libPRADA – Version 1.0
Learning and Planning with Probabilistic Relational Rules

Tobias Lang
http://userpage.fu-berlin.de/tlang/prada/

May 9, 2012

When using this library, please cite Lang and Toussaint [2010].

Contents

1 Disclaimer

libPRADA is work in progress! I did my best to avoid mistakes and to increase comprehensibility (at least at high-level interfaces). Nonetheless, you will probably encounter bugs and lacks in functionality. **Please contact me in this case!** I'll be happy to fix bugs, hear your complaints, add desired functionality and implement proposals and wishes for the library :-).

2 Introduction

This library provides a C++-implementation for using probabilistic relational rules in the context of model-based relational reinforcement learning in stochastic domains.

- libPRADA provides algorithms for **planning** in stochastic relational domains: PRADA Lang and Toussaint [2010], sparse sampling trees (SST) Kearns et al. [2002] and UCT Kocsis and Szepesvari [2006].
- libPRADA implements the algorithm for **learning** probabilistic relational rules by Pasula et al. [2007].
- Consequently, libPRADA provides plenty of basic data-structures and methods to deal with relational logic objects such as symbols (predicates), symbolic states and rules.

3 Installation

First, please build the **library**:

- cd lib
- type make, and pray

Then, please run the **example programs**:

- cd ../test/simple_demos/
- type make, and pray again
- ./x.exe
- cd ../test/plan/
- type make, and pray once again
- ./plan.exe

Finally, you might want to take a closer look at this guide.

If compiling the library did *not* work out, then you can compile the whole example programs directly in their directories. For example:

- cd test/demos/
- make -f Makefile_direct

If this still does not work, please contact me.

4 Programmer's guide

We will discuss the important headers in `libPRADA/src/`.
File overview:

- `MT/array.h`: data-structure for lists
- TODO
- `relational/reason.h`: manipulating and processing basic logical objects and rules
- `relational/plan.h`: planning with NID rules including SST and UCT
- `relational/prada.h`: PRADA for planning with NID rules
- `relational/learn.h`: learning NID rules from data

4.1 Lists

File: `MT/array.h`

I use Marc Toussaint's `Array` class to store all kinds of data (primitive data-types, objects, pointers) in lists. Likewise, it can be used to define mathematical objects such as vectors and matrices. For instances, a list of `double` becomes a vector of reals. His class provides many basic self-explanatory methods to deal with lists, vectors and matrices.

4.2 Symbols

File: `relational/symbols.h`

Data structure A symbol consists of the following attributes:

```
MT::String name;
uint arity ;    // e.g. 2 for on(.,.)
SymbolType symbol_type;
RangeType range_type;
uintA range;    // optional
ArgumentTypeL arg_types; // optional
```

`SymbolType` is an enum in `Symbol`. The most important types are `action` and `primitive` which constitute the basic predicates and functions in a domain. The truth of primitive symbols needs to be provided from outside the logical machinery (e.g., by compiling observations into logical symbols, also known as the "grounding" problem).

All other types denote symbol types which are defined in terms of other symbols. For instance, a `conjunction` symbol is $clear(X) = \forall Y \neg on(Y, X)$. `Symbol.h` provides implementations for these more complex symbol types. They should be largely self-explanatory. You can also add your own symbol types with appropriate definitions. Please note that these derived symbols can be used to specify complex logical reward functions (see also below). For instance, it is possible to thereby define a reward for stacking large towers of blocks in a blocksworld. Here are some of the provided symbol types:

- `ConjunctionSymbol`: a conjunction of (possibly negated) symbols, allowing for either existentially or universally quantified variables; example: $clear(X) = \forall Y \neg on(Y, X)$
- `TransClosureSymbol`: transitive closure for a symbol; example $above(X, Y) = +on(X, Y)$
- `CountSymbol`: counts how often a symbol holds; example $countClear() = \sharp X : clear(X)$
- `AverageFunction`: computes the average over function values; example $averageHeight() = average_X[height(X)]$
- `SumFunction`: computes the sum over function values; example $sumHeight() = \sum_X height(X)$
- `MaxFunction`: computes the max over function values; example $maxHeight() = \max_X height(X)$

`RangeType` denotes the range of a symbol. Symbols can have a `binary` range and correspond to standard predicates then: they are either true (value 1) or false (value 0). Symbols can be functions which have the `integers` or the `reals` as range. It is also possible to define a `integer_set` for a specific set of range values: this set needs to be defined in `range`.

Finally, libPRADA also allows for typed arguments by means of the data-structure `ArgumentType`. For instance, one may want to define a type `cube` for constants. However, this is still work in progress. (Alternatively, one may use typing predicates such as $cube(\cdot)$.)

File syntax The syntax for a symbols file is simply a list of symbols and (optionally) argument types as follows:

```
<Symbol>+
{ [    // optional
<ArgumentType> +
] }
```

A single symbol is represented by its spelled out attributes `(name) (arity) (type) (range) [otherstuff]`. `[otherstuff]` is needed to define derived symbols. The exact syntax depends strongly on the type of the derived symbol. Please see the example files in the `test`-directories. Here is an example for symbols file:

```
grab 1 action binary
on 2 primitive binary
size 1 primitive integers
clear 1 conjunction binary <-- All Y -on(Y X) // conjunction
```

TODO more examples here; range for functions

Object Management For object management reasons, C++-objects of symbols cannot be constructed from outside the class. Rather, the following method needs to be called to construct a symbol:

```
static Symbol* get(const MT::String& name, uint arity, SymbolType
symbol_type = primitive, RangeType range_type = binary);
```

Analogous `get`-methods exist for the derived symbols. Thereafter, symbols can also simply be accessed by their names:

```
static Symbol* get(const MT::String& name);
```

4.3 Literals

File: `relational/literals.h`

Literals are instantiated symbols (that is, with a list of arguments) together with a value. Examples: $on(61, 63) = 1$ and $size(62) = 4$.

4.3.1 Arguments: logic variables and constants

Arguments of literals are represented by non-negative integers (`uint`) $0, 1, 2 \ldots$. For reasoning, we need to distinguish between logic variables and constants. This is achieved by reserving a set of `uints` for the constants (`uintA = MT::Array< uint >`). You can find the required methods in `relational/reason.h`:

```
void setConstants(uintA& constants);
bool isConstant(uint id);
```

The default is that all `uints` < 10 are variables, all other constants.

There are also appropriate methods to fix the `ArgumentType` of constants.

4.3.2 Literals

A literal consists of the following attributes:

```
Symbol* s;
uintA args;
double value;
ComparisonType comparison_type; // default: comparison_equal
```

`uintA` is short for `MT::Array<uint>` (a list of unsigned integers). For binary symbols (aka predicates), the value 0 represents a negated (/negative/false) symbol and the value 1 a true (/positive) symbol. For instance, for $on(61, 63) = 1$ we have `s = Symbol::get(''on'')`, `args(0)=61`, `args(1)=62`, `value=1` and `comparison_type = Literal::comparison_equal`.

`ComparisonType` is an enum in `Literal` and represents the different comparison forms $=, <, \leq, >, \geq$. Hence, we can define comparisons such as $size(62) < 5$.

File syntax Simply use **plain text** descriptions:

```
on(65, 60)   // for predicates: corresponds to value=1
-inhand(71) // for predicates: corresponds to value=0
size(1)<=2
size(66)=2
```

Please note the special notation for the value of binary symbols (= predicates) which is close to standard logic notation: if it is omitted, it denotes value 1 (= true); with a leading −, it denotes value 0 (=false).

Lists of literals are usually written in one line, separated by a space or by a comma.

4.3.3 SymbolicState

A `SymbolicState` consists of:

```
MT::Array<Literal*> lits;
uintA state_constants;
bool derivedDerived;
```

The important attribute is the list of literals `lits`. We make the closed world assumption: `lits` contains binary literals only if they are positive (with value 1). Hence, all binary literals which are not explicitly stated are assumed to be false. `state_constants` can be set optionally and contains all constants in the state (note that not necessarily all state constants need to appear as an argument in `lits`). `derivedDerived` is a flag which stores whether the derived symbols have already been calculated.

File syntax: States and Trials A state is simply represented by its list of literals:

```
on(61,62) on(65,64) inhand(66) size(64)=3
```

Please note the special syntax for literals of binary symbols (predicates) as described above: no negative binary symbols are allowed and positive binary symbols don't have a value specified (no `=1`).

4.3.4 StateTransition

`StateTransition` is a convenience wrapper for the reinforcement learner's favorite triplet (s, a, s'):

```
SymbolicState pre, post;
Literal* action;
MT::Array< Literal* > del, add;
uintA changedConstants;
```

`del`, `add` and `changedConstants` are calculated automatically by the constructor.

4.4 Rules

File: `relational/rules.h`

Probabilistic relational rules are at the heart of libPRADA. They provide a transition model $P(s' \mid s, a)$ for model-based relational reinforcement learning.

4.4.1 Rules

The data-structure `Rule` implements the noisy indeterministic deictic (NID) rules of Pasula et al. [2007]. The semantics of NID rules follows exactly their paper. Please confer their paper for further details (or my Ph.D. thesis Lang [2011] :-)), including state-action coverage and the noise outcome. In particular, it is important for you to understand when a rule covers a state-action pair and when not (I'll describe this superficially also below): the concepts of a rule uniqueness and of noisy default rule are important.

`Rule` has the following attributes:

```
Literal* action;
LitL context;
MT::Array< LitL > outcomes;
doubleA probs;
double noise_changes;
arr outcome_rewards; // optional
```

Please note that `LitL` is short for `MT::Array< Literal* >`, a list of `Literal` pointers.

`action` is the rule's action (surprise, surprise). `context` is the list of (abstract) literals which need to be covered by a state so that the rule can apply. `outcomes`

contains the different outcomes, `probs` the corresponding probabilities. Please note an important convenience for `outcomes` and `probs`: the last outcome is the noise outcome. `noise_changes` is PRADA's noise outcome heuristic: it states the average number of state properties that have changed in case of the noise outcome – however, in practice, this is a negligible parameter. `outcome_rewards` is an optional parameter which specifies a reward for each outcome: planners like PRADA take these in account in addition to global rewards on states (see the IPPC domains for example domains).

`Rule` provides many convenience methods. It also provides a method to construct the **noisy default rule** which is important when learning and reasoning with NID rules (see Pasula et al. [2007]):

```
static Rule* generateDefaultRule(double noiseProb,
             double minProb, double change);
```

The default rule uses the special action $default()$ and is applied when there is no unique non-default covering rule for a state-action pair.

Sets of rules should be managed by means of the class `RuleSet`. Most importantly, `RuleSet` controls the deletion of C++-objects of `Rule` so that your working memory does not drown in `Rule` pointers. There is also an additional container structure `RuleSetContainer` for `RuleSet` in `learn.h` which is used for efficiency reasons in rule-learning and provides auxiliarly methods and statistics.

File Syntax: rules The general syntax is straightforward:

```
ACTION:
<Literal>
CONTEXT:
<Literal>+
OUTCOMES:                // List of outcomes
<double> <Literal>+      // Outcome 1 incl. probability
<double> <Literal>+      // Outcome 2 incl. probability
...
```

Example:

```
ACTION:
  puton(X)
CONTEXT:
  block(X), inhand(Y), size(X)==2
OUTCOMES:
  0.7 on(Y X), upright(Y), -inhand(Y)
  0.2 -inhand(Y)
  0.1  <noise>
```

Some guidelines for rules:

- Outcomes are lists of *primitive* literals with a leading probability. (No literals for derived symbols here! Derived literals, however, may appear in the context.)
- The last outcome of a rule *must* be the noise outcome.
- The noise outcome can specify how many (random) state properties are expected to change (required by PRADA's heuristic to deal with the noise outcome).

4.4.2 Substitution

A `Substitution` maps integers to integers. It can be used for instance for grounding abstract literals, then mapping variables to constants (please recall that both variables and constants are represented by `uint`). The essential two attributes are:

```
uintA ins;
uintA outs;
```

These are two lists of `uints`. `ins(i)` maps to `outs(i)`. `Substitution` provides many (hopefully) self-explanatory methods. The most important ones are the various `apply` methods such as applying substitutions to rules and `void addSubs(uint in, uint out)` for adding a substitution.

Sets of substitutions can be managed by means of the container class `SubstitutionSet`. Similarly as for `RuleSet` the major purpose is to control the deletion of C++-objects of `Substitution`.

4.5 Reasoning

File: `relational/reason.h`

The namespace `reason` provides methods for logical reasoning. These methods are broadly into **basic** reasoning and **rule reasoning** methods.

4.5.1 Basic reasoning

Basic reasoning methods fulfill three major functionalities: (i) distinguishing between constants and variables; (ii) deriving literals for derived symbols (symbols which are defined in terms of other symbols); (iii) basic coverage methods.

Distinguishing between constants and variables As described in Sec. 4.3.1, arguments of literals (variables and constants) are represented by `uints`. `reason` maintains the information which `uint` refers to a variable and which to a constant. By default, all `uints` $<$ 10 refer to variables, all other to constants. Alternatively, you may provide a set of constants by `void setConstants(uintA& constants)`. The fundamental methods of libPRADA for distinguishing ground and abstract literals are:

```
bool isGround(const Literal* lit );
bool isPurelyAbstract(const Literal* lit );
```

Please note that for a literal to be purely abstract, all arguments need to be variables.

Deriving literals There are primitive and derived (/non-primitive) symbols. Derived symbols are defined in terms of other symbols. To describe the world symbolically, the truth of literals for primitive symbols needs to be specified from outside the logical machinery. This is the grounding problem: how do symbol relate the true world? Please note that this is different from *grounding* an abstract literal to a ground literal. In contrast, the literals for derived symbols need to be calculated from other

literals using logical reasoning. `reason` provides the required methods. The central method for calculating the derived literals for a state is:

```
void derive(SymbolicState* s);
```

Basic coverage methods There are three types of basic coverage methods which are based on each other. The `holds` methods check for test literals whether they hold in a given list of literals (simple contains check). The `unify` methods try to unify different literals (typically, abstract and ground literals): if successful, the resulting `Substitution` provides the mapping of variables to variables/constants. The `cover` methods try to unify *lists* of literals and return lists of appropriate substitutions.

4.5.2 Rule reasoning

The methods for rule reasoning fulfill the following functionalities: (i) distinguishing between ground and abstract rules; (ii) calculating successor states and their probabilities for a rule and a given state-action pair; (iii) calculating the coverage of rules; (iv) calculating likelihoods of experiences (s, a, s') for rule-sets.

Coverage of rules The implementation of state-action coverage for NID rules in libPRADA follows directly the specification in Pasula et al. [2007]. It has a specific semantics which might be unexpected. Understanding this semantics is crucial for learning and planning with NID rules. Please see Pasula et al. [2007] and Lang and Toussaint [2010] for details.

I highlight the most important feature here: For a given state-action pair (s, a) and a rule-set Γ, we check which rules in Γ cover (s, a). That is, we substitute the arguments in the abstract rule action with the constants in a. Then, we try to find a unique substitution for the remaining variables in the rule. These remaining variables which do not appear in the action's arguments are called deictic references. If there is exactly one non-default rule in Γ covering (s, a) we call it the unique covering rule and use it to model $P(s' \mid s, a)$. If there is no such unique covering (there is no covering rule or there are at least two covering rules), we use the default rule, basically saying that we do not know what will happen.

There is a further subtlety concerning the deictic references: for a rule to cover (s, a) there needs to be a unique substitution for these references. Thus, if there several groundings for a deictic reference such that the rule's context holds then the rule does *not* cover (s, a).

Since this is so important, I give a small example here. Consider the rule:

```
ACTION:
  puton(X)
CONTEXT:
  block(X), inhand(Y)
OUTCOMES:
  0.9 on(Y X), −inhand(Y)
  0.1  <noise>
```

Y is a deictic reference here. Consider the symbolic states $s_1 = \{block(a), inhand(b)\}$, $s_2 = \{block(a)\}$, $s_3 = \{block(a), inhand(b), inhand(c)\}$. The rule covers only s_1: there is the unique grounding $\{X \to a, Y \to b\}$. However, it does not cover s_2: Y cannot be resolved. Most importantly, the rule does not cover s_3, either: there is no unique grounding. The reason is that the deictic reference Y cannot be resolved uniquely: there are two possible groundings $\{X \to a, Y \to b\}$ and $\{X \to a, Y \to c\}$.

4.6 Planning

Files: `reasoning/plan.h`, `reasoning/prada.h`

This is a core part of libPRADA. It provides four algorithms (PRADA, A-PRADA, SST, UCT) for planning with *ground* NID rules. The central methods for any planner are:

```
Literal * plan_action(const SymbolicState& current_state);
void setReward(Reward*);
void setGroundRules(RuleSet& ground_rules);
```

The first action lets the planner plan an action for a given state, yielding a policy $\pi : s \to a$. The planner tries to find the approximately best action leading to high rewards. PRADA and A-PRADA also provide a method to generate a complete plan. In contrast, SST and UCT cannot provide a complete plan due to their outcome sampling. The second method sets the planning reward function of the planner (see below). The third method sets the *ground* rules which provide the transition model $P(s' \mid s, a)$ used by the planner. Hence, a set of abstract NID rules needs to be grounded first (see `reasoning/reason.h`) with respect to the domain constants; the ground rules are then provided to the planner.

All planning algorithms share the following parameters:

```
double discount;
uint horizon;
double noise_scaling_factor;
```

`discount` is a discount factor $\gamma \in (0, 1]$ (a future reward at time t is discounted by γ^t). `horizon` is the planning horizon $h > 0$. Specific parameters for the individual planners are discussed below. `noise_scaling_factor` is a noise scaling factor η used by SST and UCT: it scales down the future values when sampling the noise outcome.

4.6.1 Individual planners

PRADA PRADA stands for "probabilistic relational action-sampling in dynamic Bayesian networks planning algorithm" Lang and Toussaint [2010].

The most important parameter for PRADA is `num_samples`: PRADA samples action-sequences and evaluates their rewards using a dynamic Bayesian network (DBN) `PRADA_DBN* dbn` for all ground symbols. `num_samples` controls the number of samples: the more samples, the higher the probability to find good plans and the higher planning complexity.

Furthermore, you may specify `threshold_reward` $\in (0, 1]$ to define what a "good" plan is: it sets the threshold on the probability to achieve the reward. Another parameters is `noise_softener` $\in (0, 1]$: there is no clear way

how to deal with the noise outcome of rules; PRADA's heuristic to do so can be harmful if the noise outcome has too high probability; this parameter may reduce the noise outcome's effects

A-PRADA Adaptive-PRADA extends PRADA as described in Lang and Toussaint [2010]. The important method called during planning is

```
double shorten_plan(LitL& seq_best, const LitL& seq, double value_old);
```

It takes a plan and examines whether this plan can be improved by deleting some actions.

SST SST stands for "sparse sampling tree" (SST) algorithm [Kearns et al., 2002]. SST is used for planning with NID rules in the work by Pasula et al. [2007]. It has the following additional parameter:

- `branch`: branch b which determines the number of samples from the successor state distribution for a given action

UCT UCT stands for "upper confidence bounds applied to trees" [Kocsis and Szepesvari, 2006]. It has the following two additional parameters:

- `c`: bias c for less often explored actions
- `numEpisodes`: number of episodes (or rollouts) e

4.6.2 Reward functions

Reward functions $R : S \to A$ (here: from states to actions) are modelled by the class `Reward`. `Reward` provides the following methods to evaluate states:

```
double evaluate(State& s);
bool satisfied(State& s);
bool possible(State& s);
```

Methods two and three may have trivial implementations by always returning true.

The following pre-defined reward types are provided by libPRADA:

- LiteralReward: the reward is given for achieving a single literal, e.g. $inhand(a)$ or $inhand(X)$
- LiteralListReward: the reward is given for achieving a conjunction of literals, e.g. $on(a, b), on(b, c)$
- MaximizeReward: the value of an atom shall be maximized, e.g. $sumHeight()$ (this defines the stacking task in good, old blocksworld)

PRADA reasons on beliefs over states (rather than on states directly). For this purpose, PRADA maintains its own class `PRADA_Reward`. For the three reward functions discussed above, automated conversion routines are used when setting the standard `Reward` for PRADA. If you come up with your own reward function type, however, you must also define a `PRADA_Reward` type that implements evaluating this reward function over beliefs.

File syntax: rewards Simply a flag of the reward type plus the literal(s) as described above.

LiteralReward:

```
1
<Literal>
```

LiteralListReward:

```
2
<Literal>+
```

MaximizeReward:

```
3
<Literal>
```

4.7 Rule learning

File: `relational/learn.h`

libPRADA provides a direct implementation of the learning algorithm for probabilistic relational rules by Pasula et al. [2007].

To learn rules, you need to come up with a set $\{(s, a, s')\}$ of state transitions (s, a, s') using the data-structure `StateTransition` (see 4.3.4). The central method is:

```
void learn_rules(RuleSetContainer& rulesC,
        StateTransitionL& experiences,
        const char* logfile);
```

For efficient rule-learning the data-structure `RuleSetContainer` is used: `RuleSetContainer` is a wrapper for `RuleSet` which stores information on covered state-transitions.

The learning algorithm is very sensitive to two parameters: the noise penalty α and the lower bound p_{min} on the probability on the noise outcome (see Lang and Toussaint [2010], Pasula et al. [2007] for details). You can set them with these methods:

```
void set_penalty(double alpha_PEN);
void set_p_min(double p_min);
```

You need to get a feeling for these parameters to be able to learn rules which you consider "good". This cannot be set automatically because what is "good" depends on your prior knowledge (for instance, you must choose whether you want a very accurate and complex model or a compact model only for typical state transition). I discuss this briefly in my thesis Lang [2011].

The learning algorithm is a heuristic search through the space of rule-sets based on search operators. You may adapt the algorithm to your needs by defining your own search operator, deriving from `SearchOperator`. Which operator is tried at each time-step is determined by the method

```
void set_ChoiceTypeSeachOperators(uint choice_type);
```

The random choice takes into account sampling weights for the search operators. Hence, you may change the algorithm by giving different weights to the search operators.

5 User's guide

To use libPRADA successfully, you have to understand: *(i)* how libPRADA represents symbols, literals, states and rules (`relational/symbols.h`, `relational/literals.h`, `relational/rules.h`); *(ii)* how to set up properly a planning scenario (`relational/plan.h`, `relational/prada.h`); *(iii)* how to set up properly a rule learning procedure (`relational/learn.h`).

This is demonstrated in three tests: `tests/relational_basics`, `tests/relational_plan` and `tests/relational_learn`. Please take a look at the `main.cpp` of the corresponding test.

5.1 Planning by programming

Take a look at the two demos in `libPRADA/test/simple_demos/main.cpp`. The first demo `logicDemo` shows you how to define your own relational logic language and rules. The second demo `planDemo` shows you how to plan with either of SST, UCT, PRADA or A-PRADA. It uses the robot manipulation domain of the experiments presented in Lang and Toussaint [2010].

In the following, I will summarize the most important steps to set up a planning problem:

1. Create your relational logic **language**: primitive and derived predicates and functions
2. Define your set of **constants** (/objects)
3. Initialise the `logicObjectManager` with the language and constants
4. Create a **starting state** for planning
5. Define a **reward function**
6. Create **rules**
7. **Ground** all rules
8. Set up the **planners**
9. Provide rules, reward and all necessary **parameters** to your planner
10. Finally, **plan** :-) !

5.2 Planning by intermediate files

Just go to `libPRADA/test/plan/` and compile the program. By means of the config file `config`, you can specify all necessary parameters for your planning scenario, including the files for rules, starting states and rewards. Then type `./x.exe` and a (hopefully great) plan will be yours.

5.3 Rule learning

Just go to `libPRADA/test/rule_learning/` and compile the program to build `x.exe`. The file `samples.dat` contains a trial (transitions of state, action, successor state) from which rules can be learned. Please check `main.cpp` for how to set parameters. The important parameters are the regularizer α (`alpha_pen`) and the lower bound for states in the noise outcome p_{min} (`p_min`).

6 Trouble shooting

Learning: parameter

Planning: Per Hand ausrechnen, ob die Regeln wirklich so sind, wie sein sollen. Haeufig anders. Wahrscheinlichkeiten Horizont

7 Future research

Limitations of NID rules and PRADA

What you can research on

References

Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Machine Learning*, 49(2-3):193–208, 2002.

Levente Kocsis and Csaba Szepesvari. Bandit based monte-carlo planning. In *Proc. of the European Conf. on Machine Learning (ECML)*, 2006.

Tobias Lang. *Planning and Exploration in Stochastic Relational Worlds*. PhD thesis, Fachbereich Mathematik und Informatik, Freie Universität Berlin, 2011.

Tobias Lang and Marc Toussaint. Planning with noisy probabilistic relational rules. *Journal of Artificial Intelligence Research*, 39:1–49, 2010.

Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29:309–352, 2007.