Faster Program Adaptation Through Reward Attribution Inference

Tim Bauer Martin Erwig Alan Fern Jervis Pinto
Oregon State University
Corvallis, Oregon 97331 USA
{bauertim,erwig,afern,pinto}@eecs.oregonstate.edu

ABSTRACT

In the adaptation-based programming (ABP) paradigm, programs may contain variable parts (function calls, parameter values, etc.) that can be take a number of different values. Programs also contain reward statements with which a programmer can provide feedback about how well a program is performing with respect to achieving its goals (for example, achieving a high score on some scale). By repeatedly running the program, a machine learning component will, guided by the rewards, gradually adjust the automatic choices made in the variable program parts so that they converge toward an optimal strategy.

ABP is a method for semi-automatic program generation in which the choices and rewards offered by programmers allow standard machine-learning techniques to explore a design space defined by the programmer to find an optimal instance of a program template. ABP effectively provides a DSL that allows non-machine-learning experts to exploit machine learning to generate self-optimizing programs.

Unfortunately, in many cases the placement and structuring of choices and rewards can have a detrimental effect on how an optimal solution to a program-generation problem can be found. To address this problem, we have developed a dataflow analysis that computes influence tracks of choices and rewards. This information can be exploited by an augmented machine-learning technique to ignore misleading rewards and to generally attribute rewards better to the choices that have actually influenced them. Moreover, this technique allows us to detect errors in the adaptive program that might arise out of program maintenance. Our evaluation shows that the dataflow analysis can lead to improvements in performance.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming—*automatic analysis of algorithms, program transformation*; D.3.4 [**Programming Languages**]: Processors—*optimization, preprocessors*

General Terms

Languages

Keywords

Partial Programming, Program Adaptation, Reinforcement Learning

1. INTRODUCTION

Programs that implement deterministic algorithms require an exact and detailed specification of all steps and conditions. However, many algorithms are easier to specify if we are permitted to insert a modicum of uncertainty in the program that can be figured out later (for example, through learning from repeated program executions). Furthermore, this flexibility of leaving some program parts under-specified can benefit problems where the best algorithm depends on characteristics of the algorithm's input distribution and expected use, which are often unclear before the program is actually run. For example, many algorithms contain a cutoff threshold to decide when to switch strategies, such as merge-sort implementations that switch to insertion sort on sufficiently small sub-lists. Exactly what is "sufficiently small" is such an uncertainty.

In previous work [6, 7] we have introduced a programming paradigm to support a form of partial programming [5] called *adaptation-based programming* (ABP) whereby the programmer can write non-deterministic self-optimizing program templates (adaptive programs), which can be instantiated into a sequence of improving programs. In ABP programmers express uncertainty via *choices* and indicate program performance via feedback signals, called *rewards*. The adaptive program utilizes reinforcement learning (RL) [14] to self-optimize itself over multiple runs. Any choice in an adaptive program can be passed information about the program state as a parameter, which is remembered by the RL agent during execution to later associate rewards with choices made in specific program situations. For each choice the RL agent will select an *adaptive value* based on the current information about program state and the performance of the different choices learned from previous program executions. As choices are made and rewards are encountered, the RL agent learns and adapts its future choices in such a way as to maximize its total expected reward.

An additional benefit of the ABP programming model (or any learning model) is that the adaptive program is optimized over the specific inputs it sees. For example, if an adaptive program was learning a cutoff point to switch from merge sort to insertion sort and then operated in an environment with nearly sorted lists, it would learn a cutoff favoring insertion sort for longer since that algorithm operates efficiently in such cases.

Another goal of ABP is to offer an easy-to-use programming model to users with little or no experience in RL. State-of-the-art RL tends to be quite complicated and requires considerable expertise to be used effectively. For instance, programs using RL must be structured carefully to respect assumptions that various RL algorithms make. ABP programs, on the other hand, consist of just choices and rewards, which are then mapped into an RL problem that can then be solved using standard techniques.

The more relaxed program structure makes ABP easier to use for the non-RL-expert (which we simply call non-expert from now on). However, this freedom comes at a price since not all program structures are equally conducive to effective learning, and a seemingly innocent adaptive programs can behave quite poorly in terms of the time it takes to learn. Since the unwary non-expert does not know the ins and outs of RL algorithms, it is the duty of ABP to abstract away as many of those details as possible so they may focus fully on their problem.

A very common class of errors that a non-expert can make involves mis-attributing rewards to the wrong choices. RL algorithms perform much better if rewards are given nearer to the choices that affect them—ideally right after them. Since the non-expert user is probably unaware of this requirement, a seemingly innocent program restructuring can cause rewards to be associated with the wrong choices, leading to a degradation of learning performance.

We refer to the problem of associating rewards with the choices that influence them as *reward attribution*. RL has worked on this problem under the name credit assignment [13]. However, these approaches do not consider program structure as those algorithms do not have the user's program available in a form they can use. In contrast, the ABP paradigm provides the novel capability to analyze the user's program and exploit facts gathered through data-flow analysis to improve the RL algorithm's behavior.

One direct way of achieving accurate reward attribution for ABP would be to explicitly represent choice-reward dependencies. For example, whenever a user creates a choice, they could be asked to provide both the value of the choice and a token to identify that choice explicitly. Then when giving a reward later, the user could be also required to explicitly supply a list of the choices that influenced it. However, burdening the user with this requirement suffers several major problems.

- Tracking the choices made incurs an undue cognitive burden on the user. The goal of allowing relaxed choice and reward placement is to allow the user full flexibility in the structure of their partial program.

- Users could get the annotations wrong; they might mistakenly annotate a choice as being influential for a reward when it is not, or conversely miss an important choice influence on a reward. Even when planned carefully, data-flow analysis can be quite tricky, and asking users to do this by hand can easily lead to mistakes. Furthermore, when code gets moved around during maintenance, failure to maintain the correspondingly changing dependences would invalidate the information. The requirement to update the choice-dependency information "by hand" severely impacts program maintainability.

In this work we illustrate how we can compute a data-flow relation between choices and the rewards that they might influence and thus automate the process of reward attribution. We then show several mistakes a programmer could easily make and how the relation can be used to automatically detect and fix these problems.

Our main contributions are:

- A formal description for a static data-flow analysis that relates choices to rewards that they influence (the *R* relation).

- A learning algorithm that successfully exploits the relation.

		Statements	
s	::=	`v := e`	*assignment*
	|	`if e then s else s`	*alternation*
	|	`while e do s`	*iteration*
	|	`s; s`	*sequence*
	|	`l:a`	*adaptivity*
		Adaptive statements	
a	::=	`reward(e)`	*reward*
	|	`choose(v,e,ē)`	*choice*
$\bar{e}$	::=	`[e,...,e]`	*alternatives*

Figure 1: Syntax for a Simple Language

- A demonstration of the utility of the algorithm by applying it to various ABP programs and showing the improvement.

The rest of the paper is structured as follows. In Section 2 we give a brief overview of ABP. We provide a formal description of the data-flow inference algorithm in Section 3 and illustrate it with several small examples. Next, we use this data-flow information in Section 4 to further improve programs containing loops. In Section 5 we discuss the learning algorithm used by ABP and the modifications we make in order for it to benefit from the reward attribution relation data. In Section 6 we perform evaluations to show the improvement our modified algorithm provides on various programs. Section 7 discusses related work and Section 8 concludes.

2. ADAPTATION-BASED PROGRAMMING

Previously we presented the idea of ABP in higher-level languages such as Java and Haskell [6, 7]. However, to illustrate this work, we present ABP within the context of a small imperative language with very simple semantics.

As shown in the figure 1, the syntax provides basic language constructs for a typical imperative language, extended by two adaptive statements for describing choice and providing rewards. The syntax above implicitly refers to expressions as *e*, variables as *v*, and labels as *l*. Labels and variables are assumed to be unique throughout the program (for example, there is no name shadowing for variables). Adaptive operations are prefixed with labels to simplify the formal description of the choice-reward (*R*) and choice-choice (*C*) influence relations with inference rules. (In fact, those labels are automatically added by our implementation.)

The choice construct `choose` is passed a variable *v*, an optional context expression *e*, and a list of alternatives to choose from (given in the form of expressions). The context expression indicates information that this choice depends on. In some cases this context is irrelevant and will be omitted defaulting to some unique fixed value. One of the alternatives will be chosen and bound to the variable *v*.

The `reward` construct takes a numeric expression for a reward value. This value is passed to the learning algorithm to indicate positive or negative program performance to the ABP learning algorithm.

We start by outlining the basic ideas of ABP at a high level with the following simple example adaptive program.

```
r := 0
c1:choose(x,[0,1])
c2:choose(y,[0,1])
if x then
    r1:reward(1)
  else
    r := y + 2
r2:reward(r)
```

Intuitively, we can see that the reward statement labeled `r1` depends only on the adaptive value chosen at `c1` (stored in `x`). However, the reward at `r2` depends on both `c1` and `c2`, which can be seen as follows. The value of the reward passed in `r1` depends on the value of `x`, hence it depends on `c1`. The influence from `c2` arises since the value of `r` at the reward statement `r2` may be defined in terms of `y`, which is set by that choice.

This discussion also alludes to the fact that our view of this relation is a static and conservative one. Because of the way the RL algorithms work, we must never accidentally disassociate a reward with a choice, we can however include false positives. Hence, if a variable may have several definitions at any given point, we assume it is influenced by all possible definitions.

Finally, the example above also illustrates what we mean by "program template". We can actually view the adaptive program as four possible programs to which it can be instantiated by (independently) choosing the values `0` and `1` for each of the variable `x` and `y`. Over repeated runs ABP will converge on those choices that generate a program that produces the highest reward. In the example, this will result in the following program (choosing `0` for `x` and `1` for `y`).

```
r := 0
x := 0
y := 1
if x then
    r1:reward(1)
  else
    r := y + 2
r2:reward(r)
```

Of course, the resulting program can be simplified further by removing the reward statements that are no longer needed and applying other (algebraic) program transformations.

3. REWARD ATTRIBUTION ALGORITHM

We now formally detail the reward attribution algorithm with a set of inference rules.

The reward attribution relationship computation has the following general type.

$$A, I, R \vdash s \Rightarrow I, R$$

The set $A \in 2^L$ is the set of *active influences*, and it consists of choice labels (L is the set of all labels), that is, those labeling `choose` locations. Throughout the algorithm A represents the set of choices that might influence the current execution path. This set is typically modified by control flow constructs such as `if` and `while` statements since these statements dictate control flow. Variable binding operations such as assignment and choice all use A to determine which choices influenced the new definition.

The second set $I \subseteq V \times L$ is the *influence map* (V is the set of all variable names), and relates variables to choices that influence their value. If a variable or value is currently influenced by some choice, we say it is *adaptive* (an adaptive value or an adaptive variable).

The set R is the algorithm's result; it associates choice locations with reward locations that they influence. (The I also appears on the right-hand side of the judgement as a result value since I consists of state information that must be threaded through the algorithm.)

We now proceed through the inference rules that define the inference algorithm, given formally in Figure 2. We use the following auxiliary functions and notation. The projection of the influence map I with respect to a set of variables X, written as is defined as follows.

$$I[X] = \{l \mid (v,l) \in I \wedge v \in X\}.$$

REWARD
$$A, I, R \vdash r\!:\!\texttt{reward}(e) \Rightarrow I, A_e \times \{r\}$$

CHOICE
$$A, I, R \vdash c\!:\!\texttt{choose}(v,e,\bar{e}) \Rightarrow I[v := A_{e,\bar{e}} \cup \{c\}], R$$

ASSIGNMENT
$$A, I, R \vdash v \;\texttt{:=}\; e \Rightarrow I[v := A_e], R$$

SEQ
$$\frac{A, I, R \vdash s_1 \Rightarrow I_1, R_1 \qquad A, I_1, R_1 \vdash s_2 \Rightarrow I_2, R_2}{A, I, R \vdash s_1\,\texttt{;}\,s_2 \Rightarrow I_2, R_2}$$

IF
$$\frac{A_e, I, R \vdash s_t \Rightarrow I_t, R_t \qquad A_e, I, R \vdash s_e \Rightarrow I_e, R_e}{A, I, R \vdash \texttt{if}\ e\ \texttt{then}\ s_t\ \texttt{else}\ s_e \Rightarrow I_t \cup I_e, R_t \cup R_e}$$

WHILE
$$\frac{A_e, I, R \vdash s \Rightarrow I, R'}{A, I, R \vdash \texttt{while}\ e\ \texttt{do}\ s \Rightarrow I, R \cup R'}$$

Figure 2: Inference Rules for Reward Attribution

We write $I[v := C]$ for updating the influence map I for the variable v with the set C. Formally, we have:

$$I[v := C] = \{(x,l) \in I \mid x \neq v\} \cup \{(v,c) \mid c \in C\}$$

Finally, we write $A_{e_1,\ldots,e_n}$ for the set $A \cup I[vars(e_1,\ldots,e_n)]$ where $vars(e_1,\ldots,e_n)$ returns the set of all variables contained in the expressions $e_1,\ldots,e_n$.

The REWARD rule states that a `reward` statement relates the (point of) reward r with all the current active influences as given by the set A. Furthermore, the numeric reward expression e passed to `reward` also might be influenced by choices and must also be considered to influence this statement. To this end, the function *vars* computes all the variables contained in e, and the projection operator on I selects the choices that influence those variables.

The CHOICE rule describes how to handle a choice. Nothing in this statement directly affects control flow so A remains unchanged. However, the rule does modify the influence mapping I since v is being (re)defined here. The new definition of v depends on three pieces: the choice c itself, the current set of choices influencing execution flow (A), and any influences on variables in the argument expressions; that is, the context expression e and argument expressions $\bar{e}$.

The ASSIGNMENT rule is very similar to the CHOICE rule in that we are redefining a variable. The main difference is that since this is not an adaptive assignment there is no choice label c to associate with the new definition of v. Consider the example below.

```
v := v + 2*x + y
```

Assume the following initial definitions of A and I.

$$A = \{\texttt{c}_1\} \text{ and } I = \{(\texttt{v},\texttt{c}_2),(\texttt{x},\texttt{c}_3),(\texttt{y},\texttt{c}_3),(\texttt{y},\texttt{c}_4)\}$$

Then I changes as follows. First, we subtract old definitions of `v` since `v` is being redefined thus removing $\{(\texttt{v},\texttt{c}_2)\}$. Next, we take the union of all influences of the expression on the right-hand side, that is, all influences on the variables within the expression. In this example this ends up being {`v`,`x`,`y`}. These are the choices $\{\texttt{c}_2,\texttt{c}_3,\texttt{c}_4\}$. Finally, we include A, just $\{\texttt{c}_1\}$ in this case. We cross these choices with $\{v\}$ to get the new definition for I given below.

$$I = \{(\texttt{v},\texttt{c}_2),(\texttt{v},\texttt{c}_3),(\texttt{v},\texttt{c}_4),(\texttt{x},\texttt{c}_3),(\texttt{y},\texttt{c}_3),(\texttt{y},\texttt{c}_4)\}$$

Sequential composition of statements, described by SEQ, threads I and R through both statements, nothing terribly special or exciting happens with this rule.

The IF rule extends the set of active influences by whatever influences are in the guard condition e (yielding the set A_e) for both the then and else statements s_t and s_e. The IF rule merges the results R_t with R_e and I_t with I_e by taking their union.[1]

Similar to IF, the WHILE rule extends A based on adaptive variables in the guard expression e. However, this construct also considers the output of the loop body statement s since variables used in e may be redefined within that body. The I in this rule must satisfy both the invariant and premise of the rule. This fixed point can be computed via a *forward-maybe* iterative data-flow analysis [3].

We now illustrate some of these rules through a few examples.

```
r := 0
c1:choose(x,[0,1])
if x then
     c2:choose(y,[0,1])
     r1:reward(1)
     if y then
          m := 2
r2:reward(m)
```

Consider the above example consisting of two choices (both without explicit context values). Initially A, I, and R are all empty. The ASSIGNMENT rule assigning 0 to `r` has no effect since nothing is influenced by any choices here. However, after the first choice, the CHOICE rule adds the pair (`x`,`c1`) to the I relation to indicate that `x` is currently being influenced by `c1`.

As we descend into the `if` statement the IF rule extends A by `c1` since that choice influences its guard expression `x`. Next, the second `choose` at `c2` associates itself to the variable it is defining, adding (`y`,`c2`) to the I relation. In addition this rule adds (`y`,`c1`) to I since this definition of `y` depends on `x`, which in turn depends on `c1`. Hence, I ends up as {(`x`,`c1`),(`y`,`c2`),(`y`,`c1`)} within this conditional statement.

We encounter the reward statement on line five, and (`c1`,`r1`) is added to R. The if statement following that reward extends A further adding its guard expression `y`'s influences {`c1`,`c2`}. The assignment within that nested if-statement `m := 2` marks the new definition of `m` as influenced by all the choices in A, which adds both (`m`,`c1`) and (`m`,`c1`) to I.

Finally, the last `reward` again extends R by adding all the influences on its argument `m`, which are (`c1`,`r2`) and (`c2`,`r2`) yielding the final result given below.

$$R = \{(\texttt{c1},\texttt{r1}),(\texttt{c1},\texttt{r2}),(\texttt{c2},\texttt{r2})\}$$

As a second example we consider a program with a small loop.

```
i := 0
while i < 8 do
   i := i + 1
   c:choose(x,[0,1])
   if x then
        i := i + 1
   r:reward(1)
```

This program executes a loop no more than 8 times collecting a reward of 1 each iteration. However, within the loop we adaptively decide whether to speed it up by advancing the loop counter an extra iteration. Hence, the loop counter's behavior is adaptive.

[1]If we intersected sets here, we would get the *definite influences* instead of the *potential influences* as we currently have.

More precisely, the WHILE rule is applied to empty A and R sets. However, the I relation will contain (`i`,`c`) at the top level; I is a fixed point chosen to satisfy the loop body as well as the I input to the loop, and within this loop body I must reflect the influence of `c` on `i`.

As the algorithm descends into the loop body, `x` is bound to an adaptive value at choice `c`, and the CHOICE rule marks `x` as being influenced by that choice (I is extended to reflect this). Next, we descend into the `if` statement and apply the IF rule, causing A to be extended to include `c` since that choice influences the guard expression `x`. Consequently, when we reach the following assignment statement that increments `i`, the A set contains the choice `c` as an influence, which makes the new definition of `i` adaptive (influenced by `c`). When we reach the `reward` on the next line I consists of {(`x`,`c`),(`i`,`c`)}, and as mentioned already, A contains choice `c` within the loop body. Hence, the REWARD rule adds the influence pair (`c`,`r`) to R to reflect the influence of the choice on the reward.

4. CHOICE INVALIDATION

A related problem encountered when mapping adaptive programs to learning problems is determining what constitutes a full test episode. The RL methods used by ABP all employ the concept of an episode, which roughly corresponds to a single program run, round, or match. Rewards and choices are independent across an episode boundary. Indeed, if one wins a chess match, they should not credit moves made in prior matches. Moreover, the notion of an episode can be applied at choice granularity rather than program run granularity.

We illustrate this with the example below.

```
i := 0
while i < 10 do
   i := i + 1
   c:choose(a,[0,1])
   if a == 1 then
        r:reward(1)
```

This program makes a choice within a loop and, if correct, receives a reward.

Without the loop the problem is very simple. However, with the loop it is much harder to learn. Suppose choice `c` chooses the correct value for the first iteration and an incorrect value the second iteration. The learning algorithm requires we collect these rewards until the end of the program run and only then apply them. Hence, it cannot tell if the first or second value it selected at choice `c` improved the reward more.

An RL expert could see that each loop execution is independent, choices made in one iteration do not influence rewards in any future iteration. Hence, what constitutes a "program run" should be a single loop execution. Now, we could provide some means to allow the user to explicitly indicate such points in their program, but this suffers from the same disadvantages as forcing users to explicitly indicate reward-choice influence. That is, novices and experts alike might miscompute this boundary by failing to see a data dependency that exists, or by thinking one exists when it does not. Maintaining this information as the program is modified might also pose a challenge.

However, what an RL expert does to determine if loop iterations are independent episodes is exactly the same as a data-flow analysis. The expert asks, "Is the effect of a choice made confined to a single loop iteration or can it affect a future reward?"

Using the sets defined in Figure 2 we can determine the set of choices *definitely* invalidated at each binding construct (a `choice` or assignment operation in our example language). For any statement that (re)defines v we compute the *invalidated* set as follows.

$$invalidated = I[v] - live(I[v := \varnothing] \cup A)$$

$I[v]$ is the set of choices that any definition of v depends on, that is, the initial candidate set of choices potentially being invalidated by this rebinding of v. From this we subtract those kept alive either by other variables ($I[v := \varnothing]$) or those in the active influence set (A). Finally, the *live* function tightens up the estimate by removing any choices that are not used in reachable adaptive rewards (another variable might be influenced by a choice that will never be used again). This function can be implemented with similar data-flow analysis techniques as described in [3].

Because R is a conservative estimate (all the choices that *may* influence a particular statement) our invalidation estimate is also conservative. Specifically, there may be cases were we fail to detect a choice's invalidation at a binding construct (a false negative), however, if a choice is reported as invalid, we are certain it cannot influence future rewards.

We illustrate choice invalidation with the example below.

```
c1:choose(x,[0,1])
c2:choose(y,[0,1])
z := x + y
...
z := 0
r2:reward(x)
```

Consider the choices killed at assignment statement `z := 0`. Here, $A = \varnothing$ and $I = \{$`(x,c1),(y,c2),(z,c1),(z,c2)`$\}$. $I[$`z`$] = \{$`c1,c2`$\}$ (`z` depends on both choices) $I[$`y:=`$\varnothing] = \{c1, c2\}$ (`x` depends on `c1` and `y` on `c2`, but is never used again) giving the result below.

$$\{\texttt{c1,c2}\} - live(\{\texttt{c1,c2}\} \cup \varnothing) = \{\texttt{c1,c2}\} - \{\texttt{c1}\} = \{\texttt{c2}\}$$

With this information we can very accurately inform the learning algorithm when a choice can no longer affect any rewards and bar that choice from seeing later rewards that do not affect it.

5. A MODIFIED LEARNING ALGORITHM

In this section we discuss how the learning algorithm exploits the R relation to make better decisions.

Our ABP learner uses a *Monte-Carlo* algorithm with ε-greedy exploration [14]. We refer to this initial algorithm as *MC* and describe it here before extending it to use the R relation.

When confronted with a choice for a given context (that is, part of the program state), the algorithm randomly picks between exploiting the best known action (with high probability) for that context and exploring an alternate action (with low probability). Whenever a reward has been seen, it is awarded to all previous choices made. After the program has run, for each choice and action chosen for that choice, we total the rewards that occurred after that point and average that sum with estimates from previous runs.

To illustrate the idea and some problems that arise consider the example below.

```
c1:choose(a,[0,1])
if a then
     r1:reward(10)

c2:choose(b,[0,1])
if b then
     r2:reward(9)
```

Over multiple runs, the learning algorithm will try different values for `a` and `b` and encounter different rewards as a result.

In general the learning algorithm cannot see the structure of the adaptive program. It only sees the program arrive at choices and rewards and experiences this as a serial list of events (sometimes called a trace or trajectory).

A Monte-Carlo learning algorithm is unbiased meaning it makes no assumptions about which previous choices a reward belongs to and thus simply associates rewards with *all* previous choices made. Given enough runs the algorithm will slowly learn the best choices. However, it can take a considerable number of tests to determine the best action to take under each situation.

Learning algorithms all assume that the choice made at `c1` has something to do with getting us to `c2`. Hence, rewards are propagated back to all choices instead of just the last. Yet with a quick inspection of the program we can see that the choices are not related and that these are two different learning problems composed together in sequence; they can and should be solved separately. Unfortunately, a learning algorithm cannot see this structure, and cannot make this distinction. It is unlikely an end user would make this critical observation either.

Continuing with the algorithm anyway, if the *MC* algorithm chose 1, for example, for both choices, it would see a reward of 9 for the second choice (which is correct), but erroneously see a $19 = 10 + 9$ for the first (since rewards get propagated back to all previous choices).

Worse, the structural freedom in ABP permits programmers to order choices and rewards however they please. A transformation that would seem perfectly reasonable to a non-expert might change the previous program into the following, which we will call the TANGLED-IF program.

```
c1:choose(a,[0,1])
c2:choose(b,[0,1])
if b then
     r2:reward(9)
if a then
     r1:reward(10)
```

This program structure is even worse than the previous one. Now both choices will accrue each others' rewards incorrectly. Adding more choices and new rewards under conditional statements makes the problem worse.

However, by applying the algorithm from Figure 2 ABP can determine the correct targets for each reward using the R relation.

Our modified algorithm, which we will refer to as *MCRA* (*MC* with Reward Attribution), extends the previous algorithm only slightly. When confronted with a reward at r, *MCRA* is also passed the contents of the R relation (computed statically). Rewards are only propagated back to choices that may have influenced the current execution path. This small change allows the algorithm to discard a major source of statistical noise in its sampling and learn more efficient choices quicker.

To support the choice invalidation as described in Section 4 we further modify *MCRA* as follows. At each binding statement we first compute the *invalidated* set as described in Section 4. Whenever a choice is invalidated, for any choice context and action, if that context is not used by any other choice (also not invalidated by that assignment), we remove it from the history, tally its rewards, and reset it as we would at the end of an episode thus preventing future rewards from being misattributed to that choice.

5.1 *MCRA* Has Reduced Variance

Since choices do not see rewards that they do not influence, our learner can more effectively estimate the value of various choices. Mathematically, this manifests itself as reduced variance in the reward signal. We give details here.

We define $Q(c,a)$ to be our estimate of the expected total reward our program will receive after making choice a when encountering context c. (Both *MC* and *MCRA* estimate this as a running average.)

Consider the sequence of reward terms when updating these Q values. For the original *MC* algorithm, this sequence can be split into two terms. The first contains only the reward terms potentially influenced by that choice while the second is a residual containing the sum of the remaining terms which provably cannot have been influenced by that choice. Notice that the expected sum of rewards is the definition of the Q value and so we can relate the two Q estimates made by the *MCRA* and *MC* algorithms as follows:

$$Q_{MC}(c,a) = Q_{MCRA}(c,a) + \text{Res}(c,a)$$

Therefore,

$$\begin{aligned}\text{Var}(Q_{MC}(c,a)) &= \text{Var}(Q_{MCRA}(c,a) + \text{Res}(c,a)) \\ &= \text{Var}(Q_{MCRA}(c,a)) + \text{Var}(\text{Res}(c,a)) \\ &\quad + 2\text{Covar}(Q_{MCRA}(c,a), \text{Res}(c,a))\end{aligned}$$

The variance of the new Q estimate for (c,a) will be less than that of the standard estimate as long as,

$$\text{Var}(\text{Res}(c,a)) > -2\text{Covar}(Q_{MCRA}(c,a), \text{Res}(c,a))$$

Since the term on the left is always positive, the above condition only fails when there is a large negative correlation between the sum of influential reward terms and the rest. Although such a negative correlation may occur in pathological cases, it is hard to construct any real examples where this occurs. We are able to verify that a dominant negative correlation does not occur in any of the examples that we studied. In all those cases, the variance in the residual dominates and the modified estimate (Q_{MCRA}) has smaller variance than the standard estimator as expected.

5.2 Spurious Rewards and Choices

Associating choices with the rewards and other choices they influence provides a wealth of knowledge within adaptive programs while permitting the user's program to retain its original structure.

For any reward r, if r does not appear within the R relation anywhere, then the reward is an orphan. Intuitively, this is indicating that no choice the ABP library can make will influence our ability to get that reward, it is spurious.

MCRA could silently ignores these rewards (they are attributed to the empty set of choices). However, spurious rewards are indicative of a programmer error. For example, maybe a choice that used to affect this reward was deemed unnecessary by the ABP programmer and removed, and they forgot to remove the associated reward. Previously, that reward would be awarded to the last choice the learner made and cloud the reward signal with noise.

With the R relation, these orphaned rewards may now be reported explicitly during translation and before program execution.

An analog to the above, is a choice that affects no rewards. If a choice does not appear in the domain of the R relation, we can conclude that it affects no rewards and is also spurious. The impact of leaving this choice in is relatively minor for the *MCRA* algorithm and currently we just ignore the choice. However, though it feels like unreachable code and harmless, if the programmer is expecting that choice to improve, they need to give it a reward signal. Hence, an error message would be quite preferable for this situation.

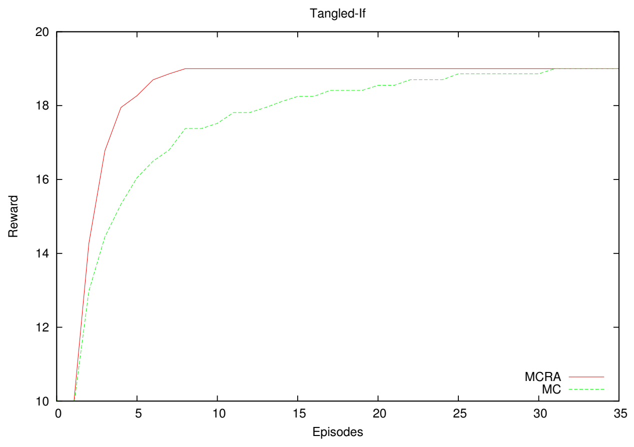

Figure 3: The TANGLED-IF Program (m=64)

6. EMPIRICAL EVALUATION

We implemented a translator for a small language very similar to the one presented in Figure 1 and the algorithm given in Figure 2 to compute the R relation. The translator emits a program, which interfaces with an implementation of ABP.

Since the learner's exploration strategy and parts of adaptive programs are stochastic, evaluation runs into the risk of an initial adaptive program getting lucky and quickly finding an optimal policy, or conversely, getting unlucky and stuck at a local optimum. To mitigate this possibility, we run m independent copies of the adaptive program (each with their own random seeds) in parallel and average their learning behavior.

We run each of the m learners a few runs and then periodically sample the optimal learned behavior (its policy).[2] In addition, when testing the policy of each of the m learners we test them t times so if the adaptive program has random behavior, we can get an average with respect that policy.

The TANGLED-IF

Figure 3 shows the learning behavior of the TANGLED-IF program given in a previous section. Even with the reward mis-attribution the underlying learning problem is remarkably simple to solve, yet the newer algorithm *MCRA* clearly learns faster. In less than 10 tries all $m = 64$ copies of the program learned the optimal policy while the older algorithm *MC* took around 25 to 30 attempts to reach that point.

The OPTIMAL-CONFIGURATION

We can generalize the TANGLED-IF and arrive at an even worse, but more realistic scenario: a program has to pick a set of configuration options during program initialization and then use those chosen options throughout the program.

Below is a synthetic program to model the described case, which we refer to as the OPTIMAL-CONFIGURATION problem.

[2]No exploration is performed during evaluation and the ABP learner always chooses the best known option for each choice.

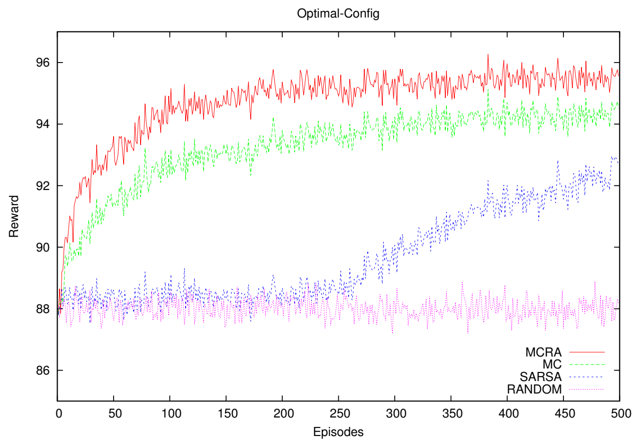

Figure 4: The OPTIMAL-CONFIGURATION Problem (m=32, t=32)

```
c1:choose(c1,[0,1])
c2:choose(c2,[0,1])
...
c8:choose(c8,[0,1])

t := 0
while t < 32 do
   t := t+1
   e := uniformR(1,9)
   r := normal() + u
   if e == 1 then
        if (c1) then
            r1a:reward(1.0 + r)
        else
            r1b:reward(0.5 + r)
   else if e == 2 then
        ... similarly
   ...
   else if e == 8
   ...
```

To make it bit more interesting we use a stochastic reward, `normal` returns a normally distributed [3] random number and `uniformR(1,9)` is used to uniformly select a configuration to "use" (`uniformR` selects from a half-open interval). In all cases we make one configuration option better than its alternate so we can easily determine optimal behavior analytically for testing.

The results of this algorithm are shown in Figure 4. *MCRA* performs better than the regular *MC* algorithm determining the optimal set of configurations after 100 to 200 attempts whereas the base algorithm has not learned the optimal policy even after 500 episodes (but is close).

For this program we also compare the algorithm to two others. The *SARSA(λ)* algorithm is a venerable and robust learning algorithm and was used in older versions of ABP as the learning algorithm. Like the *MC* algorithm, it will make an incorrect assumption that each choice is dependent on the next instead of all being independent as in this example. While it comes in third, it is quite impressive how well it actually performs considering the ill fit. Finally the *random* algorithm is at the bottom of the plot and is given to illustrate the estimated reward we would get by guessing randomly.

[3]Note, in a real optimal configuration scenario, the user's program would not necessarily know the underlying data distributions of the rewards and configuration uses.

The ROBOT PROBLEM

We now proceed to illustrate a larger ABP program below that exhibits some of these reward-attribution problems in a slightly larger example. Our program consists of several partially independent goals.

```
goods := START_GOODS
goals := 0
c1:choose(sell_threshold,[1,2,3,4])
gx := InitGoalX()
gy := InitGoalY()
x := 0
y := 0
t := 0
while (t < MAX_TIME) do
  t := t + 1
  price := GoodsPrice()
  if (goods > 0 && price > sell_threshold) then
     r1:reward(price)
     goods := goods-1

  c2:choose(m,context(dir(x,gx),dir(y,gy)),
            [N,E,S,W,X])
  x := ApplyMoveX(m,x)
  y := ApplyMoveY(m,y)
  if (x==gx && y==gy) then
     goals := goals + 1
     r2:reward(2)
     gx := InitGoalX()
     gy := InitGoalY()

if (goods == 0 && goals >= 2)
   reward(4)
```

This program controls a robot with position $(\mathtt{x},\mathtt{y})$ moving around a grid. The robot gets to move one square each time step (loop iteration) in one of the cardinal directions (`N`,`E`,`S`,`W`) or it can stay in place (`X`). The robot is trying to reach its goal at $(\mathtt{gx},\mathtt{gy})$. Moreover, each time the robot reaches its goal, it gets a reward and then requests a new goal by calling the `InitGoalX` and `InitGoalY` functions. The move choice at `c2` takes a context consisting of a normalized direction towards the robot's goal. The function `context` pairs those vectors into one value.

Additionally, there is a partially independent secondary goal. Initially, we start with a certain number of goods we must sell over the course of time (we don not know how much). Each time step, a new random market price for these goods is given and we must choose to sell or retain the goods. We deal with this uncertainty up front with choice `c1`.

As a third and final goal an extra bonus reward is given if we sell all our goods by the end of the game and achieve at least two robot-movement goals.

Figure 5 shows the learning behavior of our various algorithms on this problem. *MCRA* performed the best, most if not all $m = 32$ learning trials found the optimal behavior within a few hundred trials. The next best algorithm was *SARSA*(λ). Instances of this algorithm found the optimal, but some got stuck at sub-optimal policies.

The regular *MC* algorithm did not perform very well and although some learning runs did find an instance of the best known policy, far fewer did.

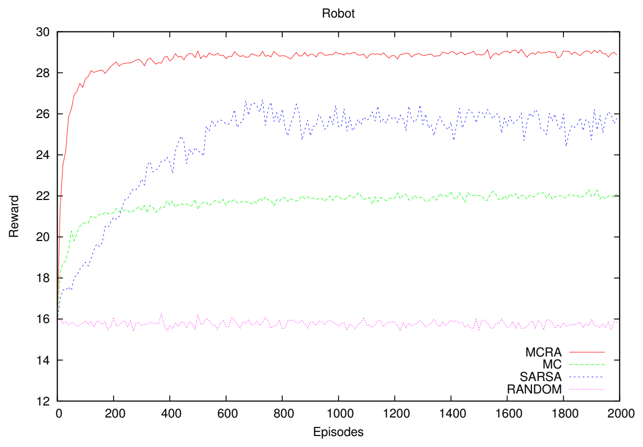

Figure 5: ROBOT Problem Learning (t=64,m=32)

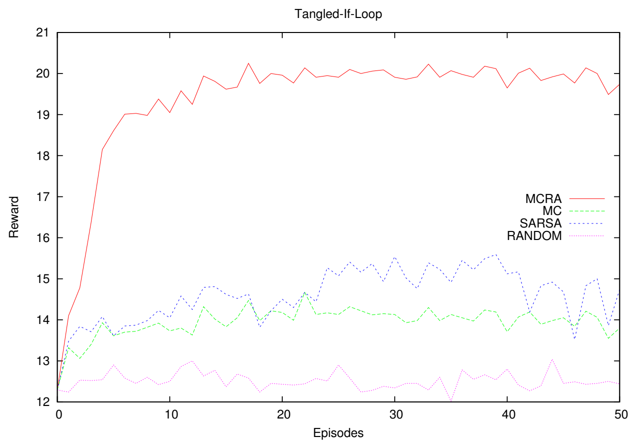

Figure 6: The TANGLED-IF-LOOP Program (t=32,m=32)

The TANGLED-IF-LOOP *Problem*

To test the benefit of the choice invalidation we consider the the following program.

```
i := 0
while i < 10 do
   i := i + 1
   ca:choose(a,[0,1,2,3])
   cb:choose(b,[0,1,2,3])
   r = normal() + 0.5
   if b == 1 then
         rb:reward(r + 0.5)
      else
         rb:reward(r)
   if a == 1 then
         ra1:reward(r + 0.5)
      else
         ra2:reward(r)
```

This program is a variation on the TANGLED-IF problem given previously. We add a few more options for each choice and randomize the reward slightly to might it slightly more difficult. However, the problem still has a unique optimal solution.

Figure 6 shows the result and illustrates that with *MCRA* all $m = 32$ learners quickly learn the optimal solution within about 10 runs, whereas the other methods struggle.

7. RELATED WORK

Recent work in the field of reinforcement learning (RL) [14] under the name of partial programming [5] inspired and influenced the development ABP. RL is a subfield of artificial intelligence and machine learning that studies algorithms where an agent (sometimes called a controller) takes some action given specific environment information (state). In response the agent (learning algorithm) receives a reward from the environment (sometimes called a simulator) for the action chosen and transitions to a new state continuing that way until it reaches a terminal state. The simulator describes and models the external world, models the domain for the problem being solved, and feeds information to the agent based on observed state transitions, rewards, and other events. The aim of the RL agent is to maximize its total reward received.

In our work the ABP library may be viewed as the RL agent and any appropriate RL learning algorithm could potentially be used for that part of the library. Conversely, the user's partial program along with its use of ABP roughly corresponds to the simulator, which monitors and observes the environment and communicates this information to the agent. Thus RL can be viewed as an extreme version of ABP with a trivial non-adaptive program that makes a single choice based on all its observational data each time step and implements the resultant action the agent returns.

However, successful application of RL to systems requires knowledge of the specific algorithm being used, complex tuning, and experimentation to function effectively. Various systems have experimented with methods of hinting which action is preferred in a given state or set of states [9]. In a similar vein, [1] experimented with providing advice via examples of good behavior rather than classifying it directly with a reward function. However, these approaches still require an RL expert to apply them as they are still tightly coupled to RL theory and specific algorithms in RL.

Making RL more accessible to the non-expert user has been studied prior under the name partial programming [5], and previous work in this field has influenced the design of ABP. The idea of partial programming was initially presented in the language ALISP, a direct integration of RL in the LISP programming language and also in its later variants of that language [4]. ALISP specifies uncertainty through a *choicepoint* construct, similar in function to our `choose` operation. Over repeated runs a learning algorithm learns the best options for each choice by observing rewards and thus completes the program. The focus of the ALISP work is on hierarchical decomposition of learning problems through subroutine abstraction, and later, through support for concurrent [10] and semi-independent tasks. None of this work considers programmer mistakes such as reward mis-attribution or choice invalidation.

A separate approach to partial programming was presented in the A^2BL [8] language, an integration of RL into the Agent Behavior Language (ABL). A^2BL treats sub-agents as independent behavioral components all interacting together rather than one complete learning problem. The result is a declarative language for specifying behaviors and goals over a hierarchy of agents, and thus this language has a more limited scope than general-purpose languages such as ALISP or ABP implementations. Moreover, in A^2BL the programmer indicates exact conditions when an agent should receive a reward, and no automatic reward attribution is performed.

In our previous work [6] we implemented ABP for Java. The learning algorithm was controlled by Q-learning [15] initially and later by *SARSA*(λ) [14]. Both of these algorithms assume (and leverage) a strong property about the order of choices and rewards within the program. If this property is correctly adhered to, it can benefit the learning algorithm, but if violated, can increase the difficulty of the learning problem.

Later work in ABP [11] showed some simple ABP programs that violated the ordering constraint and operated sub-optimally as a result. To address the problem, that work assumed the existence of a data flow analysis similar to the one defined here and showed an algorithm that made use of it, however, no exact details of the data-flow analysis or its computability were considered. The goal was to illustrate how such an analysis could benefit the learning algorithm and how it could correct the choice ordering problem, not how the analysis could be derived. Conversely, we have given a formal definition to such an analysis and are less focused on applying it to the most sophisticated RL algorithms.

Exploratory implementations of ABP using function approximation and policy gradient learning methods instead of discrete state-action spaces show promising results [12]. These methods were shown to operate effectively on some of the ill-formed programs given in prior work [11] as well as a few additional cases. However, this approach incurs a usability cost in that contexts may no longer be arbitrary values, but must be binary vectors representing important features of the current program state.

Considerable work has been performed with the goal of optimizing or transforming template programs safely. For instance work by Willcock [16] explores the issue of programmable optimizations for user-defined types. Similarly work by Agakov [2] explores the issue of how to use machine learning to speed up program improvement via iterative compiler optimization. However, the notion of optimization in all this work is strictly limited to performance improvements (space and time), and the semantics of the underlying program must remain constant. In contrast, our view of a program template is a partial program with non-determinism (choices) and a programmable notion of optimization (rewards). Consequently different template instantiations are expected to generate programs with different behaviors instead of performance.

8. CONCLUSION

ABP provides a method for non-experts to use machine-learning techniques to write adaptive program templates. Over repeated runs, these templates can be instantiated and generate an improving sequence of concrete programs.

An important concept in ABP is the freedom we allow the programmer in structuring their programs. The cost of this freedom is inefficiency and sub-optimal learning. We have developed a simple data-flow analysis that corrects many problems that occur due to mis-attribution of rewards to choices. With data-flow-driven reward attribution, the ABP programmer can focus on their programming problem and ignore complications that arise while mapping rewards to the choices that influenced them.

Acknowledgments

This work is supported by the National Science Foundation under the grant CCF-0820286 "Adaptation-Based Programming".

9. REFERENCES

[1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *International Conference on Machine Learning*, pages 1–8, 2004.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2nd edition, 2006.

[4] D. Andre and S. Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*, pages 119–125, 2002.

[5] David Andre. *Programmabler Reinforcement Learning Agents*. PhD thesis, University of California at Berkeley, 2003.

[6] T. Bauer, M. Erwig, A. Fern, and J. Pinto. Adaptation-Based Program Generation in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 81–90, 2011.

[7] T. Bauer, M. Erwig, A. Fern, and J. Pinto. Adaptation-Based Programming in Haskell. In *IFIP Working Conference on Domain-Specific Languages*, pages 1–23, 2011.

[8] S. Bhat, C.L. Isbell, and M. Mateas. On the difficulty of modular reinforcement learning for real-world partial programming. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, AAAI'06, pages 318–323. AAAI Press, 2006.

[9] R. Maclin, J. Shavlik, L. Torrey, T. Walker, and E. Wild. Giving advice about preferred actions to reinforcement learners via knowledge-based kernel regression. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 819–824, 2005.

[10] B. Marthi. Concurrent hierarchical reinforcement learning. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 4*, AAAI'05, pages 1652–1653. AAAI Press, 2005.

[11] J. Pinto, A. Fern, T. Bauer, and M. Erwig. Robust learning for adaptive programs by leveraging program structure. In *ICMLA*, pages 943–948, 2010.

[12] J. Pinto, A. Fern, T. Bauer, and M. Erwig. Improving policy gradient estimates with influence information. *Journal of Machine Learning Research*, 20:1–18, 2011.

[13] R. Sutton. *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 1984. AAI8410337.

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

[15] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989.

[16] J. Willcock, A. Lumsdaine, and D. Quinlan. Reusable, generic program analyses and transformations. *SIGPLAN Not.*, 45(2):5–14, October 2009.