Robust Learning for Adaptive Programs by Leveraging Program Structure

Jervis Pinto, Alan Fern, Tim Bauer, and Martin Erwig
School of Electrical Engineering and Computer Science
Oregon State University, Corvallis, OR 97331, USA
{*pinto,afern,bauertim,erwig*}*@eecs.oregonstate.edu*

***Abstract*—We study how to effectively integrate reinforcement learning (RL) and programming languages via adaptation-based programming, where programs can include non-deterministic structures that can be automatically optimized via RL. Prior work has optimized adaptive programs by defining an induced sequential decision process to which standard RL is applied. Here we show that the success of this approach is highly sensitive to the specific program structure, where even seemingly minor program transformations can lead to failure. This sensitivity makes it extremely difficult for a non-RL-expert to write effective adaptive programs. In this paper, we study a more robust learning approach, where the key idea is to leverage information about program structure in order to define a more informative decision process and to improve the SARSA(λ) RL algorithm. Our empirical results show significant benefits for this approach.**

I. Introduction

Deterministic programming languages are not well suited for problems where a programmer has significant uncertainty about what the program should do at certain points. To address this, we study *adaptation-based programming* (ABP) where *adaptive programs* allow for specific decisions to be left open. Instead of specifying those decisions the programmer provides a "reward" signal. The idea then is for the program to automatically learn to make choices at the open decision points in order to maximize reward.

As detailed in Section VII, prior work has studied reinforcement learning (RL) for optimizing adaptive programs. Here we identify a shortcoming of this prior work, which is a significant obstacle to allowing non-RL-experts to benefit from ABP. Mainly, the success of prior RL approaches depends on subtle details of an adaptive program's structure, which is difficult to predict for non-RL-experts.

Motivating Example. Consider the two very simple adaptive programs P1 and P2 in Figure 1 written using our ABP Java library (see Section II). The objects `A` and `B` are *adaptives*, and are used to encode the programmer's uncertainty. Program P1 is a conditional structure with rewards at the leaves. Program P2 is a trivial transformation of P1 and to a typical programmer P1 and P2 appear functionally equivalent. However, as will be detailed in Section IV, these programs induce very different learning problems, apparent when prior RL approaches easily solve P1, but can fail for P2. Such sensitivity to trivial program changes is likely to be counter-intuitive and frustrating for a typical programmer.

```
if (test()) {
  if (A.suggest()) {
    reward(1);
  } else {
     reward(2);
  }
} else {
  if (B.suggest()) {
    reward(2);
  } else {
    reward(1);
  }
}
```
Program P1

```
a = A.suggest();
b = B.suggest();

if (test()) {
  if (a) {
    reward(1);
  } else {
     reward(2);
  }
 } else {
  if (b) {
    reward(2);
  } else {
    reward(1);
  }
}
```
Program P2

```
for (i=1; i<N; i++) {
  c = randomContext();
  m = move.suggest(c);
  reward(payoff(c,m));
}
```
Program P3

Figure 1. Illustrative adaptive programs. `test`, `randomContext`, and `payoff` are non-adaptive methods. `reward` and `suggest` are part of our ABP library (Section II).

As detailed in Sections IV and V, the fundamental problem with prior work is that they ignore much information about the structure of the program being optimized. Our primary contribution is to develop a general learning approach that takes program structure into account. To our knowledge this is the first work that attempts to exploit such program structure for faster and more robust learning. We show empirically that these ideas lead to significant improvement in learning on a set of adaptive Java programs.

II. Adaptation-Based Programming

We review our Java ABP library and the learning problem.

ABP in Java. The key object in our ABP library is an *adaptive*, which is the construct for specifying uncertainty. An adaptive has a *context type* and *action type* and can be created anywhere in a Java program where it can be viewed and used as a function from the context type to the action type, which changes over time via learning. This function is called via an adaptive's `suggest` method and we refer to any call to `suggest` as a *choice point*. In P1 and P2 of Figure 1, the action type of adaptives `A` and `B` is boolean, and the context is null. As another example, an adaptive program to control a grid world agent might include an adaptive `move` with action type $\{\texttt{N}, \texttt{E}, \texttt{S}, \texttt{W}\}$ for the desired movement direction and context type `GridCell`, enumerating the grid locations. The call `move.suggest(gridCell)` would return a movement direction for `gridCell`.

The second key ABP construct is the `reward` method,

which has a numeric argument and can be called anywhere in a Java program. In P1 and P2 the `reward` calls give the desirability of each leaf of the conditional. The goal of learning is to select actions at choice points to maximize the reward during program runs. In a grid world, the reward function might assert a small negative reward for each step and a positive reward for reaching a goal.

ABP Learning Problem. A *choice function* for an adaptive A, is a function from A's context type to action type. A *policy* π for adaptive program P gives a choice function for each adaptive in P. The execution of P on input x with respect to π is an execution of P where calls to an adaptive A's `suggest` method are serviced by A's choice function in π. Each such execution yields a deterministic sequence of `reward` calls and we denote the sum of rewards by $R(P, \pi, x)$. In a grid world, an input x might be the initial grid position and $R(P, \pi, x)$ might give the number of steps to the goal when following π.

Let D be a distribution over possible inputs x. The learning goal is to find a policy π that maximizes the expected value of $R(P, \pi, x)$ where x is distributed according to D.[1] Typically, finding the optimal π analytically is intractable. Thus, the ABP framework attempts to "learn" a good, or optimal, π based on experience gathered through repeated executions of the program on inputs drawn from D.

III. Basic Program Analysis for ABP

Our proposed learning approach (Sections IV and V) is based on leveraging the structure of adaptive programs and thus assume the ability to compute the following two properties of an adaptive program during its execution. v_ℓ denotes the value suggested by a `suggest` call at program location ℓ.

(1) $\bar{P}(\ell, \ell')$ says that v_ℓ is definitely irrelevant for reaching the location ℓ', that is, ℓ' would have been reached no matter what the value of v_ℓ is. In this case we say that ℓ' is *path-independent* of the choice point ℓ.

(2) $E(\ell, \ell')$ says that v_ℓ is potentially relevant for the computation of any value in a statement at position ℓ'. We say that ℓ' is *value-dependent* on ℓ.

To understanding our learning algorithm, it is sufficient to assume an oracle for these properties. We do not require that the oracle be complete for these properties, which are formally undecidable. Rather, our learning approach only requires that the oracle be correct whenever it asserts that one of the properties is true—the oracle can be incorrect when asserting false. The design goal of our approach is for a more complete oracle to lead to faster learning.

For lack of space, we only briefly outline a relatively simple approach for implementing the oracle. The details are not important for understanding our proposed learning approach and can be safely skipped. The oracle can be implemented as a pre-processor that instruments an adaptive program with code that generates information for evaluating the above predicates. We will use the notation $V(\ell)$ for the set of variables used in the statement at position ℓ. The key to implementing the predicates is to track the data flow of suggested values as well as their use in control-flow statements, which can be done via standard data-flow analysis techniques [1]. This gives for each program block B and each choice point ℓ the set $V_B(\ell)$ of variables visible in B whose values are potentially derived from v_ℓ.

Second, we instrument the program by inserting commands after each call to `suggest` and before each branching statement to report suggested value generation and usage. Moreover, before each branching statement at a position ℓ' with $V(\ell') \cap V_B(\ell) \neq \varnothing$ (where B is the current block), we insert a command that, during the program run, will produce a *use* event for v_ℓ. Through this instrumentation it is easy to compute the predicates $\bar{P}(\ell, \ell')$ and $E(\ell, \ell')$. Our experimental results are based on instrumenting the adaptive programs by hand in the same way the above pre-processor would. A fully automated pre-processor is being developed, noting that it is a large but straightforward engineering task that is not relevant to the main contribution of this paper (using information about program structure).

IV. The Informed Decision Process

Prior work on learning for ABP follows two steps: 1) Define a decision process, which we will call the *standard decision process* for the adaptive program, 2) Apply standard RL to this decision process. Below we describe the first step, a general deficiency with it and our proposed improvement.

We first define the notion of a general *decision process*, which is defined over a set of *observations* and *actions*. We consider episodic decision processes, where each episode begins with an initial observation o_1, drawn from an initial observation distribution I. There is a decision point at each observation, where a controller must select one of the available actions. After the i'th action a_i, the decision process generates a numeric reward r_i and a new observation o_{i+1} according to a transition distribution $T(r_i, o_{i+1}|H_i, a_i)$, where $H_i = (o_1, a_1, r_1, o_2, a_2, r_2, \ldots, o_i)$ is the observation history. Note that in general the transition distribution can depend arbitrarily on the history. In the special case when this distribution only depends on the o_i and a_i, then the process is a Markov decision process (with states corresponding to observations). The goal of RL is to interact with a decision process in order to find an action-selection policy that maximizes the expected total episodic reward.

Standard Decision Process. Similar to prior ABP work [2], [3] we can define a decision process corresponding to an adaptive program P and a distribution D over its inputs. We will refer to this as the *standard decision process* to

[1] In order to make this a well defined objective we assume that all program executions for any π and x terminate in a finite number of steps so that $R(P, \pi, x)$ is always finite. If this is not the case then we can easily formulate the optimization problem in terms of discounted infinite reward.

reflect the fact that prior work on ABP define a similar process to which RL is applied. The observations of the standard process are the names of the adaptives in P and the process actions are the actions for those adaptives. Each episode of the process corresponds to an execution of P on an input drawn from D. Decision points correspond exactly to choice points during the execution, and at each decision point the observation generated is the name of the adaptive. It is easy to show that there is a well-defined but implicit initial observation distribution and transition function for the standard process, both of which will be unknown to the learning algorithm. Further, there is a one-to-one correspondence between policies for P and for the standard process and the expected total reward for a policy is the same for the adaptive program and associated process. Thus, we can directly apply RL algorithms to the process and arrive at a policy for the program.

Deficiencies. Consider now the standard process corresponding to P1 in Figure 1. Each run of the program will generate exactly one observation, either `A` or `B`. The reward after the observation is equal to the appropriate leaf reward, which depends on the selected action. It is easy to verify that this process is Markovian and accordingly it can be easily solved by nearly all RL algorithms. In contrast, P2 induces a standard process that is non-Markovian. To see this, notice that the observation sequence for each run of the program is always `A`, `B` with zero reward between those observations and a non-zero reward at the end depending on the actions. This process is non-Markovian since the final reward depends not just on `B` and the action selected, but also on the actions of `A` (in the case that the left branch of the top level IF statement is selected). Because of this, applying algorithms that strongly rely on the Markov property, such as Q-learning or SARSA(0) can fail quite badly as our experiments will show. Furthermore, the credit-assignment problem is more difficult for P2 than P1 since the rewards arrive further from the decision points responsible for them. This example shows how even the simplest of program transformations can result in an adaptive program of very different difficulty for RL.

The Informed Decision Process. The general problem highlighted by the above example is that for the standard process, the sequence of observed decision points and rewards is highly dependent on details of the program structure, and some sequences can be much more difficult to learn from than others. Here, we partially address the problem by introducing the informed decision process, which is similar to the standard decision process, but with the addition of *pseudo decision points*. The new decision points will occur during a program execution whenever an instruction "depends" on the action of a previous choice point. Intuitively this encodes information into the process about when, during a program execution, the action at a choice point is actually used which is not available in the standard process.

More formally, given an adaptive program P, the corresponding informed decision process has the same observation and action space as the standard process (names of adaptives and their actions respectively). Also like the standard process, the informed process has a decision point whenever, during the execution of P, a choice point involving an adaptive A is encountered, which generates an observation A and allows for the selection of one of A's actions. In addition, at each non-choice-point location ℓ' of P, the oracle is asked for each prior choice point location ℓ, whether ℓ' is value-dependent on ℓ. If the answer is yes, then a pseudo choice point is inserted into the informed decision process, which generates observation A, where A is the adaptive at ℓ. Further, the only action allowed at this pseudo decision point is the action a that was previously selected by A, meaning that the controller has no real choice and must select a. The only effect of adding the pseudo decision points is for the informed process to generate an observation-action pair (A, a) at times in P's execution where A's choice of a is potentially influential on the immediate future.

Properties. It is easy to show that there is a one-to-one correspondence between policies of the informed process and of the adaptive program and that policy values are preserved. Thus, solving the informed decision process is a justified proxy for solving the adaptive program. Consider again P2 in Figure 1 on an execution where `A` and `B` both select `true`. If the variable `test()` method evaluates to `true`, then the informed decision process will generate the following sequence of observations, actions, and rewards: `A, true, 0, B, true, 0, A, true, 1`, where the final observation `A` is a pseudo decision point, which was inserted into the process when it was detected that the condition in the IF statement was dependent on it via the variable `a`. The insertion of the pseudo decision point has intuitively made the sequence easier to learn from since the choice $(A, \texttt{true})$, which was ultimately responsible for the final reward, is now seen before this reward.

V. Informed SARSA(λ)

While the informed decision process will typically be better suited to standard RL algorithms, it will still often be non-Markovian and not capture all of the potentially useful information about program structure. Prior work has studied the problem of learning memoryless policies for non-Markovian processes [4], [5], [6], showing that when good memoryless policies exist, learning algorithms based on eligibility traces such as SARSA(λ) [7] can often find optimal or very good policies [6]. Thus, we take SARSA(λ) as a starting point and later introduce the informed SARSA(λ) (*iSARSA(λ)*) algorithm that leverages information about program structure not captured by the informed process.

SARSA(λ). The SARSA(λ) algorithm interacts with a decision process in order to learn a Q-function $Q(A, a)$.

While SARSA(λ) was originally developed for MDPs, it can be applied to non-Markovian processes, though certain guarantees are lost. The key idea of SARSA(λ) is to maintain an *eligibility trace* function $\eta_t(A, a)$ and to update the Q-function after each transition for each (A, a) according to their eligibilities. Intuitively, recently observed pairs are more eligible for update based on a new transition.

At the start of each program run we set $\eta(A, a) = 0$ for all pairs. The behavior of SARSA(λ) can now be described by what it does at each decision point and transition: Given a current observation A, SARSA(λ) first selects an action a according to an exploration policy, ϵ-greedy here. This action causes a transition, which produces a reward r and a new observation A' upon which it again uses the exploration policy to select an action a'. The algorithm next updates the eligibility trace and Q-values for all pairs as follows:

$$\eta(A, a) \leftarrow 1 \quad (1)$$
$$\eta(B, b) \leftarrow \lambda\eta(B, b), \quad \text{for all } (B, b) \neq (A, a) \quad (2)$$
$$\delta \leftarrow r + Q(A', a') - Q(A, a) \quad (3)$$
$$Q(B, b) \leftarrow Q(B, b) + \alpha \cdot \delta \cdot \eta(B, b), \quad \text{for all } (B, b) \quad (4)$$

where $0 \leq \lambda \leq 1$ is the eligibility parameter, and $0 < \alpha < 1$ is the learning rate. For larger values of λ, decisions are more eligible for update based on temporally distant rewards, with $\lambda = 1$ corresponding to learning from Monte-Carlo Q-value estimates. The selection of λ is largely empirical, but for non-Markovian processes, larger values are preferred.

Deficiencies. Consider the observation-action-reward sequence of the informed process for an execution of program P2 described at the end of Sec IV: `A, true, 0, B, true, 0, A, true,1`, where the final observation was a pseudo decision point inserted by the informed process. When the final reward of 1 is observed, the choice $(B, \texttt{true})$ will be eligible for update. However, a simple analysis of the program reveals that in this trace the decision of adaptive B had no influence on whether the reward was received or not. Thus, intuitively the decision involving B does not deserve credit for that reward, but this fact is missed by the generic SARSA(λ) algorithm.

A more severe example is program P3 in Figure 1, which contains a loop where each iteration corresponds to the complete play of a simple game. The action, selected by the adaptive `move` based on the current context influences the reward payoff. An execution of P3 iterates through the loop generating a sequence of choices made by the adaptive and rewards. A simple analysis reveals that the choice made at iteration i has no influence on the reward observed in future iterations and thus should not be given credit for those rewards. However, SARSA(λ) again does not capture this information and will update the choice made at i based on some combination of all future rewards, making the learning problem very difficult since the independence across iterations must be learned.

Exploiting this type of simple program analysis to improve credit assignment is what iSARSA(λ) is designed to do. The main idea is to reset the eligibility of such irrelevant choices to 0, making them ineligible for updates based on rewards they provably have no influence on.

iSARSA(λ). We now describe the working of our algorithm listed in Algorithm 1. iSARSA(λ) gets invoked at every observation-action pair (A', a') (henceforth simply 'choice') in the informed process. For each such (A', a'), let (A, a) denote the immediately preceding choice and r be the reward between (A, a) and (A', a'). Like SARSA(λ), we first set the eligibility of choice (A, a) to 1 and decay the eligibility of all prior choices by λ (Lines 2-5). Furthermore, iSARSA(λ) will exploit the program analysis and ask for every previous choice (B, b), whether it is irrelevant to getting to A' (Line 9). In this case $Q(B, b)$ is updated based on only the immediate reward r (rather than also on the estimate of $Q(A', a')$) and its eligibility is then reset to zero (Lines 10-11) so that it will not receive credit for further rewards. Otherwise if irrelevance is not detected for (B, b) the usual SARSA(λ) update is performed (Line 13).

Properties. The convergence of SARSA(λ) for arbitrary lambda is an open problem even for Markovian processes, though the algorithm is widely used for its well documented empirical benefits. However, for $\lambda = 1$, iSARSA(λ) when run in the informed decision process has a useful interpretation as a Monte Carlo algorithm. In particular, the Q-value of program choice (A, a) will be updated toward the sum of future program rewards, excluding those rewards for which (A, a) is provably irrelevant. This is an intuitively appealing property and as we will show can lead to very large empirical benefits. As a simple illustration, for program P3 it is easily verified that iSARSA(λ) will update the choice at iteration i based on only the reward observed at iteration i, rather than future iterations, as appropriate.

Algorithm 1 : **iSARSA(λ)**. All Q, η values are initially 0.

1: {Current program location ℓ' at choice (A', a'),
   Last choice (A, a) with reward r seen in between }
2: **for** each choice (B, b) **do**
3: $\quad \eta(B, b) = \eta(B, b) * \lambda$
4: **end for**
5: $\eta(A, a) = 1$
6: $\delta_e = r - Q(A, a)$ {terminal delta}
7: $\delta = r + Q(A', a') - Q(A, a)$ {regular delta}
8: **for** each previous choice (B, b) at location ℓ **do**
9: $\quad$ **if** ℓ' is path-independent of ℓ **then**
10: $\quad\quad Q(B, b) = Q(B, b) + \alpha\eta(B, b)\delta_e$
11: $\quad\quad \eta(B, b) = 0$
12: $\quad$ **else**
13: $\quad\quad Q(B, b) = Q(B, b) + \alpha\eta(B, b)\delta$
14: $\quad$ **end if**
15: **end for**

VI. Experiments

We experiment with adaptive programs that include a Yahtzee playing program, a sequence labeling program,

and 3 synthetic programs containing a variety of program structures. The programs and our ABP library are available upon request. Space permits only a brief description of each.

Yahtzee [8]. This is a complex, stochastic dice game played over 13 rounds or cycles where each round typically requires 3 high-level decisions. We wrote an adaptive program to learn to play Yahtzee over repeated games that includes 2 adaptives to make the 2 most difficult sub-decisions while other easily-made decisions were hard-coded. The program structure is such that certain choices made by the adaptives may get ignored depending on the game context making for a difficult credit assignment problem.

Sequence Labeling (SeqL). This is an important problem in machine learning, where the goal is to accurately label a sequence of input symbols by a sequence of output labels. The programmer will often have significant domain knowledge about proper labelings, and ABP provides a tool for encoding both, the knowledge and the uncertainty. We wrote an adaptive program that repeatedly cycles through a set of labeled training sequences, and for each sequence makes a left to right pass assigning labels. The reward statements reflect whether the selected labels are equal to the true labels. At each sequence position the choice of label is made using a conditional structure that contains a combination of the programmer's knowledge and a single adaptive, which ends up making the credit assignment problem quite difficult for standard RL. We generated a synthetic data set based on a Hidden Markov Model (HMM) involving 10 input and 10 output symbols. The training set contains 500 sequences, while the test set has 200 sequences. The test set is hidden during training and used to evaluate the learned program.

Synthetic Programs. These are composed of adaptives and `reward` statements scattered across branches and loops. All adaptives have binary actions and possibly non-null context types. *Program* S1 is based on P3 from Figure 1, where the adaptive has 10 possible contexts. This program captures the common adaptive-programming pattern of repeated runs which can cause failure when using standard RL. *Program S2* contains an IF-THEN-ELSE tree followed by another tree. The reward generated in the first tree depends only on the choices made in there and the reward seen in the lower tree is independent of the first. This type of program might be written when the input is used to solve two independent sub-tasks, each with its own reward. *Program S3* contains a sequence of independent trees. The upper tree has a loop with a tree inside it. The lower tree has a similar structure but the reward depends on the context stochastically.

Results. We evaluate three algorithms: 1) SARSA(λ) applied to the standard decision process, which is representative of prior work, 2) SARSA(λ) applied to the informed decision process, and 3) iSARSA(λ) applied to the informed decision process. We used a constant learning rate of $\alpha = 0.01$ and ϵ-greedy exploration with $\epsilon = 0.1$. For each synthetic adaptive program we conducted 20 learning runs of each learning algorithm and averaged the resulting learning curves. Each learning run consisted of repeated program executions during which the learning algorithm adapts the policy. After every 10 program executions, learning was turned off and the current policy was evaluated by averaging the total reward of another 10 program executions, which are the values we report. For Yahtzee we evaluate every 1000 games by averaging the score of 100 games and average over 20 learning runs. For the SeqL program, we use 1000 passes over the training set, evaluating performance (i.e. number of correct labels) on the *training* set after every 10 passes. At the end of the program run, we evaluate performance on the test set. This is done over 5 learning runs, each one using different datasets generated by the HMM. Figures 2(a-e) show the averaged learning curves when using $\lambda = 0.75$ for all algorithms on all benchmarks. Figure 2(f) shows the average number of correct labels per sequence over the test set for SeqL.

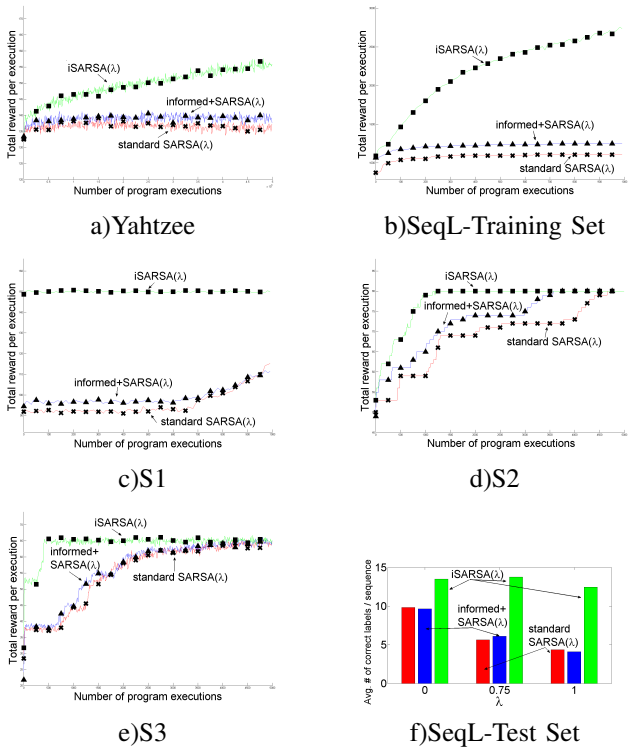

Figure 2. (a-e) are performance graphs of programs Yahtzee, SeqL (training set), S1,S2,S3 for $\lambda = 0.75$. (f) SeqL (test set) for varying λ and algorithms

Benefit of Informed Decision Process. For all example programs, we see a small improvement in learning efficiency for SARSA(λ) when learning in the informed process versus the standard process. This shows that the additional information in the informed process is useful, however, the relatively small improvement indicates that the credit assignment problem in the informed process is still difficult.

Benefit of iSARSA(λ). Now compare learning in the informed process with SARSA(λ) versus iSARSA(λ). For

each of our programs there is a substantial boost in learning efficiency when using iSARSA(λ). Furthermore, for the Yahtzee, SeqL and S1(loop) programs, it appears that iSARSA(λ) leads to substantially better steady state performance in the informed process than SARSA(λ). All of these programs contain loops with independent iterations, which makes credit assignment difficult with standard eligibility traces, but much easier with the informed traces of iSARSA(λ). These results give strong evidence that iSARSA(λ) is able to effectively leverage information about the program structure for faster learning. Particularly impressive is its performance on the test set in SeqL, in Figure 2(f) where on average, it predicts 40% more labels correctly for $\lambda = 0.75$.

Varying λ. We ran similar experiments for $\lambda = 0$ (pure TD-learning) and $\lambda = 1$ (pure Monte Carlo). The general observation of these experiments is that the relative performance of the methods is similar to $\lambda = 0.75$. Further iSARSA(λ) is much less sensitive to the value of λ than SARSA(λ) in the informed process, which was less sensitive to λ than SARSA(λ) in the standard process. This shows that the additional information used by iSARSA and contained in the informed process have a large benefit in terms of addressing the credit assignment problem, which is traditionally dictated by the precise value of λ.

Overall our results give strong evidence that: 1) Learning in the informed process is beneficial compared to learning in the standard process, and 2) iSARSA(λ) is able to leverage program structure to dramatically improve learning compared to pure SARSA(λ) in the informed process.

VII. Related Work

There have been several prior and similar ABP efforts, sometimes under the name partial programming [2], [3], [9], [10]. Most notably ALISP [9] extends LISP with choice structures, which are similar to adaptives in our library. There is an important semantic distinction between our work and prior work such as ALISP. The semantics of ALISP are tied to an interface to an external MDP (e.g. which produces reward). This coupling to MDP theory is a hurdle for non-RL expert programmers. Rather, our ABP semantics are not tied to the notion of an external MDP, but defined completely in terms of a program and a distribution over its inputs. This makes our library immediately applicable in any context that Java programs might be written.

Prior learning algorithms for partial programming make very limited use of the program structure, which is the key contribution of our work. The only exception is work on ALISP that leverages program structure in order to decompose the value function according to subroutines [9]. While this is a powerful mechanism it is orthogonal to the type of structure exploited in our work. In particular, the program structure that we exploit can be present within a single sub-routine and does not require a sub-routine decomposition. Combining our work with decomposition based on sub-routines is an interesting direction.

ABP should not be confused with work on RL frameworks/libraries (e.g. RL-Glue[11]). Such libraries are intended for *RL experts* to develop, test and evaluate RL algorithms and provide a language-independent, standardized test harness for RL experiments. Rather, ABP is intended to be a more thorough integration of learning into a programming language, allowing arbitrary programmers to benefit from learning algorithms by simply including adaptive constructs into their programs in a natural way.

VIII. Summary

We have highlighted the fact that subtle differences in adaptive programs can lead to large differences in the performance of standard RL. For non-RL experts, this seriously impedes their ability to use the ABP paradigm. To address this issue we showed how to leverage program structure to both define a new induced decision process and inform the SARSA(λ) RL algorithm. The results show that this approach leads to good performance on complicated adaptive programs, both real and synthetic.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," in *NIPS*, 1998.

[3] D. Andre and S. Russell, "Programmable reinforcement learning agents," in *NIPS*, 2001.

[4] S. Singh, T. Jaakkola, and M. Jordan, "Learning without state-estimation in partially observable Markovian decision processes," in *ICML*, 1994.

[5] M. Pendrith and M. McGarity, "An analysis of direct reinforcement learning in non-Markovian domains," in *International Conference on Machine Learning*, 1998.

[6] J. Loch and S. Singh, "Using eligibility traces to find the best memoryless policy in partially observable Markov decision processes," in *ICML*, 1998.

[7] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2000.

[8] J. Glenn, "An optimal strategy for yahtzee," Loyola College, Technical Report CS-TR-0002, 2006.

[9] D. Andre and S. Russell, "State abstraction for programmable reinforcement learning agents," in *AAAI*, 2002.

[10] C. Simpkins, S. Bhat, M. Mateas, and C. Isbell, "Toward adaptive programming: Integrating reinforcement learning into a programming language," in *OOPSLA*, 2008.

[11] B. Tanner and A. White, "Rl-glue: Language-independent software for reinforcement-learning experiments," *J. Mach. Learn. Res.*, vol. 10, pp. 2133–2136, 2009.