

$(\forall X, Y, Z)\ X \circ (Y \circ Z) = (X \circ\ Y) \circ Z$	associativity
$(\forall X, Y)\ X \circ Y = Y \circ X$	commutativity
$(\forall X)\ X \circ 1 = X$	unit element

In the fluent calculus, actions are represented by a predicate *action/3*, where the arguments encode preconditions, name, and effects of an action. For every action, we will have the predicate *action* of the form $action(Pre, A, E)$.

With the help of the predicate *causes*/3, we can express that one state is transformed into another one by applying sequence of actions.

$$
\begin{aligned}
&causes(Z, [\,], Z') \leftarrow Z =_{AC1} Z'.\\
&causes(Z, [A|P], Z') \leftarrow\ action(Pre, A, E) \wedge\\
&\qquad\qquad\qquad\quad Z' =_{AC1} E \circ Z'' \wedge\\
&\qquad\qquad\qquad\quad causes(Z, P, Z'' \circ Pre).
\end{aligned}
$$

where the first clause corresponding to the base case says that if the goal state contains the initial one then there is nothing to do, namely that we have reached the initial situation. The second clause can be read as follows: an execution of the plan $[A|P]$ transforms a state Z into a

$rin\,(R, C)$	racing car R is in city C
$tin\,(T, C)$	truck T is in city C
$on\,(R, T)$	racing car R is on truck T
$rain$	it is raining
$unload\,(R, T)$	racing car R is unloaded from truck T
$load(R, T)$	racing car R is loaded onto truck T
$drive\,(T, C)$	truck T is driven to city C

Figure 7: The fluents and actions of a logistics example.

state Z' if there is an action A with the preconditions Pre and effects E and there is a state Z'' satisfying an equation $Z' =_{AC1} E \circ Z''$ and after performing an action A the plan P transforms Z into $Z'' \circ Pre$.

The $\mathcal{FC}$ theory comprises $AC1$-axioms for the function $\circ/2$ and the definitions of *action* and *causes* predicates.

For more details about the version of fluent calculus presented here see [11]. For the current extensions of the fluent calculus please refer to [12].

4.3 Logistics Example

Let us consider the logistics example with cars, trucks and cities. Cars are delivered from one city to another on trucks. Cars may be loaded on trucks and unloaded from them. The trucks are driven between cities. This scenario can be described using the fluents and actions depicted on the Figure 7.

Please note that all actions are stochastic. To deal with stochastic actions within the fluent calculus we will use the same idea as in the situation calculus. Namely, we will decompose a stochastic action into the deterministic primitives under the nature's control. For instance, the stochastic action $load(B, T)$ will be decomposed in the following way.

$$choice(load(B, T), a) \equiv a = loadS\,(B, T) \vee a = loadF\,(B, T)$$

Similarly, for the actions $unload\,(B, T)$ and $drive\,(T, C)$

$$choice(unload\,(B, T), a) \equiv a = unloadS\,(B, T) \vee a = unloadF\,(B, T)$$

$$choice(drive\,(T, C), a) \equiv a = driveS\,(T, C) \vee a = driveF\,(T, C)$$

$$
\begin{array}{l}
prob\,(unloadS\,(R,T), unload\,(R,T), Z) = .7 \leftrightarrow holds\,(rain\,, Z) \\
prob\,(unloadS\,(R,T), unload\,(R,T), Z) = .9 \leftrightarrow \neg holds\,(rain\,, Z) \\
prob\,(unloadF\,(R,T), unload\,(R,T), Z) = .3 \leftrightarrow holds\,(rain\,, Z) \\
prob\,(unloadF\,(R,T), unload\,(R,T), Z) = .1 \leftrightarrow \neg holds\,(rain\,, Z) \\
prob\,(loadS\,(R,T),,T), Z) = .99 \\
prob\,(loadF\,(R,T),,T), Z) = .01 \\
prob\,(driveS\,(T,C), drive\,(T,C), Z) = .99 \\
prob\,(driveF\,(T,C), drive\,(T,C), Z) = .01
\end{array}
$$

Figure 8: The probabilities of nature's choices in the logistics example.

$$
\begin{array}{l}
action\,(rin\,(R,C) \circ tin\,(T,C),\ \ loadS\,(R,T),\ \ on\,(R,T) \circ tin\,(T,C)) \\
action\,(rin\,(R,C) \circ tin\,(T,C),\ \ loadF\,(R,T),\ \ rin\,(R,C) \circ tin\,(T,C)) \\
action\,(on\,(R,T) \circ tin\,(T,C),\ \ unloadS\,(R,T),\ \ rin\,(R,C) \circ tin\,(T,C)) \\
action\,(on\,(R,T) \circ tin\,(T,C),\ \ unloadF\,(R,T),\ \ on\,(R,T) \circ tin\,(T,C)) \\
action\,(tin\,(T,C'),\ \ driveS\,(T,C),\ \ tin\,(T,C)) \\
action\,(tin\,(T,C'),\ \ driveF\,(T,C),\ \ tin\,(T,C'))
\end{array}
$$

Figure 9: The action specifications of the logistics example.

For each of the nature's choices associated with an action we define the probability with which it is chosen in a particular state. For example,

$$prob\,(unloadS\,(R,T), unload\,(R,T), Z) = .7 \leftrightarrow holds\,(rain\,, Z)$$

states that the probability for the successful execution of an *unload* action in state Z is .7 if it is raining, where *holds* is a macro defined by:

$$holds\,(F, Z) \stackrel{def}{=} (\exists Z').Z = F \circ Z'.$$

Altogether, we specify the probabilities shown on Figure 8.

Recall that in the fluent calculus, actions are represented by a predicate *action/3*, where the arguments encode preconditions, name, and effects of an action. Figure 9 shows the specification.

In the next step, we have to define the value of the reward function for each state. We give

a reward of 10 to all states in which the car Ferrari is in Monte Carlo and 0, otherwise:

$$\begin{aligned} &\mathit{reward}(Z) = 10 \leftrightarrow \mathit{holds}(\mathit{rin}(f,m), Z) \\ &\mathit{reward}(Z) = 0 \leftrightarrow \neg\mathit{holds}(\mathit{rin}(f,m), Z). \end{aligned}$$

One should observe that we have specified the reward function without explicit state enumeration. Rather, the state space is divided into abstract states depending on whether the Ferrari is in Monte Carlo or not. Likewise, the value function can be specified with respect to the abstract states only. This is in contrast to classical DP algorithms like the one shown in Section 2, in which the states are explicitly enumerated.

We proceed by specifying the n-stage-to-go Q-function

$$Q_n(a,z) = \mathcal{R}(z) + \gamma \sum_{z' \in \mathcal{Z}} \mathcal{P}(z,a,z') V_{n-1}(z'). \tag{14}$$

within the fluent calculus. In our approach, the only way the execution of an action may lead to different outcomes is by means of nature's choices. Let $a(\overline{X})$ be an action and $a_j(\overline{X})$, $1 \leq j \leq k$, its nature's choices, where $\overline{X}$ is a sequence of arguments. We may replace (14) by

$$q_n(a(\overline{X}), Z) = \mathit{reward}(Z) + \gamma \sum_j \mathit{prob}(a_j(\overline{X}), a(\overline{X}), Z)\, v_{n-1}(Z'_j) \tag{15}$$

with

$$\mathcal{FC} \models \mathit{causes}(Z, [a_j(\overline{X})], Z'_j) \tag{16}$$

for all $1 \leq j \leq k$. One should observe that in the fluent calculus formalization of the value iteration algorithm we are not going to enumerate all states but rather divide the state space into abstract states. In our running example, these are initially the two abstract states, where the Ferrari is in Monte Carlo or is not. In order to meet condition (16), Z'_j is instantiated with these abstract states whereas the variable Z and the parameters $\overline{X}$ are left uninstantiated. By computing the entailment relation, we effectively regress Z'_j through the corresponding nature's choice $a_j(\overline{X})$ to obtain Z. This reflects the backward nature of the value iteration algorithm.

We can now come back to the running example and illustrate the fluent calculus formalization of the value iteration algorithm. Let the discount factor γ be set to .9 and initialize $v_0(Z) = \mathit{reward}(Z)$ for all abstract states Z. The next step is to compute $q_1(a(\overline{X}), Z)$ with respect to the given actions $a(\overline{X})$ and the abstract states. But before doing so let us introduce Condition 4.3.1 the impact of which will be clear in a sequel.

Condition 4.3.1 *Ground $\mathit{rin}(f,m)$-fluents occur at most once in a state.*

Let us start with performing the regression through the action $\mathit{unload}(R,T)$. In this case, we obtain:

$$\begin{aligned} q_1(\mathit{unload}(R,T), Z) \;=\; & \mathit{reward}(Z) + \\ & \gamma\, \mathit{prob}(\mathit{unloadS}(R,T), \mathit{unload}(R,T), Z)\, v_0(Z'_1) + \\ & \gamma\, \mathit{prob}(\mathit{unloadF}(R,T), \mathit{unload}(R,T), Z)\, v_0(Z'_2) \end{aligned} \tag{17}$$

with

$$\mathcal{FC} \models causes(Z, [unloadS(R,T)], Z_1') \tag{18}$$

and

$$\mathcal{FC} \models causes(Z, [unloadF(R,T)], Z_2'). \tag{19}$$

Starting from the given abstract state space, we compute the possible abstract predecessor states with respect to (18) and (19). Consider first the abstract state where the Ferrari is in Monte Carlo, i.e., $holds(rin(f,m), Z_1')$ is true. In this case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes(Z, [unloadS(R,T)], rin(f,m) \circ Z_1'') \tag{20}$$

and

$$\mathcal{FC} \models causes(Z, [unloadF(R,T)], rin(f,m) \circ Z_2''). \tag{21}$$

To solve (20), we effectively have to find a refutation for:

$$?- causes(Z, [unloadS(R,T)], rin(f,m) \circ Z_1'').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- rin(R,C) \circ tin(T,C) \circ X =_{AC1} rin(f,m) \circ Z_1'', \\ \quad causes(Z, [\,], on(R,T) \circ tin(T,C) \circ X). \end{array} \tag{22}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a set of most general unifiers consisting of

$$\sigma_1 = \{R \mapsto f,\ C \mapsto m,\ Z_1'' \mapsto tin(T,m) \circ X\}$$

and

$$\sigma_2 = \{X \mapsto rin(f,m) \circ Y,\ Z_1'' \mapsto rin(R,C) \circ tin(T,C) \circ Y\}.$$

Thus, we obtain the two goals

$$?- causes(Z, [\,], on(f,T) \circ tin(T,m) \circ X)$$

and

$$?- causes(Z, [\,], on(R,T) \circ tin(T,C) \circ rin(f,m) \circ Y).$$

Both goals can be refuted in one step leading to the computed answer substitutions

$$\theta_1 = \{Z \mapsto on(f,T) \circ tin(T,m) \circ X,\ R \mapsto f,\ Z_1'' \mapsto tin(T,m) \circ X, C \mapsto m\}$$

and

$$\theta_2 = \{Z \mapsto on(R,T) \circ tin(T,C) \circ rin(f,m) \circ Y,\ Z_1'' \mapsto rin(R,C) \circ tin(T,C) \circ Y\},$$

respectively. Furthermore, it can be computed that $reward(Z\theta_1) = 0$ and $reward(Z\theta_2) = 10$.

Thus, we obtain two abstract predecessor states Z specified by

$$A \stackrel{def}{=} holds(on(f,T) \circ tin(T,m), Z)\ \wedge \neg holds(rin(f,m), Z)$$

and

$$B \stackrel{def}{=} holds(on(R,T) \circ tin(T,C) \circ rin(f,m), Z).$$

Now consider the second abstract state where the Ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m), Z_1')$ is true. In this case, we obtain:

$$\mathcal{FC} \models causes(Z, [unloadS(R,T)], Z_1') \tag{23}$$

with extra condition that $\neg holds(rin(f,m), Z_1')$ is true.

To solve (23), we have to find a refutation for:

$$?- causes(Z, [unloadS(R,T)], Z_1').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- rin(R,C) \circ tin(T,C) \circ X =_{AC1} Z_1', \\ \quad causes(Z, [\,], on(R,T) \circ tin(T,C) \circ X). \end{array} \tag{24}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_1' \mapsto rin(R,C) \circ tin(T,C) \circ X\}$$

Then we conclude that $\neg holds(rin(f,m), X)$ is true and $(R \neq f\ \vee\ C \neq m)$. The computed answer substitution is:

$$\theta = \{Z \mapsto on(R,T) \circ tin(T,C) \circ X\}$$

with the condition that $\neg holds(rin(f,m), X)$ is true and $(R \neq f\ \vee\ C \neq m)$.

Since $\neg holds(rin(f,m), X)$ is true $reward(Z\theta) = 0$.

Thus we obtain one abstract predecessor state Z specified as

$$C \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z)\ \wedge \neg holds(rin(f,m), Z)\ \wedge (R \neq f\ \vee\ C \neq m).$$

Likewise, solving (19) we have to consider two cases. First, when the Ferrari is in Monte Carlo, i.e., $holds(rin(f,m), Z_2')$ and second, when the Ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m), Z_2')$.

For the first case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes(Z, [unloadF(R,T)], rin(f,m) \circ Z_2'') \tag{25}$$

To solve (25), we effectively have to find a refutation for:

$$?- causes(Z, [unloadF(R,T)], rin(f,m) \circ Z_2'').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- on(R,T) \circ tin(T,C) \circ X =_{AC1} rin(f,m) \circ Z_2'', \\ \quad causes(Z, [\,], on(R,T) \circ tin(T,C) \circ X). \end{array} \tag{26}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{X \mapsto rin(f,m) \circ Y,\ Z_2'' \mapsto on(R,T) \circ tin(T,C) \circ Y\}.$$

Thus, we obtain the goal

$$?- causes(Z, [\,], on(R,T) \circ tin(T,C) \circ rin(f,m) \circ Y).$$

The goal can be refuted in one step leading to the computed answer substitution

$$\theta = \{Z \mapsto on(R,T) \circ tin(T,C) \circ rin(f,m) \circ Y\}.$$

Since $holds(rin(f,m), Z)$ is true, $reward(Z\theta) = 10$.

Thus, we obtain the abstract predecessor state Z specified by

$$D \stackrel{def}{=} holds(on(R,T) \circ tin(T,C) \circ rin(f,m), Z).$$

Now consider the second abstract state where the Ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m), Z_2')$ is true. In this case, we obtain:

$$\mathcal{FC} \models causes(Z, [unloadF(R,T)], Z_2') \tag{27}$$

with extra condition that $\neg holds(rin(f,m), Z_2')$ is true.

To solve (27), we have to find a refutation for:

$$?- causes(Z, [unloadF(R,T)], Z_2').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?\text{–}\ on(R,T) \circ tin(T,C) \circ X =_{AC1} Z_2', \\ \quad causes(Z, [\,], on(R,T) \circ tin(T,C) \circ X). \end{array} \tag{28}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_2' \mapsto on(R,T) \circ tin(T,C) \circ X\}$$

Then we conclude that $\neg holds(rin(f,m), X)$ is true and obtain the computed answer substitution

$$\theta_1 = \{Z \mapsto on(R,T) \circ tin(T,C) \circ X\}$$

with the condition that $\neg holds(rin(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$. Or,

$$\theta_2 = \{Z \mapsto on(R,T) \circ tin(T,C) \circ X\}$$

with the condition that $\neg holds(rin(f,m), X)$ is true and $(R = f \ \wedge \ C = m)$.

Since $\neg holds(rin(f,m), X)$ is true $reward(Z\theta_1) = 0$ and $reward(Z\theta_2) = 0$.

Thus we obtain two abstract predecessor states Z specified as

$$E \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z) \ \wedge \neg holds(rin(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m)$$

and

$$F \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z) \ \wedge \neg holds(rin(f,m), Z) \ \wedge (R = f \ \wedge \ C = m).$$

Since the states A and F, B and D, E and C are specified in the same way and have the same rewards, we finally end up with 3 abstract predecessor states.

$$A \stackrel{def}{=} holds(on(f,T) \circ tin(T,m), Z) \ \wedge \neg holds(rin(f,m), Z)$$

$$B \stackrel{def}{=} holds(on(R,T) \circ tin(T,C) \circ rin(f,m), Z)$$

and

$$C \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z) \ \wedge \neg holds(rin(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m).$$

For each of the abstract predecessor states, we can now compute its q_1-value. There is a slight complication, though, because the probabilities of nature's choices for the *unload* action depend on whether it is raining. Altogether, we obtain the six cases shown in Table 1. Because (B1) and (B2) lead to the same q_1-value, it does not make a difference whether it is raining.

	abstract pred. state Z	q_1-value
(A1)	$A \wedge holds(rain, Z)$	$0 + .9 \cdot .7 \cdot 10 + .9 \cdot 3 \cdot 0 = 6.3$
(A2)	$A \wedge \neg holds(rain, Z)$	$0 + .9 \cdot .9 \cdot 10 + .9 \cdot .1 \cdot 0 = 8.1$
(B1)	$B \wedge holds(rain, Z)$	$10 + .9 \cdot .7 \cdot 10 + .9 \cdot .3 \cdot 10 = 19$
(B2)	$B \wedge \neg holds(rain, Z)$	$10 + .9 \cdot .9 \cdot 10 + .9 \cdot .1 \cdot 10 = 19$
(C1)	$C \wedge holds(rain, Z)$	$0 + .9 \cdot .7 \cdot 0 + .9 \cdot 3 \cdot 0 = 0$
(C2)	$C \wedge \neg holds(rain, Z)$	$0 + .9 \cdot .9 \cdot 0 + .9 \cdot .1 \cdot 0 = 0$

Table 1: The q_1-values for the *unload* action.

The same holds for (C1) and (C2). Altogether, we obtain an abstract predecessor state space with four abstract states Z_1^u, Z_2^u, Z_3^u, and Z_4^u.

$$Z_1^u \stackrel{def}{=} holds(on(f,T) \circ tin(T,m) \circ rain, Z) \ \wedge \neg holds(rin(f,m), Z)$$

$$Z_2^u \stackrel{def}{=} holds(on(f,T) \circ tin(T,m), Z) \ \wedge \neg holds(rin(f,m) \circ rain, Z)$$

$$Z_3^u \stackrel{def}{=} holds(on(R,T) \circ tin(T,C) \circ rin(f,m), Z)$$

$$Z_4^u \stackrel{def}{=} holds(on(R,T) \circ tin(T,C), Z) \ \wedge \neg holds(rin(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m).$$

One should observe that, for example, in case (C), we were considering the action $unload(R,T)$ without instantiating R and T. Thus, the fluent calculus formalization of the classical dynamic programming abstracts not only from the state but also from the action space.

Let us analyze the obtained predecessor abstract states and the corresponding q_1-values. The state Z_1^u differs from the state Z_2^u only in presence of the fluent $rain$. This fact leads to the different q-values. The state Z_3^u has the maximum value because both actions $unloadS$ and $unloadF$ lead us to the goal state. And the last state Z_4^u is given value of 0 because none of the nature's choices of the action *unload* reach the goal state.

Now the same computations should be performed for other actions, i.e., $load(R,T)$ and $drive(T,C)$.

Performing the decision-theoretic regression through the action $load(R,T)$, we obtain:

$$\begin{aligned} q_1(load(R,T), Z) \quad = \quad & reward(Z) + \\ & \gamma \, prob(loadS(R,T), load(R,T), Z) \, v_0(Z_1') + \\ & \gamma \, prob(loadF(R,T), load(R,T), Z) \, v_0(Z_2') \end{aligned} \tag{29}$$

with

$$\mathcal{FC} \models causes(Z, [loadS(R,T)], Z_1') \tag{30}$$

and

$$\mathcal{FC} \models causes(Z, [loadF(R,T)], Z_2'). \tag{31}$$

Starting from the given abstract state space, we compute the possible abstract predecessor states with respect to (30) and (31). Consider first the abstract state where the Ferrari is in Monte Carlo, i.e., $holds(rin(f,m), Z_1')$ is true. In this case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes(Z, [loadS(R,T)], rin(f,m) \circ Z_1'') \tag{32}$$

and

$$\mathcal{FC} \models causes(Z, [loadF(R,T)], rin(f,m) \circ Z_2''). \tag{33}$$

To solve (32), we effectively have to find a refutation for:

$$?- causes(Z, [loadS(R,T)], rin(f,m) \circ Z_1'').$$

It can be rewritten to:

$$\begin{array}{l} ?- action(Pre, loadS(R,T), E), \\ \quad E \circ X =_{AC1} rin(f,m) \circ Z_1'', \\ \quad causes(Z, [\,], Pre \circ X). \end{array} \tag{34}$$

And Equation (34) results in:

$$\begin{array}{l} ?- on(R,T) \circ tin(T,C) \circ X =_{AC1} rin(f,m) \circ Z_1'', \\ \quad causes(Z, [\,], rin(R,C) \circ tin(T,C) \circ X). \end{array} \tag{35}$$

In the next step, the AC1-unification algorithm terminates successfully with a most general unifier

$$\sigma = \{X \mapsto rin(f,m) \circ Y,\ Z_1'' \mapsto on(R,T) \circ tin(T,C) \circ Y\}.$$

And we obtain the goal:

$$?- causes(Z, [\,], rin(R,C) \circ tin(T,C) \circ rin(f,m) \circ Y).$$

The goal is refuted in one step leading to the computed answer substitution

$$\theta = \{Z \mapsto rin(R,C) \circ tin(T,C) \circ rin(f,m) \circ Y\}$$

with extra condition by 4.3.1 that $(R \neq f \vee C \neq m)$.
Furthermore, it can be computed that $reward(Z\theta) = 10$ since $holds(rin(f,m), Z)$.
Thus, we obtain the abstract predecessor state Z specified by

$$A \stackrel{def}{=} holds(rin(R,C) \circ tin(T,C) \circ rin(f,m), Z)\ \wedge (R \neq f \vee C \neq m).$$

Now consider the second abstract state where the Ferrari is not in Monte Carlo, i.e., $\neg holds\,(rin\,(f,m), Z_1')$ is true. In this case, we obtain:

$$\mathcal{FC} \models causes\,(Z, [loadS\,(R,T)], Z_1') \tag{36}$$

with extra condition that $\neg holds\,(rin\,(f,m), Z_1')$ is true.

To solve (36), we have to find a refutation for:

$$?- causes\,(Z, [loadS\,(R,T)], Z_1').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- on\,(R,T) \circ tin\,(T,C) \circ X =_{AC1} Z_1', \\ \quad causes\,(Z, [\,], rin\,(R,C) \circ tin\,(T,C) \circ X). \end{array} \tag{37}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_1' \mapsto on\,(R,T) \circ tin\,(T,C) \circ X\}$$

with the condition that $\neg holds\,(rin\,(f,m), X)$ is true. And we obtain the goal:

$$?- causes\,(Z, [\,], rin\,(R,C) \circ tin\,(T,C) \circ X).$$

The goal is refuted in one step leading to two computed answer substitutions:

$$\theta_1 = \{Z \mapsto rin\,(R,C) \circ tin\,(T,C) \circ X\}$$

with the condition that $\neg holds\,(rin\,(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$.

Since $\neg holds\,(rin\,(f,m), X)$ is true $reward\,(Z\theta_1) = 0$.

In the second case the computed answer substitution is:

$$\theta_2 = \{Z \mapsto rin\,(R,C) \circ tin\,(T,C) \circ X\}$$

with the condition that $\neg holds\,(rin\,(f,m), X)$ is true and $(R = f \ \wedge \ C = m)$.

Since $holds\,(rin\,(f,m), Z)$ is true $reward\,(Z\theta_2) = 10$.

Thus we obtain two abstract predecessor states Z specified as

$$B \stackrel{def}{=} holds\,(rin\,(R,C) \circ tin\,(T,C), Z) \ \wedge \neg holds\,(rin\,(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m).$$

$$C \stackrel{def}{=} holds\,(rin\,(f,m) \circ tin\,(T,m), Z).$$

Likewise, solving (31) we have to consider two cases. First, when the Ferrari is in Monte Carlo, i.e., $holds\,(rin\,(f,m), Z_2')$ and second, when the Ferrari is not in Monte Carlo, i.e., $\neg holds\,(rin\,(f,m), Z_2')$.

For the first case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes\,(Z, [loadF\,(R,T)], rin\,(f,m) \circ Z_2'') \tag{38}$$

To solve (38), we effectively have to find a refutation for:

$$?- causes\,(Z, [loadF\,(R,T)], rin\,(f,m) \circ Z_2'').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- rin\,(R,C) \circ tin\,(T,C) \circ X =_{AC1} rin\,(f,m) \circ Z_2'', \\ \quad causes\,(Z, [\,], rin\,(R,C) \circ tin\,(T,C) \circ X). \end{array} \tag{39}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a set of most general unifiers

$$\sigma_1 = \{R \mapsto f, C \mapsto m, Z_2'' \mapsto tin\,(T,m) \circ X\}$$

with $\neg holds(rin\,(f,m), X)$ by Condition 4.3.1. And

$$\sigma_2 = \{X \mapsto rin\,(f,m) \circ Y, Z_2'' \mapsto rin\,(R,C) \circ tin(T,C) \circ Y\}$$

with $(R \neq f \vee C \neq m)$ by Condition 4.3.1.

Thus, we obtain two goals:

$$?- causes\,(Z, [\,], rin\,(f,m) \circ tin\,(T,m) \circ X)$$

and

$$?- causes\,(Z, [\,], rin\,(R,C) \circ tin\,(T,C) \circ rin\,(f,m) \circ Y).$$

These goals can be refuted in one step leading to the computed answer substitutions

$$\theta_1 = \{Z \mapsto rin\,(f,m) \circ tin\,(T,m) \circ X\}.$$

with $reward\,(Z\theta_1) = 10$ and

$$\theta_2 = \{Z \mapsto rin\,(R,C) \circ tin\,(T,C) \circ rin(f,m) \circ Y\}$$

with $reward\,(Z\theta_2) = 10$, correspondingly.

Thus, we obtain two abstract predecessor states Z specified by

$$D \stackrel{def}{=} holds\,(rin\,(f,m) \circ tin\,(T,m), Z)$$

and

$$E \stackrel{def}{=} holds\,(rin\,(R,C) \circ tin\,(T,C) \circ rin\,(f,m), Z) \ \wedge (R \neq f \vee C \neq m).$$

Now consider the second case when the Ferrari is not in Monte Carlo, i.e., $\neg holds\,(rin\,(f,m), Z_2')$ is true. In this case, we obtain:

$$\mathcal{FC} \models causes\,(Z, [loadF\,(R,T)], Z_2') \tag{40}$$

with extra condition that $\neg holds\,(rin\,(f,m), Z_2')$ is true.
To solve (40), we have to find a refutation for:

$$?- causes\,(Z, [loadF\,(R,T)], Z_2').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- rin\,(R,C) \circ tin\,(T,C) \circ X =_{AC1} Z_2', \\ \quad causes\,(Z, [\,], rin\,(R,C) \circ tin\,(T,C) \circ X). \end{array} \tag{41}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_2' \mapsto rin\,(R,C) \circ tin\,(T,C) \circ X\}$$

Then we conclude that $\neg holds\,(rin\,(f,m), X)$ is true and obtain the computed answer substitution

$$\theta = \{Z \mapsto rin\,(R,C) \circ tin\,(T,C) \circ X\}$$

with the condition that $\neg holds\,(rin\,(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$.
Since $\neg holds\,(rin\,(f,m), X)$ is true and $(R \neq f \ \vee \ C \neq m)$, $reward\,(Z\theta) = 0$.
Thus we obtain the abstract predecessor state Z specified by

$$F \stackrel{def}{=} holds\,(rin\,(R,C) \circ tin\,(T,C), Z) \ \wedge \neg holds\,(rin\,(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m).$$

Since the abstract states A and E, F and B, D and C are specified in the same way and have the same rewards, we finally end up with three abstract predecessor states.

$$Z_1^l \stackrel{def}{=} holds\,(rin\,(R,C) \circ tin\,(T,C) \circ rin\,(f,m), Z) \ \wedge (R \neq f \vee C \neq m).$$

$$Z_2^l \stackrel{def}{=} holds\,(rin\,(f,m) \circ tin\,(T,m), Z)$$

and

$$Z_3^l \stackrel{def}{=} holds\,(rin\,(R,C) \circ tin\,(T,C), Z) \ \wedge \neg holds\,(rin\,(f,m), Z) \ \wedge (R \neq f \ \vee \ C \neq m).$$

Table 2 depicts the q_1-values for $load(R,T)$-action. Please note that probabilities of a successful or failure loading of a racing car R to a truck T are independent of the fact whether it is raining or not.

abstract pred. state Z	q_1-value
Z_1^l	$10 + .9 \cdot .99 \cdot 10 + .9 \cdot .01 \cdot 10 = 19$
Z_2^l	$10 + .9 \cdot .99 \cdot 0 + .9 \cdot .01 \cdot 10 = 10.09$
Z_3^l	$0 + .9 \cdot .99 \cdot 0 + .9 \cdot .01 \cdot 0 = 0$

Table 2: The q_1-values for the *load* action.

Let us analyze the obtained result. The first abstract predecessor state has the q_1-value 19 because both actions *loadS* and *loadF* will lead to the goal state. Namely, in both cases we will end up with the state in which the Ferrari is in Monte Carlo. The second predecessor state has q_1-value 10.09 due to successful execution of *loadF*-action. And the last predecessor state is given q_1-value of 0 because none of nature's choices of *load*-action successively reaches the goal state.

The last action left to consider is *drive*-action. Performing the decision-theoretic regression through the action $drive(T, C)$, we obtain:

$$\begin{aligned} q_1(drive\,(T,C), Z) \;=\; & reward\,(Z) + \\ & \gamma\, prob\,(driveS\,(T,C), drive\,(T,C), Z)\, v_0(Z_1') + \\ & \gamma\, prob\,(driveF\,(T,C), drive\,(T,C), Z)\, v_0(Z_2') \end{aligned} \tag{42}$$

with

$$\mathcal{FC} \models causes\,(Z, [driveS\,(T,C)], Z_1') \tag{43}$$

and

$$\mathcal{FC} \models causes\,(Z, [driveF\,(T,C)], Z_2'). \tag{44}$$

Starting from the given abstract state space, we compute the possible abstract predecessor states with respect to (43) and (44). Consider first the abstract state where the Ferrari is in Monte Carlo, i.e., $holds\,(rin\,(f,m), Z_1')$ is true. In this case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes\,(Z, [driveS\,(T,C)], rin\,(f,m) \circ Z_1'') \tag{45}$$

and

$$\mathcal{FC} \models causes\,(Z, [driveF\,(T,C)], rin\,(f,m) \circ Z_2''). \tag{46}$$

To solve (45), we effectively have to find a refutation for:

$$?- causes\,(Z, [driveS\,(T,C)], rin\,(f,m) \circ Z_1'').$$

It can be rewritten to:

$$\begin{aligned} ?- \; & tin\,(T,C) \circ X =_{AC1} rin\,(f,m) \circ Z_1'', \\ & causes\,(Z, [\,], tin\,(T,C') \circ X). \end{aligned} \tag{47}$$

In the next step, the AC1-unification algorithm terminates successfully with a most general unifier

$$\sigma = \{X \mapsto rin\,(f, m) \circ Y,\ Z_1'' \mapsto tin\,(T, C) \circ Y\}.$$

And we obtain the goal:

$$?- \ causes\,(Z, [\,], tin\,(T, C') \circ rin\,(f, m) \circ Y).$$

The goal is refuted in one step leading to the computed answer substitution

$$\theta = \{Z \mapsto tin\,(T, C') \circ rin(f, m) \circ Y\}.$$

Furthermore, it can be computed that $reward\,(Z\theta) = 10$ since $holds(rin(f, m), Z)$.
Thus, we obtain the abstract predecessor state Z specified by

$$A \stackrel{def}{=} holds\,(tin\,(T, C') \circ rin\,(f, m), Z).$$

Now consider the second abstract state where the Ferrari is not in Monte Carlo, i.e., $\neg holds\,(rin\,(f, m), Z_1')$ is true. In this case, we obtain:

$$\mathcal{FC} \models causes\,(Z, [driveS\,(T, C)], Z_1') \tag{48}$$

with extra condition that $\neg holds\,(rin\,(f, m), Z_1')$ is true.
To solve (48), we have to find a refutation for:

$$?- \ causes\,(Z, [driveS\,(T, C)], Z_1').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- \ tin\,(T, C) \circ X =_{AC1} Z_1', \\ \quad\ causes\,(Z, [\,], tin\,(T, C') \circ X). \end{array} \tag{49}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_1' \mapsto tin\,(T, C) \circ X\}$$

with the condition that $\neg holds\,(rin\,(f, m), X)$ is true. And we obtain the goal:

$$?- \ causes\,(Z, [\,], tin\,(T, C') \circ X).$$

The goal is refuted in one step leading to the computed answer substitution:

$$\theta = \{Z \mapsto tin\,(T, C') \circ X\}$$

with the condition that $\neg holds(rin(f,m),X)$ is true. Since $\neg holds(rin(f,m),X)$ is true $reward(Z\theta) = 0$.

Thus we obtain the abstract predecessor state Z specified as

$$B \stackrel{def}{=} holds(tin(T,C'),Z) \ \wedge \neg holds(rin(f,m),Z).$$

Likewise, solving (46) we have to consider two cases. First, when the Ferrari is in Monte Carlo, i.e., $holds(rin(f,m),Z_2')$ and second, when the Ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m),Z_2')$.

For the first case, the *holds*-macro can be merged into the *causes*-statement. We obtain:

$$\mathcal{FC} \models causes(Z,[driveF(T,C)],rin(f,m) \circ Z_2'') \tag{50}$$

To solve (50), we effectively have to find a refutation for:

$$?- causes(Z,[driveF(R,T)],rin(f,m) \circ Z_2'').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- tin(T,C') \circ X =_{AC1} rin(f,m) \circ Z_2'', \\ \quad causes(Z,[\,],tin(T,C') \circ X). \end{array} \tag{51}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{X \mapsto rin(f,m) \circ Y, Z_2'' \mapsto tin(T,C') \circ Y\}.$$

Thus, we obtain the goal:

$$?- causes(Z,[\,],tin(T,C') \circ rin(f,m) \circ Y).$$

This goal can be refuted in one step leading to the computed answer substitution

$$\theta = \{Z \mapsto tin(T,C') \circ rin(f,m) \circ Y\}$$

with $reward(Z\theta) = 10$.

Thus, we obtain the abstract predecessor state Z specified by

$$C \stackrel{def}{=} holds(tin(T,C') \circ rin(f,m),Z).$$

Now consider the last case when the Ferrari is not in Monte Carlo, i.e., $\neg holds(rin(f,m),Z_2')$ is true. In this case, we obtain:

abstract pred. state Z	q_1-value
Z_1^d	$10 + .9 \cdot .99 \cdot 10 + .9 \cdot .01 \cdot 10 = 19$
Z_2^d	$10 + .9 \cdot .99 \cdot 0 + .9 \cdot .01 \cdot 0 = 0$

Table 3: The q_1-values for the *drive* action.

$$\mathcal{FC} \models \mathit{causes}\,(Z, [\mathit{driveF}\,(T, C)], Z_2') \tag{52}$$

with extra condition that $\neg \mathit{holds}\,(\mathit{rin}\,(f, m), Z_2')$ is true.

To solve (52), we have to find a refutation for:

$$?- \mathit{causes}\,(Z, [\mathit{driveF}\,(T, C)], Z_2').$$

Within two steps, this goal can be reduced to:

$$\begin{array}{l} ?- \mathit{tin}\,(T, C') \circ X =_{AC1} Z_2', \\ \quad \mathit{causes}\,(Z, [\,], \mathit{tin}\,(T, C') \circ X). \end{array} \tag{53}$$

In the next step, the AC1-unification algorithm is called and terminates successfully with a most general unifier

$$\sigma = \{Z_2' \mapsto \mathit{tin}\,(T, C') \circ X\}.$$

Then we conclude that $\neg \mathit{holds}\,(\mathit{rin}\,(f, m), X)$ is true and obtain the computed answer substitution

$$\theta = \{Z \mapsto \mathit{tin}\,(T, C') \circ X\}$$

with the condition that $\neg \mathit{holds}\,(\mathit{rin}\,(f, m), X)$ is true.

Since $\neg \mathit{holds}\,(\mathit{rin}\,(f, m), X)$ is true, $\mathit{reward}\,(Z\theta) = 0$.

Thus we obtain the abstract predecessor state Z specified by

$$D \stackrel{def}{=} \mathit{holds}\,(\mathit{tin}\,(T, C'), Z) \ \wedge \neg \mathit{holds}\,(\mathit{rin}\,(f, m), Z).$$

Since the abstract states A and C, B and D are specified in the same way and have the same rewards, we finally end up with two abstract predecessor states.

$$Z_1^d \stackrel{def}{=} \mathit{holds}\,(\mathit{tin}\,(T, C') \circ \mathit{rin}\,(f, m), Z).$$

$$Z_2^d \stackrel{def}{=} \mathit{holds}\,(\mathit{tin}\,(T, C'), Z) \ \wedge \neg \mathit{holds}\,(\mathit{rin}\,(f, m), Z).$$

Table 3 depicts the q_1-values for the $\mathit{drive}(T, C)$-action. Please note that probabilities of a successful or failure driving truck T to a city C are independent of the fact whether it is raining

or not. Moreover, please note that the result we have obtained is very natural. The state in which the Ferrari is in Monte Carlo can be reached by performing the action $drive(T, C)$ only if the Ferrari has already been in Monte Carlo before the execution of an action. Conversely, the state in which the Ferrari is not in Monte Carlo can be reached by performing the action $drive(T, C)$ only if the Ferrari has not been in Monte Carlo before the execution of an action.

We have computed the q_1-values for all actions. Following the value iteration algorithm, in the next section we are going to choose the optimal action.

4.4 Symbolic Dynamic Programming

The value iteration algorithm within the fluent calculus is very similar to the one for the situation calculus with the difference that we operate on states instead of situations. Namely,

$$\begin{gathered} v_n(Z) = W \leftrightarrow \\ [\bigvee_a (\exists \overline{X}).q_n(a(\overline{X}), Z) = W] \wedge \\ [\bigwedge_{a'} (\forall Y, W').q_n(a'(\overline{Y}), Z) = W'] \rightarrow W' \leq W, \end{gathered} \tag{54}$$

where Z is a state. In other words, the value function $v_n(Z)$ applied to a state Z has the value W iff there exists an action a with the q-value W and for all other actions a' having the q-value W', we find that W' is less or equal than W.

Recalling our running example, any action of *unload*, *load*, or *drive* can be chosen because all of them have an abstract state which has the maximum q_1-value, i.e., 19. If we choose *unload*-action then the predecessor abstract state is:

$$Z_3^u \stackrel{def}{=} holds\,(on\,(R, T) \circ tin\,(T, C) \circ rin\,(f, m), Z).$$

If *load* then the state is described as:

$$Z_1^l \stackrel{def}{=} holds\,(rin\,(R, C) \circ tin\,(T, C) \circ rin\,(f, m), Z) \ \wedge (R \neq f \vee C \neq m).$$

And finally, if we opt for *drive* then the specification of the predecessor state is:

$$Z_1^d \stackrel{def}{=} holds\,(tin\,(T, C') \circ rin\,(f, m), Z).$$

From a theoretical point of view, value iteration algorithm in its pure style is not sensitive to exploration. That is, it will eventually converge to the right values independent on the action selection, given that there are indefinitely many learning trials. There is a large variety of action selection and exploration methods [4, 10]. As the agent needs to explore the search space during learning, one may use a randomized action selection strategy, e.g., take the action

---

$n := 0$.
Specify $\varepsilon > 0$.
Initialize $v_0(Z) = reward(Z)$
`loop`
    $n := n + 1$.
    For each action type $a(\overline{X})$ compute $q_n(a(\overline{X}), Z)$ using $v_{n-1}(Z')$
        where $Z' \in \mathcal{Z}$ and satisfies the condition
        $\mathcal{FC} \models causes(Z, [a_j(\overline{X})], Z')$ for every nature's choice $a_j(\overline{X})$.
    Compute $v_n(Z)$ using $q_n(a(\overline{X}), Z)$ as in Equation (54).
`until` $|v_n(Z) - v_{n-1}(Z)| < \varepsilon$ for all abstract states $Z \in \mathcal{Z}$.

Figure 10: The first-order value iteration algorithm.

with the highest value with a probability p, and take a random action with the probability $(1 - p)$.

Coming back to our example, in order to have more certainty about the optimal action choice one may associate the probability to each action. These probabilities will denote the probabilities of choosing an action at the particular state. Since we have not introduced the concept of action choice probabilities in our approach, we will have three optimal actions and the value function $v_1(Z)$ will be equal to 19. One should note that we have assumed that the termination condition, i.e., $|v_n(Z) - v_{n-1}(Z)| < \varepsilon$, has been successively met and our algorithm terminates after a single iteration step.

The algorithm of the first-order value iteration within the fluent calculus is summarized on the Figure 10. This algorithm together with the first-order decision-theoretic regression for computing q-values provides the ability to construct a sequence of value functions in a symbolic way, obviating state and action space enumeration. As a result, the state space is partitioned into abstract states which are specified symbolically within the fluent calculus. Moreover, the presented algorithm produces an abstraction from the action space, eliminating the need in domain grounding. The later is a crucial problem of the existing decision-theoretic algorithms because the number of propositions grows dramatically with the number of domain objects. Having computed the values of q-function and the optimal value function, the extraction of an optimal action is rather straightforward. In order to get an optimal action (actions) one has to take an action (actions) which has (have) the maximum q-value, or in other words, an action which corresponds to the optimal value function.

5 Summary and Future Work

We have provided a symbolic dynamic programming approach for modelling first-order Markov decision processes within the fluent calculus. We have addressed the scalability problem of the classical DP concept by partitioning the state space into "abstract states". The partitioning is done according to the conditions represented as the first-order formulae produced by the decision-theoretic regression. Furthermore, by exploiting the logical structure of our approach, we avoid domain grounding.

Our approach is very similar to one presented within the situation calculus [1] but is still distinguished by the following features. First, for the situation calculus, or to be precise for GOLOG [17] the closed-world assumption is applied to the initial state specification. In comparison with the situation calculus, the version of the fluent calculus presented here allows specifying and computing with an incomplete state knowledge. This fact is reflected in the symbolic value iteration algorithm. See, for example, the definition of the nature's choices probabilities

$$prob(unloadS(R,T), unload(R,T), Z) = .7 \leftrightarrow (\exists Z').\ Z = rain \circ Z'$$

or the rewards' definition

$$reward(Z) = 10 \leftrightarrow (\exists Z').\ Z = rin(f,m) \circ Z'.$$

Second, the version of the fluent calculus presented here does not introduce such notion as "situation" (as in case of the situation calculus) which, in fact, is not needed, because the classical value iteration algorithm works on states. More precisely, the predicate $causes/3$, which describes the transformation of one state into the another and reflects the backward nature of the value iteration algorithm, operates on states. This feature of the fluent calculus allows us the rather direct translation from the classical value iteration algorithm to its symbolic counterpart. As a result, the first-order representation of the Q-functions and the value functions within the fluent calculus is given over states. See, for example, Equation 15, where Z and Z' represent the states.

Moreover, the ability to operate on states directly turns to be very efficient when computing the value of a fluent. In the fluent calculus the value is readily available from the list of fluents. In the situation calculus the current situation term should be unfolded either until the situation is reached where the fluent was caused true or false by the preceding action or until the initial situation. Only after such unfolding we will have access to the value of the fluent.

Although some restrictions were made, the example given illustrates the ability of using the fluent calculus for decision-theoretic regression and inspires to obviate these restrictions developing a more realistic presentation. A number of interesting directions remains to be explored. "Computer-based" simplification techniques not used so far because of their complexity will definitely enhance the implementation. As an alternative direction of the future work, we may concentrate on integrating the presented decision-theoretic regression algorithm for the

fluent calculus with powerful first-order theorem provers to enhance the performance of the algorithm. Finally, it looks promising to investigate further the possibility of using the version of the fluent calculus presented here decorating it by such practical features as sensing, that is a treatment of sensor outputs, ramifications, i.e., indirect effects, and qualifications caused by some exogenous events.

References

[1] Craig Boutilier, Ray Reiter, and Bob Price. Symbolic dynamic programming for First-Order MDPs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 690-700, San Francisco, CA, August 4-10 2001. Morgan Kaufmann Publishers, Inc.

[2] R.E.Bellman. Dynamic Programming. *Princeton University Press*, Princeton, 1957.

[3] M.L.Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming, *Wiley*, 1994.

[4] L.P.Kaelbling, M.Littman, A.W.Moore. Reinforcement Learning: A Survey. *AI Access Foundation and Morgan Kaufmann Publishers*, 1996.

[5] J.McCarthy. Situations, actions and causal laws. Tech.Report, Stanford University, 1963. Repr. *Semantic Information Processing*, MIT Press, Cambridge, 1968, 410-417.

[6] R. Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In Vladimir Lifschitz, editor,*Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, p. 359-380. Academic Press, San Diego, CA, 1991.

[7] C.Boutilier, T.Dean, and S.Hanks. Decision Theoretic Planning: Structural Assumptions and Computational Leverage, *Journal of AI Research (JAIR) 11:1–94*, 1999.

[8] C.Boutilier,R.Reiter, M.Soutchanski, and S.Thrun. Decision-theoretic, High-level Agent Programming in the Situation Calculus. *AAAI-2000*, Austin, TX, 2000.

[9] S.Russell, P.Norvig. Artificial Intelligence: A Modern Approach. *Prentice Hall*, 1995.

[10] R.S.Sutton,and A.G.Barto. Reinforcement learning: An Introduction. *MIT Press*, 1998.

[11] S.Hölldobler, J.Schneeberger. A New Deductive Approach to Planning. *New Generation Computing 8:225-244*, 1990.

[12] M.Thielscher. The Fluent Calculus: A Specification Language for Robots with Sensors in Nondeterministic, Concurrent, and Ramifying Environments. 2000.

[13] T.Dean, J.Allen, Y.Aloimonos. Artificial Intelligence: Theory and Practice. *Benjamin Cummings*, 1995.

[14] T.Dean, M.Wellman. Planning and Control. *Morgan Kaufmann*, San Mateo, California, 1991.

[15] S.French. Decision Theory. *Halsted Press*, New York, 1986.

[16] G.Dantzig, P.Wolfe. Decomposition principle for dynamic programs: Operations Research. 1960.

[17] H.J.Levesque, R.Reiter, Y.Lespérance, F.Lin, and R.Scherl. GOLOG:A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59-83, 1997.