AN ABSTRACT OF THE DISSERTATION OF

Tim Bauer for the degree of Doctor of Philosophy in Computer Science presented on January 31, 2013.

Title: Adaptation-Based Programming

Abstract approved: _________________________________________________________________

Martin Erwig

Partial programming is a field of study where users specify an outline or skeleton of a program, but leave various parts undefined. The undefined parts are then completed by an external mechanism to form a complete program. Adaptation-Based Programming (ABP) is a method of partial programming that utilizes techniques from the field of reinforcement learning (RL), a subfield of machine learning, to find good completions of those partial programs.

An ABP user writes a partial program in some host programming language. At various points where the programmer is uncertain of the best course of action, they include choices that non-deterministically select amongst several options. Additionally, users indicate program success through a reward construct somewhere in their program. The resulting non-deterministic program is completed by treating it as an equivalent RL problem and solving the problem with techniques from that field. Over repeated executions, the RL algorithms within the ABP system will learn to select choices at various points that maximize the reward received.

This thesis explores various aspects of ABP such as the semantics of different implementations, including different design trade-offs encountered with each approach. The goal of all
approaches is to present a model for programs that adapt to their environment based on the points of uncertainty within the program that the programmer has indicated.

The first approach presented in this work is an implementation of ABP as a domain-specific language embedded within a functional language. This language provides constructs for common patterns and situations that arise in adaptive programs. This language proves to be compositional and to foster rapid experimentation with different adaptation methods (e.g. learning algorithms). A second approach presents an implementation of ABP as an object-oriented library that models adaptive programs as formal systems from the field of RL called Markov Decision Processes (MDPs). This approach abstracts away many of the details of the learning algorithm from the casual user and uses a fixed learning algorithm to control the program adaptation rather than allowing it to vary. This abstraction results in an easier-to-use library, but limits the scenarios that ABP can effectively be used in. Moreover, treating adaptive programs as MDPs leads to some unintuitive situations where seemingly reasonably programs fail to adapt efficiently. This work addresses this problem with algorithms that analyze the adaptive program’s structure and data flow to boost the rate at which these problematic adaptive programs learn thus increasing the number of problems that ABP can effectively be used to solve.

This work demonstrates a powerful new model for writing adaptive computer programs, and seamlessly integrates advanced RL technologies into general-purpose programming languages.
Adaptation-Based Programming

by

Tim Bauer

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented January 31, 2013
Commencement June 2013

APPROVED:

________________________________________________________________________
Major Professor, representing Computer Science

________________________________________________________________________
Director of the School of Electrical Engineering and Computer Science

________________________________________________________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

________________________________________________________________________
Tim Bauer, Author
ACKNOWLEDGEMENTS

I wish to thank my advisor Martin Erwig for guiding me through the arduous Ph.D. process and for providing me with a totally new set of tools for reasoning about, understanding, and developing programming languages. I wish to thank all my colleagues and instructors at Oregon State and Cal-Poly over the past years for igniting and fueling my interest in so many different aspects of the science of computing. Finally, I wish to offer a special thanks to all my family for providing unwavering support throughout the process, for believing in me, and for convincing me that I was capable of performing such a task.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Adaptive Programming</td>
<td>4</td>
</tr>
<tr>
<td>1.1.1 Adaptation Constructs</td>
<td>7</td>
</tr>
<tr>
<td>1.1.2 Choice Contexts</td>
<td>8</td>
</tr>
<tr>
<td>1.1.3 Feedback Mechanisms</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Contributions</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Outline</td>
<td>11</td>
</tr>
<tr>
<td>2 Background</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Reinforcement Learning</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Markov Decision Processes</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Mapping Adaptive Programs to Reinforcement Learning Problems</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1 Formalizing the Semantics of ABP Programs</td>
<td>19</td>
</tr>
<tr>
<td>2.4 State Abstraction</td>
<td>25</td>
</tr>
<tr>
<td>3 Literature Review</td>
<td>27</td>
</tr>
<tr>
<td>3.1 ALISP-Based Approaches</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 HAM — Hierarchies of Abstract Machines</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2 PHAM — Programmable Reinforcement Learning Agents</td>
<td>32</td>
</tr>
<tr>
<td>3.1.3 ALISP</td>
<td>34</td>
</tr>
<tr>
<td>3.1.4 Concurrent ALISP</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Adaptive Agent Behavior Languages</td>
<td>41</td>
</tr>
<tr>
<td>3.3 Program Sketching</td>
<td>46</td>
</tr>
<tr>
<td>3.4 Systems for Automatic Configuration and Self-Aware Computing</td>
<td>54</td>
</tr>
<tr>
<td>3.4.1 FFTW — Learning a Faster Fourier Transform</td>
<td>54</td>
</tr>
<tr>
<td>3.4.2 PetaBricks</td>
<td>55</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>59</td>
</tr>
<tr>
<td>4 Functional Adaptation-Based Programming</td>
<td>61</td>
</tr>
<tr>
<td>4.1 Adaptive Values</td>
<td>63</td>
</tr>
<tr>
<td>4.1.1 Defining Adaptives</td>
<td>63</td>
</tr>
<tr>
<td>4.1.2 Derived Adaptives</td>
<td>69</td>
</tr>
<tr>
<td>4.1.3 Contextual Adaptives</td>
<td>70</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4 Nested and Recursive Adaptives</td>
<td>71</td>
</tr>
<tr>
<td>4.2 Programs for Adaptive Computation</td>
<td>74</td>
</tr>
<tr>
<td>4.2.1 Adaptation Combinators</td>
<td>75</td>
</tr>
<tr>
<td>4.2.2 Recursive Adaptation</td>
<td>78</td>
</tr>
<tr>
<td>4.2.3 Transactional Adaptations</td>
<td>82</td>
</tr>
<tr>
<td>4.3 Monitoring Adaptation Behavior</td>
<td>84</td>
</tr>
<tr>
<td>4.4 Empirical Results</td>
<td>86</td>
</tr>
<tr>
<td>4.4.1 Adaptive Sorting</td>
<td>86</td>
</tr>
<tr>
<td>4.4.2 Adaptive Budgeted Optimization</td>
<td>87</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>96</td>
</tr>
<tr>
<td>5 Object-Oriented Adaptation-Based Programming</td>
<td>100</td>
</tr>
<tr>
<td>5.1 Example Scenario</td>
<td>101</td>
</tr>
<tr>
<td>5.2 Adaptation Concepts</td>
<td>105</td>
</tr>
<tr>
<td>5.3 Constraining Adaptation Behavior</td>
<td>111</td>
</tr>
<tr>
<td>5.4 Evaluation</td>
<td>112</td>
</tr>
<tr>
<td>5.4.1 Yahtzee</td>
<td>113</td>
</tr>
<tr>
<td>5.4.2 Java ABP in Automated Testing</td>
<td>118</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>120</td>
</tr>
<tr>
<td>6 Automatic Reward Attribution Through Data Flow Analysis</td>
<td>122</td>
</tr>
<tr>
<td>6.1 A Simple Adaptation Programming Language</td>
<td>123</td>
</tr>
<tr>
<td>6.2 Reward Attribution Algorithm</td>
<td>126</td>
</tr>
<tr>
<td>6.3 Choice Invalidation</td>
<td>131</td>
</tr>
<tr>
<td>6.4 A Modified Learning Algorithm</td>
<td>134</td>
</tr>
<tr>
<td>6.4.1 MCRA Has Reduced Variance</td>
<td>137</td>
</tr>
<tr>
<td>6.4.2 Spurious Rewards and Choices</td>
<td>138</td>
</tr>
<tr>
<td>6.5 Empirical Evaluations</td>
<td>139</td>
</tr>
<tr>
<td>6.6 Summary</td>
<td>147</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>150</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (Continued)

<table>
<thead>
<tr>
<th>Bibliography</th>
<th>152</th>
</tr>
</thead>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An ABP template is a partial program and expands into a space of completions (concrete programs).</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>The typical arrangement of a system using reinforcement learning.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>An example MDP. The big circles are states the process might be in, the small circles are actions that can be chosen. Upon choosing an action, the MDP transitions to a new state with a certain probability (indicated on the lines from the actions to the successor states).</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>An MDP for the NAVIGATION problem. Each subgraph corresponds to a different goal (chosen at run time).</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>The example problem from Parr and Russell’s work [55] (figure copied). The entire grid maze (a), a zoomed in view of an obstacle within that maze (b), and one of the machines in the hierarchy used for constraining this problem (c).</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Taxi World (from Andre’s ALISP paper [4]).</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>This figure illustrates of differences between HAM/ALISP and ABP. ALISP splits state management and reward attribution off into an external module (MDP), but drives program flow and state transition with actions. However, its structure depends heavily on those states and rewards within M (the dotted line). ABP merges these components and makes this dependency explicit by passing a state to the algorithm and receiving a suggested action to take in return.</td>
<td>38</td>
</tr>
<tr>
<td>4.1</td>
<td>An example of the LM algorithm using the Rosenbrock function (the true minimum value is (1,1)) as an example function. The algorithm starts at any point ((0.5, –0.5) in this example). Then at each point it computes the gradient and steps in that direction by a function of λ. If a step “overshoots”, it backs up and adjusts the λ parameter as in the 4th sample in this example.</td>
<td>89</td>
</tr>
<tr>
<td>5.1</td>
<td>This illustrates the wolf hunting problem on a 3 x 4 grid. The x-coordinate wraps, and the wolves must work together to trap the rabbit.</td>
<td>102</td>
</tr>
<tr>
<td>5.2</td>
<td>Average number of moves to catch the rabbit.</td>
<td>111</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3</td>
<td>113</td>
</tr>
<tr>
<td>An example Yahtzee game. The possible categories are on the left showing only two filled. The current dice are on the right. Here the player is choosing to re-roll one of the 5 dice with the hope of getting a 4 and getting a large straight (1, 2, 3, 4, 5).</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>117</td>
</tr>
<tr>
<td>Performance of the Yahtzee program as a function of the number of learning games.</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>122</td>
</tr>
<tr>
<td>A programmer generates two independent values a and b from adaptive variables given contexts s and t and then rewards them independently. The programmer desires to print the two values together before computing the reward and must first reward across a suggest to accomplish this. The move intuitively seems safe since the rewards functions are clearly independent (f is only a function of a, not b), but the ABP library cannot see this independence and the underlying MDPs are different.</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>124</td>
</tr>
<tr>
<td>Syntax for a Simple Language</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>127</td>
</tr>
<tr>
<td>Inference Rules for Reward Attribution</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>140</td>
</tr>
<tr>
<td>The TANGLED-IF Program (m=64)</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>143</td>
</tr>
<tr>
<td>The OPTIMAL-CONFIGURATION Problem (m=32, t=32)</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>146</td>
</tr>
<tr>
<td>ROBOT Problem Learning (t=64, m=32)</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>148</td>
</tr>
<tr>
<td>The TANGLED-IF-LOOP Program (t=32, m=32)</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Comparison of Adaptive Programs and Markov Decision Processes</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of Partial Programming Approaches</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance on Yahtzee. All results are averaged over 1000 games.</td>
<td>116</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Programs that implement deterministic algorithms require exact and detailed instructions of what to do at all steps. However, many algorithms can be much easier to specify if the programmer is permitted to include a small bit of uncertainty (non-determinism) in their program. Even if a programmer can initially present a deterministic algorithm for all possible inputs they typically just make guesses about various characteristics of the input distribution and the environment the program will run in since these properties are not observable until program run time.

To illustrate this point, consider standard sorting algorithms. Many libraries use a divide-and-conquer approach such as mergesort or quicksort on large lists and then switch to insertion sort on the small sublists. The decision of what is small is an instance of input uncertainty in such an algorithm. Given random data and fixed hardware there is some optimal threshold value for the sorting algorithm where it switches to insertion sort. A rigorous developer might even sample the performance of their algorithm on random lists of integers to compute an optimal cutoff threshold to switch algorithms and then hardcode this cutoff into the algorithm (or define it as a symbol with a constant value). For instance, consider sorting algorithms from various libraries:

- The Microsoft Visual C standard library’s sorting algorithm `qsort` uses a median-of-three quicksort, but switches to a quadratic algorithm for lists smaller than 8.\(^1\)

\(^1\)From `qsort.c` provided in the C run-time library with MSVC++ 2010.
• The Java Collections API [6] maintains a block of threshold constants for various algorithms such as sorting and searching. Their default sorting algorithm of mergesort switches to an insertion sort for lists smaller than 7 elements as well.²

• The Standard Template Library has used 15 as a cutoff in std::sort for years. Recent work [5] has shown this to be much too low.

This ad-hoc “back-of-the-envelope” optimization pattern is frequently used in software development. More systematic approaches have been proposed to solve this type of problem. Section 3.4 discusses some of those approaches.

By fixing such cutoff values in the library definition the developer has just made some assumptions up-front about the operating environment of their algorithms. For instance, a developer might assume the cost of comparison on their sorting algorithm to be the cost of comparing integers. However, the algorithm may then be applied to more complex data-types with more time-consuming comparison logic. Similarly, the developer may assume their algorithm will always work randomly distributed input when in fact the algorithm more frequently works with (nearly) sorted input. Hence insertion sort is nearly linear in its execution time and should be favored. In both the above cases a larger threshold will be appropriate. However, the developer cannot anticipate this at development time; they must just guess a reasonable value.

A different sort of strategic uncertainty also exists in algorithm development, one of choosing the best strategy independent of the types of inputs the program might see. For instance, in computer games a computer-controlled player must make decisions in an attempt to defeat its opponent. How it chooses to go about this might be unclear and thus may be an appropriate play to use a non-deterministic programming construct. The key difference is that in the previous

²From the java.util.Arrays class in the JDK 1.6
case success is somehow measured by program performance, yet in a simulation the notion of success is not necessarily dependent on program time or space usage.

One systematic way of addressing uncertainty in program development is to manually evaluate the various options in a controlled environment that models the expected usage. For instance, it is common to manually test various cutoff and threshold values on different platform configurations and environments that the program is expected to run in (much like in the sorting library examples above). From these tests a “good” value is selected that generally works well for all cases.

However, a different view is to model such uncertainty explicitly within the program itself. Such an approach permits the programmer to write partially specified programs with definitions that permit them to adjust and fit to the environment they are executing in. This thesis refers to such programs as adaptive programs.

Adaptive programs in this model are able to self-adjust or adapt given example inputs. They react to a programmable feedback mechanism, which specifies how well the adaptive program is doing. In the simplest case this feedback signal could be a numeric reward or penalty, perhaps corresponding to execution efficiency (e.g. running time). However, more sophisticated feedback mechanisms might include error correction terms (that directly drive the adaptation).

Finally, it is desirable for this framework to be accessible to the “average user”. There have been years of research into methods of learning functions in the fields of machine learning and artificial intelligence. However, these fields are typically quite sophisticated and require significant expertise and practice to use effectively. By presenting a simpler interface to users, adaptive languages can hide some of the complexity introduced by the algorithms that drive adaptation. One form of programming that naturally supports the requirement of keeping the problem specification in the user’s domain is partial programming [4].
Partial programming is a form of automatic programming where a user partially specifies a program’s specification and leave parts unfilled or unspecified. Systems for partial programming use various mechanisms to fill in the unspecified parts with intelligent choices to generate a full program specification. The specified parts can be viewed as constraints, and the unspecified parts can be thought of as variables. Most importantly, partial programming systems can be defined in almost any domain. Hence, such a tool permits the definition of a system where the programmer can directly define their programming problem as a partial program in a general-purpose language rather than translating it into some specialized system.

This thesis presents the Adaptation-Based Programming (ABP) paradigm of programming to provide the above-described non-deterministic programming model to users in the domain of general-purpose programming. This work also evaluates the effectiveness of several adaptive programming languages (languages with ABP support) through empirical examples and explores some of the problems encountered in these languages as well as evaluating solutions to these problems. Furthermore, we compare this programming model to other approaches similar to ABP in goal or function.

1.1 Adaptive Programming

In partial programming, a relatively new field in programming language research, one may specify a skeleton program, but omitting various pieces. Those omitted pieces are completed (filled in) automatically via some form of automatic programming such as program synthesis or machine learning. Adaptation-Based Programming is a specialized kind of partial programming that studies ways of extending languages with constructs that adapt. The goal of this approach is to abstract away some of the underlying learning methods used in partial programming (such as machine learning and program synthesis) so that the user of the adaptive language need not
represent their problem explicitly in those domains. In other words, part of the goal of ABP is to make these optimization technologies more accessible to the average user. ABP should permit adaptive programs to exhibit the general structure of the underlying host language rather than requiring a problem conversion step.

The approach taken by this thesis involves applying the general principles of ABP to different general-purpose languages in ways that best fit those languages. Different languages require different types of constructs to gracefully represent adaptive concepts. For instance, in an object-oriented language objects that represent adaptive values are a natural fit. Conversely, adaptive functions may make more sense in a functional language’s implementation of ABP. Chapters 5 and 4 apply ABP to each of these paradigms, respectively.

In addition to exploring how ABP fits into industrial-strength languages this work considers a version for a small imperative language similar to the WHILE language detailed in [54] or similar languages from other compiler texts [1]. The reasons for this are two-fold. First, these languages are very simple, but complete. Hence, they are appropriate examples of minimalistic languages that can be modified to be adaptive. Second, and more importantly, Chapter 6 uses this simple language to demonstrate the impact of adaptation on data flow, and shows how a compiler or interpreter can reason about program structure to make better decisions about how to adapt programs efficiently.

Any adaptive language requires at least two adaptive constructs. The first construct is a *choice* operation, which indicates how and where the program can adapt and vary. Choices allow a programmer to explicitly indicate uncertainty with a point for non-determinism in the program. For an imperative language this could be realized with a *choose* procedure, which non-deterministically selects amongst a list of arguments passed in. However, other approaches are also possible here. For instance, in the Program Sketching approach to partial programming
(discussed in Section 3.3) choices consist of fragments of program syntax that are expanded much like a macro before the program runs rather than during run time.

The second necessary construct for an ABP implementation is a feedback mechanism. This construct drives the adaptive process and influences future choices. A common direct approach for a feedback system is to use a numeric reward that indicates good or bad behavior. For an imperative language a reward procedure would be suitable. Other possible approaches for feedback might be correctness assertions such as in [65] or error terms that somehow directly influence future terms. Chapter 4 explores some of these alternative approaches.

![Figure 1.1: An ABP template is a partial program and expands into a space of completions (concrete programs).](image)

A program with choices is an adaptive program. Choices can be thought of as non-deterministic values at various points in an adaptive program. A program instance is an adaptive program with
all of its choices fixed to particular values. An adaptive program corresponds to a space of pro-
gram instances as illustrated in Figure 1.1.

One important design decision is in how to express choices in an adaptive program. For
instance, are choices made statically (compile-time) or dynamically (as the program runs)? In
other words, does a choose function expand to a fixed program fragment (like a fancy macro)
or is it adjusts at execution time (like a fancy function)? The static option is appealing
since it allows the programmer to instantiate concrete tangible programs, which may be tested
separately, and where the meaning of each concrete program is easily understood. However, the
cost of this clarity is expressiveness. Moreover, this static approach limits the user by forcing
them to make early decisions, and choose from statically defined values.

1.1.1 Adaptation Constructs

To illustrate the spirit of ABP this section presents a simplistic optimization problem and show
an adaptive program to model it. Consider a controller for a semi-autonomous robot. The robot’s
goal is to transmit as many units of data as possible over the course of a full day. Each hour the
robot must decide whether or not to attempt to transmit or not. Moreover, the data transfer rate
varies each hour due to a varying signal strength (represented by one of a set of fixed values).
Finally, the robot has a finite battery, and may only transmit for a few hours before it must be
recharged.

In a simple imperative language a programmer might represent this problem as follows.
tx = 0    // total transferred
h = 0     // current hour
b = INIT_BATTERY    // battery left
s_cutoff = choose([0,1,2,3,4])
while (h < HOURS_IN_DAY)
   h = h + 1
   s = CurrentSignalStrength()    // one of [0,1,2,3,4]
   if (b > 0 && s >= s_cutoff)
      tx = tx + Transmit()
      b = b - 1
   else
      Sleep()
reward(tx)

The choose function is the choice construct for this simple language. It will select one of the list of values passed in. The reward function gives this language a simple means of evaluating success numerically. In this problem, the goal is to maximize tx, which is a measure of the number of units successfully transmitted. The Transmit function activates the transmitter for an hour and returns the number of units successfully transferred over that hour. The Sleep function deactivates the transmitter and sleeps for an hour.

The above adaptive program \( P \) is quite simple, it expands to a space of only five possible completions or program instances (one for each value in the list passed to choose).

### 1.1.2 Choice Contexts

Reconsider the signaling optimization problem presented above. An alternate way of viewing this problem is to observe that each hour the program must decide whether to attempt to transmit or not. This decision is potentially influenced by three factors: how much time there is left (\( h \)), how much battery power remains (\( b \)), and the current signal strength (\( s \)) that hour. This

\(^3\)List literals are given in square brackets.
contextual information illustrates that choices may depend on context. Hence, it makes sense to extend the choose function to take a recognition context argument to aid in the adaptation. A program below illustrates this variation.

```plaintext
tx = 0       // total transferred
h = 0        // current hour
b = INIT_BATTERY  // battery left
while (h < HOURS_IN_DAY)
    h = h + 1
    s = CurrentSignalStrength() // one of [0,1,2,3,4]
    if (b > 0)
        attempt_transfer = choose((s,b,h),[TRUE,FALSE])
        if (attempt_transfer)
            tx = tx + Transmit()
            b = b - 1
        else
            Sleep()
    reward(tx)
```

The tuple \((s, b, h)\) argument to choose represents the context relevant in making a good decision at that point in the program.

The above program can still be viewed as a template that expands to a space of completions. However, the meaning of a program instance must be refined. Instead of replacing the choice with a concrete value (one of its alternatives), the choice is replaced with a function mapping context values to an optimal alternative. In this model, one could still express the previous context-less version of the program by providing a dummy value as the context (e.g. the unit value or an empty tuple).
1.1.3 Feedback Mechanisms

The underlying adaptation system in ABP is responsible for driving program change in such a way that future program instances improve in some way. Hence, it requires some sort of metric to create a meaningful evaluation of program instances. For this simple language, and in most cases, a simple \texttt{reward} operation suffices (realized as a function in this language). The function takes a numeric argument where greater values are more desirable than lesser values. Finally, program fitness is realized as an \textit{expected} sum of all these rewards for a program instance.

In general the adaptation system can never see the program input or all aspects of the external environment it is running in. Hence even a program instance (i.e. choices are fixed) might encounter radically different reward signals for the same inputs. For example, the reward in the above signaling program is a function of whatever the \texttt{Transmit} procedure returns. This subroutine could be stochastic and might operate differently in different weather for instance. Consequently, the expected sum of rewards is not necessarily deterministic.

The adaptation system makes use of the reward it sees to make better decisions in later program runs. Exactly how this is achieved is a discussion deferred to later; for now, suffice it to say the adaptation system desires to maximize its reward (or sum of rewards if \texttt{reward} is called multiple times). However, this implies that the adaptive programmer is responsible for placing rewards in such a way that it correctly reflects the programmer’s goals.

1.2 Contributions

This thesis makes the following contributions.

\footnote{This convention can easily be reversed so that a \texttt{cost} or \texttt{punish} operation that discourages higher values would be equivalent.}
• It presents ABP as a generic embedded domain-specific language (DSL) hosted within a functional language from earlier work [11]. This DSL explores the meaning of adaptation within a purely functional language (i.e. side effects are not allowed in functions) and explores different ways of representing adaptive programs. The language is centered around the idea of adaptive values. This work explores example adaptation patterns that occur in adaptive programs in this language and generalizes them into higher-level constructs and combinator functions.

• This work explores ABP as object-oriented library from work in [10], which shows how adaptive programming can be realized within that language paradigm. Adaptivity is expressed within an object model, which presents choices as adaptive variables within a larger adaptive process. This approach also explores how to leverage the sequential nature of programs to rapidly improve them.

• Finally, this thesis presents a data-flow method [12] for automatically associating choices with the rewards they influence within an adaptive program as well as learning algorithms to exploit that information statically. This information can speed up learning (sometimes drastically) and detect subtle errors in adaptive programs.

1.3 Outline

The rest of the thesis is structured as follows.

Chapter 2 discusses some background material relevant to this thesis including a discussion of how to use techniques from the field of artificial intelligence to drive adaptation and make programs adapt.
Chapter 3 reviews literature in fields surrounding this thesis. It surveys related approaches to partial programming within various disciplines such as machine learning, program synthesis, and automatic system configuration.

Chapter 4 explores a possible adaptive program implementation within the context of the strongly-typed functional language Haskell. This chapter presents ABP as an embedded DSL and demonstrates its utility on several problems.

Chapter 5 presents a significantly different view of ABP in the context of an object-oriented languages. This approach extends the functional approach by modeling the sequence of choices the program makes in order to leverage more powerful reinforcement learning algorithms.

Chapter 6 presents the potential value of a data-flow analysis to support the ABP programmer. This chapter demonstrates various mistakes that novice programmers can make and shows how data flow analysis can repair some of those errors automatically as well as how it can speed up the rate at which adaptive programs improve.

Chapter 7 concludes by summarizing this work as well as discussing future directions for this programming paradigm.
Chapter 2: Background

This chapter reviews background material for some of the mechanisms used to drive the adaptation process in ABP. For the most part this is a discussion of reinforcement learning and how it can be used to allow adaptive programs to improve.

2.1 Reinforcement Learning

![Diagram of reinforcement learning system](image)

Figure 2.1: The typical arrangement of a system using reinforcement learning.

Reinforcement learning (RL) [70] is a subfield of machine learning within the larger field of artificial intelligence. A typical RL system consists of a simulator and an agent. The simulator collects observations and sensory input from the world and sends this *state* information to the agent. Additionally, the simulator indicates success and failure via a *reward* signal. For its part the agent selects an *action* for the simulator to take (see Figure 2.1). The general goal of RL
is to be able to answer the question: “Given the state we are in, what action should we take to maximize our (expected) reward?” An answer to that question is called a policy and an optimal answer is called an optimal policy.

2.2 Markov Decision Processes

The RL agent models the simulator as a Markov Decision Processes (MDPs). An MDP $M$ consists of the following elements $(S,A,T,R)$.

- $S$ represents a set of states. Each element of this set typically corresponds to some sensory information collected by the learning algorithm. For example, if an RL agent is trying to solve a maze, the current state might consist of the agent’s current location.

- $A$ represents a set of actions that the agent may choose from in each state. In the maze example problem this might correspond to the cardinal directions the agent may take: $N$, $E$, $S$, and $W$.

- $T$ is a transition probability function $T(s'\mid s,a)$, which indicates the probability of transitioning to state $s'$ if the process takes action $a$ in state $s$. This function highlights an important power MDPs allow in their model: just because an agent attempts an action, does not mean it will succeed. Consider the maze example once again. Suppose the RL agent decides to move $S$ from some cell into an open cell, but before the action is executed some other object moves into that cell and only one object can be in that cell. Then the action would fail. This support for actions that can fail fundamentally increases the expressive power of MDPs in problem modeling by supporting a notion of non-determinism. (Situations where actions always succeed are a specialization of this more general model, and can be represented with a transition probability of 1.)
An important property of an MDP is that the probability transition function $T$ depends only on the current state, not all previous states. This is referred to as the Markov Property.

- Finally, a reward function $R(s)$ (sometimes called an objective function) indicates reward received from being in some state $s$. This function is how the agent gauges success or failure. This function may be stochastic. Alternate, but equivalent, definitions of MDPs permit the type of $R$ to be a function of state and action $R(s, a)$ or even to be a function of state, action, and successor state $R(s, a, s')$. Equivalent definitions of MDPs use a cost function and solutions have the goal of minimizing overall cost.

Solving an MDP means that for each state $s$ the system is in being able to choose an action $a$ that leads to the maximum expected sum of rewards. This mapping of a state to a preferred action is called a policy and is a function of type $\pi : S \rightarrow A$. An optimal policy is typically denoted $\pi^*$. The space of policies $\Pi$ is exponential in the number of possible actions and can be intractably large.\(^1\) Hence, we might be content with policies that approximate the theoretical optimal.

Figure 2.2 presents an example MDP to illustrate how they work. At any given state the process can select from multiple actions each of which may lead to different successor states. In that MDP we start out poor and have to choose whether to try and get a job or take up a life of crime. The risky life of crime (bank robbery) may lead to lots of money that we can spend and enjoy (high reward), but will lead to prison (no reward) if it fails. Honest work leads to a smaller payoff, but tends to be steadier. If we have a job, we will most likely not lose it, and even when we have a job, we may turn to a life of crime.

\(^1\)For example, a system with $|S|$ states with $|A|$ actions possible in each state gives rise to a policy space $\Pi$ with $|A|^{S}$ policy functions.
Figure 2.2: An example MDP. The big circles are states the process might be in, the small circles are actions that can be chosen. Upon choosing an action, the MDP transitions to a new state with a certain probability (indicated on the lines from the actions to the successor states).

2.3 Mapping Adaptive Programs to Reinforcement Learning Problems

The RL model fits the problem of solving partial programs very elegantly. Wherever an adaptive program makes a choice, it is basically asking the same “best action” question given above. The state of the MDP can be the state of (a subset of) the program’s memory. In the example in the introduction, the context argument of the choose function represents this. The set of options to choose from passed into the choose function roughly represents the actions in an RL program. The reward function in ABP roughly maps to the notion of reward functions in RL. Thus an adaptive program can be modeled by an MDP.

Another reason RL fits so nicely is that it deals with the problem of delayed reinforcement very well. That is, a program might make a decision early on that is key in reaching some large reward much later on. RL algorithms account for such possibilities and have ways of dealing with these scenarios effectively [70].
An adaptive program in our simple imperative language that models the example MDP in Figure 2.2 might look like the following.

```
    t = 0
    while (t < MAX_TIME)
        t = t + 1
        if state == POOR
            if choose(Poor, [ROB_BANK,GET_JOB]) == ROB_BANK
                state = TryRobbingBank()
            else
                state = TryGettingJob()
        else if state == WEALTHY
            SpendMoney()
            reward(5)
            state = POOR
        else if state == IN_JAIL
            Rot()
        else if state == HAVE_JOB
            if choose(HAVE_JOB, [WORK,ROB_BANK]) == WORK
                state = TryWorking()
            if (state == HAVE_JOB)
                reward(1) // else (we got fired) and state == POOR
            else
                state = TryRobbingBank()
```

This program’s control flow encodes the structure of the example MDP shown. In fact it turns out that, at least in this case, the program maps statically to the MDP (i.e. you can write down the MDP just by looking at the program without it running).\(^2\) Actions are external functions such as `TryWorking` and `TryRobbingBank`. The various `choose` functions correspond to transitions out of various states.

Unfortunately, not all adaptive programs directly generate a full MDP. Consider a path-finding problem on a \(2 \times 2\) grid, which we refer to as the NAVIGATION problem.

\(^2\)This particular adaptive program is interesting in that all the arguments to `choose` are constant values. Generally this is not the case, the state arguments will usually be a variable.
pos = (0,0)
goal = GetGoal();
while (pos != goal)
    pos = Move(pos, choose((pos,goal),[N,E,S,W]))
    reward(1)

The program initially starts at (0,0) and must navigate to some goal. The state consists of the current position paired with the goal. The Move function takes a position tuple and updates it to reflect a move taken in a given direction. While quite simple, this program does not give us an obvious view of what the underlying MDP is. However, the underlying MDP does exist and an approximation is shown and discussed in Figure 2.3.

If the entire MDP is known ahead of time, then methods such as value iteration and policy iteration [70] can typically solve them (find the optimal policy) ahead of time and without even running the program.

In almost all cases, we cannot fully describe the learning model statically. Even in the bank robbery example in Figure 2.2 the transition probabilities are shown in the MDP. However, the example adaptive program for this MDP includes no information to support the numbers given in the MDP figure. For instance, the action TryRobbingBank does not indicate that it will transition the state variable to WEALTHY 70% of the time and to INJAIL 30% of the time as the figure indicates. However, algorithms such as Q-learning [76] or SARSA [61] handle situations where the underlying model is not known up-front. These methods explore and learn the model as they operate.

Adaptive programs are very well represented with MDPs. Program state corresponds to MDP state, choice locations such as choose operations correspond to actions, and feedback corresponds nicely to rewards in MDPs. The fact that programs behave differently over different inputs is modeled by the transition probability component $T$. Moreover, even the notions of

\footnote{Moving into a wall is defined as a no-op.}
solutions map directly. Solutions to MDPs are policies; this maps directly to a program instance (completion). The optimal program instance is represented directly by the optimal policy. This correspondence between ABP and MDPs is illustrated by Table 2.1.

### 2.3.1 Formalizing the Semantics of ABP Programs

This section discusses the relationship between an ABP program and an RL problem. Our work [10] on an object-oriented implementation of ABP formalized this relationship for that language paradigm. The aim of this section is to make a similar mapping for our simple adaptive imperative language to illustrate the basic idea.

Suppose we have an example robot moving around a grid-world maze with the following adaptive program.

```p = InitPos()
g = GoalPos()
f = InitFuel()
while (p != g & & f > 0)
    m = choose((p,g),[N,E,S,W,X]);
    p = Move(p,m)
    f = f - 1
if (p == g)
    reward(f)
```
This program initializes the robot’s position and goal to input values (i.e. the adaptive program cannot statically determine them). It then attempts to move from its current position $p$ to the goal $g$ by selecting from the cardinal directions $\{N, E, S, W\}$ as well as staying in place (represented by $X$). The adaptive program above to solve this problem limits the number of moves to find the goal to whatever value $\text{InitFuel}$ returns. If the robot makes it to the goal successfully, the program specifies the remaining fuel as a reward, hence, more efficient navigation is rewarded. The $\text{Move}$ function applies a move to a position and returns the updated position. An illegal move is treated as a no-op.

The general idea of ABP is to turn adaptive programs into RL problems. It is expected that the program gets to run multiple times to learn better and better solutions. We now formalize what we mean by “better” and “solution”.

A program instance of an adaptive program $P$ follows a fixed policy $\pi$, which is a function $\pi : C \rightarrow A$ (mapping contexts to actions). In the example above, the set of contexts $C$ is all possible pairs of locations on the map. Notice that this is not always something that can be observed statically. The set $A$ is the union over all actions passed to $\text{choose}$ functions. In the above example this is five elements, the four directions and the no-op. The space of all possible policies $\Pi$ is the set of all functions from contexts to actions.

We can evaluate the success of a single program instance of $P$ with policy $\pi$ by considering the total reward it receives on average when executed against various inputs. We formalize this sum of rewards as $R(P, \pi, x)$ where $x$ is a program input drawn from some distribution $X$. For example, the input in the example above would be the initial and goal positions as well as our start fuel. Finally, for now assume that adaptive programs always terminate, hence $R(P, \pi, x)$ is always finite. In practice, this has always been a reasonable assumption. However, straightforward extensions to various RL algorithms we will discuss gracefully deal with the non-termination case [70].
If we knew all possible inputs \( x \) in \( X \) and had a finite set of policies \( \Pi \), we could apply a brute-force approach to solving this problem of finding the best policy. Specifically, we could enumerate each program instance (policy) and test it on all inputs from \( X \). The optimal policy \( \pi^* \) could be defined as:

\[
\pi^* = \arg \max_{\pi \in \Pi} \sum_{x \in X} R(P, \pi, x)
\]

However, this quickly becomes infeasible for two reasons. First, the set \( X \) might be too large (perhaps infinite) to iterate through efficiently. Worse yet, even if we just sample inputs drawn from \( X \) and take an average, the set \( \Pi \) is also usually too large (also possibly infinite) to iterate through efficiently. In fact, with a context space \( C \) and action space \( A \), there are \( |A|^{|C|} \) unique functions. Even a tiny \( 4 \times 4 \) grid in our above example program would generate an enormous space of policies to be tested (\( 5^{16} \)). Moreover, it is also possible that the sets \( C \) and \( A \) are not statically visible, which might further complicate determining the set of possible policies to iterate. Consider the grid world example above, nowhere in the grid world are the maximum dimensions defined. In fact, the initial values and updates to those values all happen externally.

The field of RL has developed many algorithms for learning good policies in these types of situations. One used by ABP is called Q-learning [76, 70]. We discuss how ABP uses it below.

2.3.1.1 Q-Learning

The general idea behind Q-learning is to maintain a Q-function, \( Q(c, a) \), which maps contexts and actions to real value estimates of rewards the process can receive. Output values (elements of the image) of this function (called Q-values) have an intuitive meaning: \( Q(c, a) \) is an estimate of the reward the program will receive for choosing action \( a \) in context \( c \) and then behaving
optimally from then on. Often times it is helpful to visualize a Q-function as a table (called Q-table) of Q-values. Before learning begins, the Q-learning algorithm may start with any Q-function (e.g. all zeros). Then over repeated test executions the function is updated at various time steps via the following rule.

\[
Q(c_t, a_t) \leftarrow (1 - \alpha)Q(c_t, a_t) + \alpha (r_{t+1} + \max_{a'} Q(c_{t+1}, a'))
\]  

(2.2)

The \(\alpha\) constant, usually some small value \(0 < \alpha < 1\), represents the learning rate. The first term is the old estimate and the second term is the new estimate. Hence, this update performs a weighted average update using \((1 - \alpha)\) of the old estimate and \(\alpha\) of the new estimate. Reasonable values for \(\alpha\) used in ABP typically range from 0.01 to 0.1. Schemes to decay this value over time also exist. The second term also mathematically illustrates what a Q-value really is: a reward \(r_{t+1}\) for taking action \(a_t\) in context \(c_t\) and all the rewards the process can anticipate in the future if it behave optimally (the max term). When the program terminates, the Q-learning algorithm uses a slightly different update rule than the one defined above.

\[
Q(c_t, a_t) \leftarrow (1 - \alpha)Q(c_t, a_t) + \alpha (r_{t+1} + 0)
\]  

(2.3)

This is the same rule, except the term representing our future expectation is simply 0 since the program is ending and no future rewards can exist.

The ABP library uses Q-learning as follows. The first call that the library sees is of the form \texttt{choose}(c_0, ...)\). The library records the context \(c_0\) and selects an action \(a_0\) according an exploration strategy that we will discuss later. Next the library observes all calls to \texttt{reward} and keeps a running sum \(r_0\) of those reward arguments. The next call to \texttt{choose} observes new context \(c_1\). The library again selects an action, and the update rule shown above in Equation 2.2 is
executed. In this case, \( i = 0 \) and we have \( a_0, c_0, \) and \( r_0 \) from the previous calls as well as \( c_{i+1} = c_1 \) from the current choose call. The adaptive program continues this way until termination after the \( i^{th} \) choice. Upon termination, the terminal form of the update rule 2.3 is applied.

Consider our robot example above. Suppose the program starts with an initial position \((0,0)\) with a goal of \((1,0)\) and fuel of 10. Initially the library will make a choice at \((0,0)\). Suppose, the sequence of choose calls is \( S,E,N \) at which point the robot has found the goal \((p == g)\) and the loop terminates. At this point the program gets a reward of \(10 - 3 = 7\) and the program exits. Assuming that the table of Q-values is all initially 0’s, then the following updates would have been performed.

<table>
<thead>
<tr>
<th>( i )</th>
<th>( c_i )</th>
<th>Reward</th>
<th>Update</th>
<th>( a_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((0,0),(1,2))</td>
<td>0</td>
<td>No update for first choose</td>
<td>( a_0 = S )</td>
</tr>
<tr>
<td>1</td>
<td>((0,1),(1,2))</td>
<td>0</td>
<td>( Q(c_0,S) = (1 - \alpha)Q(c_0,S) + \alpha(0 + Q(c_1,X)) )</td>
<td>( a_1 = E )</td>
</tr>
<tr>
<td>2</td>
<td>((1,1),(1,2))</td>
<td>0</td>
<td>( Q(c_1,E) = (1 - \alpha)Q(c_1,E) + \alpha(0 + Q(c_2,X)) )</td>
<td>( a_2 = N )</td>
</tr>
</tbody>
</table>

Loop Exits

\( \text{reward}(10 - 3) \)

Program exits: terminal update

<table>
<thead>
<tr>
<th>( i )</th>
<th>( c_i )</th>
<th>Reward</th>
<th>Update</th>
<th>( a_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>N/A</td>
<td>7</td>
<td>( Q(c_2,N) = (1 - \alpha)Q(c_2,N) + \alpha(7) )</td>
<td>N/A</td>
</tr>
</tbody>
</table>

This table illustrates the sequence of Q-updates that occur along with the events that stimulated those updates. The selection of \( Q(c_1,X) \) and \( Q(c_2,X) \) are an arbitrary side effect of the empty table; with all values 0, the optimal action for that context is any of them. In this first run, we can see how the final update is the only one to change any Q-values \( Q(c_2,N) \) becomes \( 7\alpha \). All the rest simply update their old values to 0. However, if one considers successive program
runs, we can see how the reward slowly trickles back. Given the same run, a second time, the update with \( i = 2 \) would detect \( N \) to be the optimal choice and the update for that step would be
\[
Q(c_1, E) = (1 - \alpha)Q(c_1, E) + \alpha(0 + Q(c_2, N)).
\]
The value of \( 7\alpha \) is propagated back. If we execute the same program and pick the same values once more, we would propagate a non-zero value back to the update of \( Q(c_0, S) \).

2.3.1.2 Exploration — Selecting an Action

During a choice of the form \( \text{choose}(c, [a_1, a_2, \ldots]) \), the ABP library must select a value from the set of options passed in. Given the context \( c \) the algorithm can look up all estimates \( Q(c, a_1), Q(c, a_2), \) and so forth so far amongst those actions and can select the best or “greedy” option that indicates the best chance for the most reward. However, the learning algorithm must balance this exploitation strategy with the need to explore new options, which might lead to better solutions. One approach that works well is an \( \varepsilon \)-greedy strategy, which chooses the best known “greedy” option with probability \( 1 - \varepsilon \) and tries a random option with probability \( \varepsilon \). Reasonable values for \( \varepsilon \) might be 0.1 or even higher such as 0.3.

As the adaptive program is run on many example inputs from \( X \), the Q-values slowly adjust towards more and more accurate estimates. At any point we can extract a policy from the adaptive program’s Q-table by simply iterating the set of contexts \( c \in C \) and for each defining
\[
\pi(c) = \arg \max_a Q(c, a).
\]

When evaluating an adaptive program instance, the system will have some means for disabling the learning and exploration components of the learning framework, and instead, always choosing the best known actions for each context. For example, this simple imperative language, this process could be accomplished by defining a different interpreter.
2.4 State Abstraction

Recall how program memory corresponds to state in an MDP. The obvious problem is that program memory is very large and most parts of it are not relevant to the choice. State abstraction can be thought of as the process of deciding which subset of program variables influence the underlying learning problem. All partial programming systems have some facility for this task of projecting out the useful features in the program state for the learning algorithm. Hence, it is worth a brief discussion of this subject.

As an example, suppose a robot is at \((x, y)\) and is attempting to go to goal \((g_x, g_y)\) and must choose a direction to take. If the robot’s internal timer \(t\) is also available, but does not affect our route or goal, then it makes sense to abstract away the \(t\) from consideration when selecting a direction to take. Otherwise the RL algorithms will treat its navigation choices from the same point to be different if performed at different times even though they are the same. Given enough time and trials the optimal solution can still be found; it just might take a lot longer since the underlying MDP is potentially much larger.

Conversely, the cost of failing to include relevant state information is equally dangerous and results in less effective solutions. From the previous example, suppose the time variable does influence our robot’s ability to move (the robot is solar powered and can only move in daylight, moves during the night fail). Under these conditions, the learning algorithm is learning optimal behavior that merges two very different scenarios. Perhaps the optimal behavior should be, stay in place during the night and wait for daylight to move.

Almost all the approaches to partial programming covered in the next chapter as well as some of those in ABP require the user to explicitly make this computation.
Figure 2.3: An MDP for the NAVIGATION problem. Each subgraph corresponds to a different goal (chosen at run time).
Chapter 3: Literature Review

There has been considerable work in the past couple decades on systems to support partial programming languages and tools. This chapter surveys a few of those approaches that are most similar to the approaches that ABP languages take. In Section 3.1 we discuss a vein of work from the field of reinforcement learning, which culminated in an adaptive variant of the LISP programming language. Next in Section 3.2 we consider a related approach to partial programming through a specialized adaptive behavior language called A²BL in which users are able to define adaptive behaviors for agent controllers. The system then uses a mixture of reinforcement learning techniques find good completions for the behavior. A third related approach called Program Sketching is covered in Section 3.3. This system for partial programming is notable as it uses a significantly different method for presenting and adapting programs. In Section 3.4 we discuss systems used to automatically configuration software across different hardware and platform configurations with the aim of minimizing the execution time for the configured program.

3.1 ALISP-Based Approaches

One of the first and most closely related approaches to ABP comes directly from the field of reinforcement learning and initially invigorated the definition and study of partial programming. The major goal of these approaches was to constrain reinforcement learning problems in such a way that larger problems could be efficiently solved.

Initially this work started by Parr et al. [55] was a method for specifying a set of constraints on RL problems through a hierarchy of specialized state machines (HAM). Later work [3] ex-
tended the expressiveness of the machines by adding useful constructs so that complicated problem constraints could be specified more concisely. Shortly thereafter the authors showed how the same constraint specifications could be given as programs in a specialization of the LISP language, which was called ALISP [4, 2].

The HAM/ALISP work breaks the learning problem into two parts. The first $M$ is an MDP, which defines the basic structure of the problem to be solved including relevant state and reward information. In theory this component is sufficient definition to solve the problem with standard RL methods; however, such MDPs can easily be too large to practically solve. The second piece $H$ is a hierarchy of machines (in the HAM and PHAM work) or an ALISP program (in that approach) that defines a set of constraints on $M$. The goal of $H$ is to constrain the learning problem defined by $M$ to the point that it can more easily be solved. The various approaches focus on exploring different ways of defining the $H$ component.

One of the most important types of constraint in this work is hierarchy. Hence, a major defining characteristic for this work is in its use of hierarchic reinforcement learning (HRL) [24]. HRL algorithms work well on large learning problems that naturally decompose into smaller learning problems, much like those in Brooks-style (or subsumptive) architectures [17]. For example, selecting a route to take is a higher-level problem than navigating some given route. All the examples in this work have some sort of hierarchic aspect that can be exploited.

We next discuss the family of ALISP work in more detail.

3.1.1 HAM — Hierarchies of Abstract Machines

Parr and Russell [55] provided the initial work in this field with a constraint language that coupled a learning problem, defined as an MDP $M$, with an extra set of constraints $H$ on that problem. The extra set of constraints coupled with the MDP can be thought of as an adaptive pro-
gram of sorts. The MDP $M$ is sufficient to define the learning problem and could theoretically be solved given enough computing power. However, it is easy to conceive of intractably large MDPs, hence the additional constraints are necessary to limit the search activity of the learning algorithm.

A HAM (Hierarchy of Abstract Machines) $H$ can be composed with an MDP $M$ to create a new MDP $H \circ M$ with the same optimal policies, but one that can be solved much quicker than the original $M$. The $H$ component of the composition speeds up the search in $M$ by constraining the set of policies the learning algorithm must consider.

A HAM $H$ can be thought of as a set of state machines where each element machine consists of the following types of states.

- Action: this state type executes an action in the external MDP $M$. This calls into $M$ and signals a state transition. This is the primary way $H$ interacts with $M$. It tells $M$ that some action just occurred and $M$ is responsible for updating in response to that action. For example, if $H$ indicates action SHOOT, then it is assumed that $M$ will decrease the variable representing available ammunition by 1 ($M$ will transition into a state representing available ammunition decreased by 1).

- Choice: indicates a point of non-determinism and specifies a set of possible states. This is the key adaptive construct within their language and is how they explicitly express uncertainty in their partial programs.

- Call: executes another machine as a subroutine. A call state pushes the active machine onto a stack and transfers control to the start state of the HAM being called. This construct is necessary to support hierarchical RL algorithms and is qualitatively similar to a procedure call in a programming language.
- Stop: stops execution in the current HAM and returns control to the previous HAM. Much like a return statement in a programming language, transitioning into this type of state will return control to the last machine that entered a call state.

The main contribution is a formal algorithm for how \( H \circ M \) can be converted into a new MDP with the same optimal policies as \( M \). A HAMQ-Learning algorithm is also presented that can learn \( H \circ M \) when \( M \) lacks an explicit definition, much in the same way Q-Learning can work in MDPs whose structures are not known a priori.

![Figure 3.1](image)

Figure 3.1: The example problem from Parr and Russell’s work [55] (figure copied). The entire grid maze (a), a zoomed in view of an obstacle within that maze (b), and one of the machines in the hierarchy used for constraining this problem (c).

The example presented in this work is a path-finding problem (reproduced in Figure 3.1). A robot is placed in a maze of fairly wide corridors (several cells wide). However, the corridors contain various regular-shaped obstacles that must be passed. The problem has two levels, navigating around objects within any given corridor is a lower-level learning problem, while choosing which corridor to take is a higher-level problem. Hence, this problem fits the hierarchic RL model nicely.
In fact, the maze figure can be considered a graphical representation of the MDP being solved. For instance, the set of states $S$ in the MDP might just be the set of a cell locations that the robot can be in (around 3600 for this problem). The set of actions in the MDP is implicitly represented as the directions the robot can move (i.e. N,E,S,W). The transition probability matrix $T$ for this problem is given separately: in any given cell the robot moves successfully to the desired neighbor cells 80% of the time and fails 20% of the time, instead moving in an unintended direction. The reward function $R$ is 0 for all states except the bottom right (goal), absorbing (final) state, which gives a reward of 5.

Given sufficient computing power the MDP described above could be solved directly and no additional information would be necessary. However, the problem detailed is sufficiently complicated to thwart such direct methods. Hence, the authors make this problem tractable by defining an accompanying hierarchical abstract machine $H$ to constrain the problem.

The intuition for why this method works is that while the state space of the MDP $M$ is very large the state space for the HAM $H$ is much smaller. Transitions in the composed machine $H \circ M$ must move in parallel, and the smaller HAM state-space effectively constrains the activity of the composite MDP. Strategic decisions are indicated in the HAM via choice states. So rather than every state transition being a point where something has to be learned only a smaller set of choices must be learned.

The machine hierarchy for this particular problem consists of an initial machine that selects between two sub-machines, which represent the heuristics “follow wall” and “back off from obstacle”. This top-level machine uses a choice state to choose between those two sub-machines. This part of the machine is similar to an ABP program choosing between two algorithms to accomplish some task.

To sum this approach up, the MDP $M$ ((a) and (b) in the Figure 3.1 above) sufficiently describe the learning problem being solved and how various actions the program can take will
affect the world. The hierarchy of machines $H$ (one level shown in (c) of Figure 3.1 above) provides extra constraint information that indicates a “partial description of desired behaviors” and more effectively focuses the learning. HAM is a language for a set of rules that constrain an MDP.

3.1.2 PHAM — Programmable Reinforcement Learning Agents

Andre extended Parr and Russell’s work on HAMs in [3] by adding more state types into the hierarchy that mimicked patterns found in general-purpose programming languages such as variables, signals, and a few others. These extended “Programmable-HAMs” are referred to as PHAMs.

The primary contributions of this work was not in solving larger and harder learning problems, but in more concisely describing the constraints (i.e. the machine hierarchy component $H$). In all cases an equivalent HAM could be given for any PHAM, the PHAMs are just significantly smaller.

The most important additions that make PHAM more expressive than HAM are as follows.

- Machine parameterization supports the idea of a machine templates. For example, the paper gives the example of machine behaviors such as `WalkNorth`, `WalkEast`, `WalkSouth`, and `WalkWest` that may be combined into a parameterized `Walk(d)` where $d$ is a direction parameter. Parameterization reduces the complexity of the machine hierarchy since a machine template needs to be specified once instead of multiple times.

- Memory variables are also introduced in PHAM. A new type of state called a *set-variable* is included in PHAM. These variables are then used by machine templates. A PHAM might set a variable to indicate that it is trying to progress to a location $p$. When it invokes
a templated sub-machine by entering a call state it parameterizes the submachine by the value of that variable. Continuing our example above, a state might set a direction variable $d$ to be North, then when entering the call state for the Walk($d$) machine, it would pass that variable, thus instantiating the machine WalkNorth.

- Aborts and interrupts are explicitly supported in PHAM. The use case for an interrupt is typically when the machine detects a more important task that must happen while performing a less important task. If it is necessary to resume the less important task when done, an interrupt would be appropriate since it resumes control where it was interrupted. An abort state is like an interrupt state except control does not return to the interrupt point, but instead transitions out of the abort state at that higher level. An interrupt could be modeled in HAM by specifying Call state to the interrupt state linked to every state in the HAM.

The underlying composite MDP contains the same number of choices that must be learned as in the HAM approach [3, Sec.5]. Hence, the major motivation of this newer approach is not one of program efficiency or space usage, but rather one of improved expressibility in partial programs. In fact, the authors compare their work to the previous approach with Parr’s HAMs [55] and show similar performance in program learning rate. However, they show comparable programs in both languages. The example from Parr’s work on HAMs discussed above requires a hierarchy of 37 machines versus the approach with PHAMs, which required only 7, and the problem presented in the PHAM work required only 9 machines to the 63 required by the HAM approach. In short, this can be viewed as a first attempt to offer a “higher-level” language for reinforcement learning problem constraints.
3.1.3 ALISP

The observation that adding additional constructs to HAM made it look more like a programming language naturally progressed into ALISP [4, 2], an integration of the previous concepts from PHAM in an existing programming language (LISP [69]). In fact, one can think of ALISP as a method for quickly specifying PHAMs.

ALISP is essentially LISP with the addition of three major constructs (implemented as macros) each of which corresponds directly to constructs in HAMs.

- The macro \(\text{choice label form}_0 \text{ form}_1 \ldots\) represents a choice point within the partial program. This construct fulfills the same role as the choice states in HAMs or PHAMs, and specifies uncertainty or non-determinism in a program. The choice function is roughly similar to the suggest method in our Java ABP in Chapter 5 or the choose operation used in the simple imperative ABP language from earlier chapters.

- The second macro \(\text{call subroutine arg}_0 \text{ arg}_1 \ldots\) is also identical in function to call states in HAMs or PHAMs and exists to support decomposition of hierarchic learning problems. Moreover, it obviates the need for the return (stop) states in HAMs since the subroutine boundary implies that information automatically. Since ABP has no support for hierarchic RL, there is no corresponding construct for this.

- The final construct has the form \(\text{action name}\). This construct specifies an action to be taken in the external environment MDP \(M\) and replaces the notion of an action state in HAMs. This construct may be paired with additional information that indicates which state variables are relevant to it. We discuss this state abstraction component later.

In contrast, since ABP does not separate the state and reward components from the partial program, it has no need for a similar construct.
The ALISP work uses an example to illustrate the language involving a taxi moving about a grid world picking up and delivering passengers at various locations, which is reproduced below from [4].

(defun root () (if (not (have-pass)) (get)) (put))
(defun get () (choice get-choice
    (action load)
    (call navigate (pickup))))
(defun put () (choice put-choice
    (action unload)
    (call navigate (dest))))
(defun navigate(t)
    (loop until (at t) do
        (choice nav (action N)
            (action E)
            (action S)
            (action W))))

Figure 3.2: Taxi World (fromAndre’s ALISP paper [4]).

Figure 3.2 illustrates a graphical representation of the MDP for world. It consists of the 5 × 5 grid shown. There are several state variables: x and y keep track of the taxi’s location, and pickup and dest correspond to the location of the passenger and where they are to be dropped off (the squares labeled by one of: R, G, B, and Y in the figure).
The root-level function `root` decides whether or not to load a passenger and either calls `get` or `put` to accomplish this. Note that these calls are function invocations within the LISP language and not the special `call` macro that indicates a sub-learning problem is to be solved. The `get` and `put` functions are similar, depending on the `pickup` (one of RGBY) or `dest`, they decide between loading or unloading the passenger and navigating to the named location. Finally, the `navigate` routine continually moves the taxi around the grid until the it lands on a target location (detected via the `at` function).

In ABP choice functions (such as `choose` in our example adaptive language) take an explicit recognition context argument which corresponds to the state in the underlying MDP. In the ALISP (and HAM and PHAM) the state is globally defined. For instance, in the taxi example above, the state was explicitly defined as a set of features in the form of four global variables: `x`, `y`, `pickup`, and `dest` as well as any relevant local variables. By default transitions in the MDP assume that all variables contribute to decisions being made. However, as we already discussed in the section on state abstraction this is not always the case. To address this lack of state abstraction, the authors extended the `action` macro to optionally take an extra argument, which lists that dependency information.

For example, an action from the taxi domain (action `N`) must expand to a more complicated form as shown below.

```
((action N)
 :reward-depends-on nil
 :completion-depends-on (x y t)
 :external-depends-on (pickup dest))
```

Without getting into the gritty details and equations, suffice it to say that the three annotation components to this are a requirement of the implementation of hierarchical reinforcement learning used. Consequently, to make effective use of this construct, the programmer must understand the details of the specific hierarchic learning algorithm used.
In contrast, ABP does not make use of hierarchic RL. While there is no technical reason why ABP could not support hierarchic RL, it does keep the learning algorithm from leaking from the abstraction a bit better. The extra complexity in the state abstraction component in the above example illustrates why that might be beneficial. Instead, in ABP a context value is explicitly constructed and passed to the choice function (\texttt{choose} in our simple language), which indicates the relevant parts of program memory that might influence the current choice.

A second difference between ABP systems and ALISP programs has to do with the lack of an explicit reward statement in ALISP. Recall, their work started as a constraint system for an independent MDP. It made sense to leave details involving state and reward behavior in the accompanying MDP $M$ and limiting the constraint system $H$ (HAM or ALISP) to learning problem structure. That philosophy never changed as this work progressed, though with the addition of state abstraction in ALISP some of $M$’s state necessarily began leaking into $H$. In contrast ABP makes the reward explicitly part of the adaptive program.

Another difference between ALISP (and HAM) and ABP is in their opposite views of how an adaptive program should be “driven”. ALISP and HAM abstract away all state modification rules within the external MDP $M$. $H$ drives behavior in $M$ with an action construct. In ABP the adaptive program itself encompasses both parts, including a formal notion of reward and state within the partial program itself.

One can think of the ALISP-based work as two separate modules, the partial program $H$ and the external environment $M$. For example, if the $H$ component decides to effect the action \texttt{MoveNorth}, $M$ will be notified (via the action construct) and must carry out the action updating state variables (such as changing the position variables) and assess any rewards (for example if the taxi reaches its destination). This split forces a decoupling point and separation of concerns on the programmer that might not be beneficial; specifically since $H$ heavily depends on the internal structure within $M$ in many ways.
Figure 3.3: This figure illustrates the differences between HAM/ALISP and ABP. ALISP splits state management and reward attribution off into an external module (MDP), but drives program flow and state transition with actions. However, its structure depends heavily on those states and rewards within $M$ (the dotted line). ABP merges these components and makes this dependency explicit by passing a state to the algorithm and receiving a suggested action to take in return.

- Control flow in $H$ depends on state in $M$. For example, consider how the `navigate` function in the taxi program makes use of `at`, which must interact with $M$ or the `have-pass` function.

- $H$ drives $M$ via the `action` construct; however, this implicitly assumes that a given action is legal or makes sense for the given state within $M$.

- The state abstraction component depends on how rewards are assessed in $M$ [4, page 4].

- The state abstraction component explicitly references state variables maintained by $M$. In fact, later versions of ALISP rectify this by adding a `get-state` function to the language for this purpose [48].
Part of this forced decoupling between \( H \) and \( M \) is due to the fact that HAM, PHAM, and ALISP are just constraint languages to specify \( H \). They are not theoretically necessary for solving the underlying problem represented by \( M \).

ALISP programs only have constructs to formally define the \( H \) component. It is assumed that \( M \) is an MDP, but there is no programming language constructs specifically to define that piece or its interaction with the learning algorithm.

In contrast to ALISP and HAM, ABP merges the functionality of \( H \) and \( M \) since they are already highly coupled. The result can still be considered an MDP, and there are explicit constructs for each component of the underlying MDP. This gives adaptive programs in ABP a more formal semantics in relating program to learning problem.

- The \texttt{choose} function takes a context argument, which corresponds to the current state within the MDP. In languages with static types the typing of this argument translates to a notion of the set type of \( S \) within the MDP. This approach also obviates any need to address the state abstraction problem as in ALISP.

- The \texttt{reward} function can be placed anywhere in the partial program (multiple instances) and translates into the MDP’s reward function \( R \).

- The transition probabilities \( T \) and model for the MDP are something learned by the algorithm, but in some cases can be inferred from program structure.

Some of the above differences discussed in the previous section are illustrated in Figure 3.3.

3.1.4 Concurrent ALISP

Following Andre’s work in ALISP, Marthi et al. introduced Concurrent ALISP [49, 48]. The main additions in this work were constructs to support efficient learning in problems with an RL
agent executing logically parallel threads (tasks). For instance, consider extending the previous taxi problem with multiple taxis where no two can share the same cell. In this case, the optimal behavior learned by taxis independent of one another does not benefit the group (since they obstruct one another). Additionally, the state and action spaces are intractably large for such problems (e.g. $K$ taxis each choosing one of four directions implies $4^K$ possible joint actions at each step).

Concurrent ALISP rectifies this by allowing multiple agents to coordinate their activity (and not interfere with each other) by making joint choices. This language introduces several constructs to allow inter-task communication as well as task definition and execution. The run-time allows threads to run in parallel, but synchronizes them at choice or action constructs. If one or more thread is at a choice, a joint choice is made for all threads. If all threads are synchronized at an action, then a joint action is executed within the external environment $M$.

The state space explosion is dealt with by modeling the domain of features (state) as real values and applying a linear function approximation approach [70].

The choice selection problem is mitigated by the fact that there are rarely more than a few tasks making a joint choice. Hence, even with $K$ taxis presumably only a few need to be part of a joint choice. Additionally, this work presents a method, which effectively breaks up some of the joint choices into smaller joint choices between potentially interfering pairs.

Much of the comparison between the original work in ALISP work and ABP applies to this extension of ALISP. The additional Concurrent RL features are not something ABP ventures to scale to currently.

---

1Features in the model (variables in the relevant problem state) are represented with real values in a feature vector. The learning algorithm finds an optimal weight for coefficients of these vector components.
3.2 Adaptive Agent Behavior Languages

Agent Behavior Language (ABL) [51] is a reactive planning language based on an earlier planning language called Hap [46]. Reactive planning languages have the goal of efficiently specifying believable agents for interactive environments. In ABL programmers specify complex agent behaviors through a tree-hierarchy of parallel and sequential sub-behaviors. Behaviors have preconditions constraining their application, and various sub-actions change the state of the world when successfully applied.

The Adaptive Agent Behavior Language (A^2BL) grew out of this in work by Simpkins et al. [64] and Bhat et al. [14]. The key observation being that instead of explicitly defining behaviors through complex preconditions one could leverage reinforcement learning to adaptively select behaviors given just a few details.

The example from the A^2BL papers presents the problem of defining a behavior for a “furry creature” living on a grid world. The creature has two goals, finding food that randomly appears, while also avoiding a slower-moving predator. The general outline of an ABL program (not A^2BL) for this problem is given below.
behaving_entity FurryCreature
{
    parallel behavior LiveLongProsper() {
        subgoal FindFood();
        subgoal AvoidPredator();
    }
    sequential behavior FindFood() {
        with (ignore_failure) subgoal MoveNorthForFood();
        with (ignore_failure) subgoal MoveSouthForFood();
        with (ignore_failure) subgoal MoveEastForFood();
        with (ignore_failure) subgoal MoveWestForFood();
    }
    sequential behavior AvoidPredator() {
        with (ignore_failure) subgoal MoveNorthAwayFromPredator();
        ... similarly for South/East/West
    }
    sequential behavior MoveNorthForFood() {
        precondition {
            (FoodWME x::foodX y::foodY)
            (SelfWME x::myX y::myY)
            ((foodY - myY) > 0) // The food is north of me
        }
        // Code for moving agent to the north elided
    }
    ... similarly for South/East/West
    sequential behavior MoveNorthAwayFromPredator() {
        precondition {
            (PredatorWME x::predX y::predY)
            (SelfWME x::myX y::myY)
            (moveNorthIsFarther(myX,myY,predX,predY))
        }
        // Code for moving agent to the north elided
    }
    ... similarly for South/East/West
}
The top-level parallel behavior LiveLongProsper groups the two parallel goals of the agent, find food and avoid the predator. The FindFood sub-behavior is an example of a sequential behavior and encodes the furry creature’s desire to find food while the AvoidPredator behavior encodes that desire. The parallel keyword indicates that these two sub-behaviors are executed at the same time.

Behaviors marked as sequential such as FindFood attempt to execute a sequence of subgoals until one fails. If a failure is encountered, the enclosing behavior also fails propagating the failure upwards. In FindFood the preconditions of the subgoals are mutually exclusive, for instance, one cannot be both north and south of the food at any given moment. Hence, some of the subgoals will fail. The with (ignore failure) modifier on each subgoal suppresses any failure so that FindFood will try all subgoals and indicate success itself regardless of the outcome of child behaviors.

Finally, the MoveNorthForFood behavior will execute some statements (in Java) if the precondition clause is specified. The precondition clause consists of two lines that bind variables to working memory elements (WMEs), which roughly correspond to a world state, and the expression on the final line of the precondition block specifies the precondition logic as a function of those variables.

The authors of A²BL made the observation that behaviors in ABL programs could be learned instead of requiring explicit definitions. The previous ABL program could be realized as an adaptive version in A²BL as shown below.
behaving_entity FurryCreature
{
    adaptive collection behavior LiveLongProsper() {
        subgoal FindFood();
        subgoal AvoidPredator();
    }
    adaptive sequential behavior FindFood() {
        reward { 100 if (FoodWME) }
        state {
            (FoodWME x::foodX y::foodY)
            (SelfWME x::myX y::myY)
            return (myX,myY,foodX,foodY)
        }
        subgoal MoveNorth();
        subgoal MoveSouth();
        subgoal MoveEast();
        subgoal MoveWest();
    }
    adaptive sequential behavior AvoidPredator() {
        reward { -10 if (PredatorWME) }
        state {
            (FoodWME x::predX y::predY)
            (SelfWME x::myX y::myY)
            return (myX,myY,predX,predY)
        }
        subgoal MoveNorth();
        subgoal MoveSouth();
        subgoal MoveEast();
        subgoal MoveWest();
    }
    // ...
}
To make ABL adaptive A²BL includes several adaptive constructs such as the state and reward clauses as well as a special adaptive modifier on behaviors. Additionally, there is a natural mapping for sequential and parallel behaviors to adaptive variants.

A²BL supports explicit state-abstraction, the state block specifies the subset of global state that influences a decision being made. For instance, the decision in the FindFood behavior does not depend on the predator’s location so that WME is not included as part of the relevant state. The reward clause specifies a reward to be given to the learning algorithm for certain state conditions. For instance, in the example above the reward clause in FindFood rewards the creature 100 if it has landed on food.

The adaptive keyword has special meaning. In behaviors marked as adaptive sequential such as AvoidPredator A²BL will learn an optimal policy for selecting a subgoal (an optimal subgoal for each state). Hence, these behaviors roughly correspond to a choose operation in ABP or choice in ALISP. For behaviors marked adaptive collection, the sub-behaviors are learned in parallel (modeling the parallel construct in ABL) using algorithms from the field of modular reinforcement learning (MRL).

Modular reinforcement learning is a sub-field of RL interested in situations where an agent has sub-modules (sub-agents) with distinct goals (sometimes opposing). The agents have distinctive state spaces and reward streams, but must jointly choose one action. Typically an arbitrator merges the desired actions of the sub-agents in some way such as fusion (e.g. averaging the actions chosen by each agent) or weighting the concerns of each agent in some way and choosing the best option for the group. Different approaches for this have been surveyed [37], and variations on classic RL algorithms exist for this problem such as Feudal Q-learning [23] and, more recently, GM-SARSA(0) [68].

MRL fits the above problem in the following way. The LiveLongProsper parallel behavior can only take one action each time step. However, the sub-behaviors FindFood and
AvoidPredator have different aims and may indicate conflicting behaviors that must be reconciled in some way. A\(^2\)BL uses a general arbitration algorithm developed by Bhat et al. \cite{14} which reportedly works fairly well for their purposes.

A final construct in this language that is worth mentioning is the `success` condition. This condition can be included in adaptive behaviors to indicate that the behavior’s goal has been achieved. The feature is a technical necessity to delimit the end of a learning `episode` for the underlying learning algorithm. For instance, if a behavior was searching for something and found it, it would indicate success with this condition. This helps inform the learning algorithm that later rewards have nothing to do with any choices made before this point in the program. Our ABP implementation in Chapter 5 defines a similar `endEpisode` construct for the same purpose.

A\(^2\)BL is an application of concepts from partial programming to a language in a specialized domain (behavior languages). Like ABP there are `choices` to be made with the goal of maximizing `reward`. In addition A\(^2\)BL necessarily must also use techniques from the field of modular reinforcement learning since their problems tend to be structured that way. The problems listed above could easily be encoded as ABP programs and we illustrate slightly similar problems to the FurryCreature problem in chapter 5.

### 3.3 Program Sketching

*Program sketching* is a technique presented in work by Solar-Lezama \cite{65, 66, 67}.

A Program Sketch is a high-level outline of an algorithm, but one which may contain syntactic `holes` to be filled. In each of these holes a programmer specifies a finite number of program fragment alternatives. In addition, the programmer also specifies assertions to indicate correct-
ness. These assertions guide a synthesis process, which searches for a program completion satisfying all assertions for a set of given inputs.

Program synthesis [32] is a subfield of programming language research interested in discovering programs that satisfy some user intent. At a high-level the motivation for this work is the argument that, in general, it is easier to certify a solution than it is to generate one. Some applications of program synthesis are: discovery of new algorithms [8, 50], automating repetitive programming tasks for end-users [22, 43], general-purpose programming assistance (figuring out mundane, but perhaps tricky details of algorithms) [40], test cases [18], specifications [19], or even automated program debugging [77, 38].

Just as the ALISP work started as a system to constrain hierarchical reinforcement learning problems so that more complex problems could be solved, one could argue that Program Sketching is also a method of constraining a specification for a program synthesis problem.

An example given in Solar-Lezama’s thesis that illustrates Program Sketching nicely is the task of writing an algorithm to reverse a linked list iteratively (they say “efficiently”). We reproduce this below directly from the author’s work [65].
#define LOC { | (l | nl).(head | tail)(.next)? | null | }
#define LOC2 { | LOC | tmp |}
#define LHS { | (l | nl).(head)(.next)? | nl.tail | tmp |}
#define COMP { | LOC ( == | != ) LOC |}

list reverseEfficient(list l){
    list nl = new list();
    node tmp = null;
    bit c = COMP;
    while (c) {
        if (COMP) { LHS = LOC2; }
        if (COMP) { LHS = LOC2; }
        if (COMP) { LHS = LOC2; }
        if (COMP) { LHS = LOC2; }
        if (COMP) { LHS = LOC2; }
        c = COMP;
    }
    return nl;
}

This system can be viewed as a sort of non-deterministic macro expansion for C-style language program. The #define lines form grammars for fragments of program syntax thus allowing the programmer a means for specifying a set of possible syntactic arguments for each program hole. Each non-terminal in these small grammars can then be used in regular program code to specify points of non-determinism. For instance, the #define LOC line means that instances of that generator can expand to any string generated by that grammar, that is one of: {l.head, l.head.next, l.tail, l.tail.next, nl.head, ..., null}. Each use of a generator may expand to a different string.

For the above example, the author argues for each use of a generator found in this Sketch. For example, they assume there has to be a loop, that its start and exit conditions might be
different (hence the two assignments to the bit\textsuperscript{2} c. Moreover, the author inserts five different conditionally executed statements (the five if (COMP) ... statements) to allow the synthesis system plenty of opportunity to adapt. The decision to use 5 was arbitrary, if a solution is found using only two, the rest can synthesize code that is effectively idempotent (and which can be optimized).

To adapt the system the calling function might execute at the top-level the following program.

```c
main(bit[N] elems, int n){
    if (n < N) {
        // create an n element list from the input bit-vector.
        list l1 = populate(elems, n);
        list l2 = populate(elems, n);
        l1 = reverse(l1);
        l2 = reverseEfficient(l2);
        assert compare( l1, l2) ;
    }
}
```

For the above example, the author assumes a recursive list reversal algorithm (called reverse) is easy to implement and may be used as a reference implementation for the assertion statement. The program synthesizer then attempts to find a version of reverseEfficient that satisfies the assertion statement (i.e. one that matches the reference implementation).

The program synthesis is based on a method called CEGIS for counterexample-guided inductive synthesis. The validation procedure starts with a random solution initially. After a failed correctness assertion, the synthesis algorithm will adjust to only generate programs that satisfy that assertion. In other words, for each incorrect program, the synthesizer refines its solution space by adding constraints to omit classes of programs that failed on previous cases.

\textsuperscript{2}A bit in their language is a boolean value.
The authors observe that, in practice, only a few examples are needed to solve most sketches. Hence, the approach appears to be efficient in practice. Moreover, the work illustrates the approach on quite a few non-trivial problems including examples involving concurrency.

One disillusioning attribute of this approach is that, even in simple examples, the assertion logic ends up being nearly as complicated as any hand-crafted solution. For instance, in the iterative linked list reversal example the assertion requires a recursive implementation. In addition, the correctness of the generated solution is dependent on the correctness and completeness of the assertion and program inputs. Had the recursive linked list program failed to handle empty lists, there would be no guarantee that the synthesized solution would have adapted for to handle that case. In fairness this latter point is a price to be paid in all partial programming; if one specifies an incorrect reward, the system will generate a suboptimal or incorrect program.

Another difficulty with this system is in the boolean aspect of using correctness assertions as a feedback mechanism. A Sketch is either correct or incorrect. This does not allow us to easily model situations where feedback is a relative fitness signal (such as the time taken or length of a path computed). Program Sketching partitions the space of program instances between correct and incorrect instances; ABP (or methods using RL) order the space of program instances by affixing an expected reward to each program instance. One might view this boolean-feedback approach as a fundamental limitation, but that trade-off is necessary to leverage program synthesis methods such as the CEGIS method described above.

Another notable difference between ABP and Program Sketching is how the program instances are generated. Program Sketching syntactically expands fragments much like a macro system at program build time (or start time). In contrast ABP uses a fixed program, but permits its variables to adjust automatically at run time. This has a significant effect on the types of problems that each approach excels at (as we will soon see).
The feedback signal (in ABP terms) is an assertion of correctness in this partial programming model. In ABP, this signal may be a fitness value that simply indicates a “better” solution rather than a correct or incorrect solution. As such ABP cannot use the SAT-based inductive synthesis methods that Program Sketching uses, yet ABP can define orderings on candidate program instances and thus support problems where the feedback is not a discrete “correct” or “incorrect” signal.

ABP can emulate Program Sketching and in some cases vice-versa. For simplicity, consider an implementation of ABP with a choose function and reward function. Sketching makes its selection of choices effectively at program start time, that is, each of the generators in the program is expanded at that time. To emulate this ABP must make its choose calls at the start of the algorithm. Secondly, since none of our implementations allow syntactic expansion in the form of grammars, we have to emulate each argument as an integer index for each of the strings from the COMP generator, and then looks up the value of the selected expression at the point it is used. For instance, consider the signal line at the end of the while loop of the previous Program Sketch.

```c
  c = COMP;
```

This can be emulated by expanding all possible syntactic strings generated by COMP and indexing each in some scheme such that we can map it back to the expression’s value. At the top of the function (program), we would include the following code.

```c
  // COMP: generates 19 syntactic fragments:
  // 0 -> l.head == l.head
  // 1 -> l.head.next == l.head
  // ...
  // 19 -> null != null
  c_exit_loop = choose([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18])
```
That is we identify each of the 19 (in this case) expansions that the COMP generator can generate and uniquely identify each. Then we replace the \( c = \text{COMP} \) statement with one that look up the chosen generated expression as a function of the current run-time values.

```c
// c = COMP
  c = lookup_COMP(c_exit_loop, 1, nl)
```

where

```c
boolean lookup_COMP(chc,l,nl) {
  if (chc == 0)
    return (l.head == l.head)
  else if (chc == 1)
    return (l.head.next == l.head)
  ...
}
```

A key observation in this transformation is that we make the selection of syntax to use at program start time in order to emulate the static selection of syntax that Program Sketching uses, but lookup the value of the chosen expression dynamically each time it is encountered. Finally, correctness can be rewarded with any positive reward in lieu of an assertion.

Another key difference between Program Sketching and other approaches, including ABP, is the lack of program state or context in Sketches. Expansion decisions for generator expressions are made statically, hence there is no direct way to support context as in ABP. For example, consider a path finding problem, a robot at \((x,y)\) trying to get to a goal \((gx,gy)\). An ABP-style language could write something akin to the following statement within a loop.

```c
while (notAtGoal(x,h,gx,gy))
  dir = choose([x,y,gx,gy],[N,E,S,W])
  (x,y) = move(dir,(x,y))
```

In ABP, the above choice can vary upon successive invocations. This allows one choice point to encode a tremendous amount of adaptivity.
Contrast this with the structurally similar Program Sketch given below.

```c
#define DIR { N E S W }
while (notAtGoal(x,h,gx,gy))
    dir = DIR
    (x,y) = move(dir,(x,y))
```

The above sketch can only express solutions for problems where the location and goal are on the same row or column. Said differently, this Sketch only defines a space of completions that include constant movement along a single row or column.

Defining a solution to emulate the ABP approach with a Program Sketch would require explicit choice points for every possible value of the context values x, y, gx, and gy as in the following.

```c
#define DIR { N E S W }
while (notAtGoal(x,h,gx,gy))
    if (x == 0 && y == 0 && gx == 0 && gy == 0)
        dir = DIR
    else if (x == 0 && y == 0 && gx == 0 && gy == 1)
        dir = DIR
    else if (x == 0 && y == 0 && gx == 0 && gy == 2)
        dir = DIR
    ...
    else if (x == MAX_X && y == MAX_Y && gx == MAX_X && gy == MAX_Y)
        dir = DIR
    (x,y) = move(dir,(x,y))
```

Moreover, if the context’s type is infinite or cannot be statically determined for instance: an unbound integer, a string of characters, or unbound list, then this transformation could not be represented.
3.4 Systems for Automatic Configuration and Self-Aware Computing

As discussed in the introduction, tunable constants such as cutoff threshold for different algorithms are a motivation for some form of automatic programming. These and other configuration “knobs” are a common practice in general-purpose algorithm libraries (such as searching and sorting algorithms) and thus are a popular target for “automated tuning”. This work can be viewed as a very minimal form of partial programming in that there is an existing program or algorithm. However, the exact threshold and cutoff values or perhaps strategies used are uncertainties that must be learned or tuned in some way. The following are various formal approaches this “auto-tuning” problem.

3.4.1 FFTW — Learning a Faster Fourier Transform

In fact, the earlier example of finding optimal cutoffs in sorting algorithms (between mergesort and insertion sort) is quite trivial compared to recent work in the field of auto-tuning algorithms. Frigo and Johnson take a more scientific approach to automatic optimization by demonstrating a systematic adaptive technique [26, 27] in their industrial-strength FFTW library, which is used for computing discrete Fourier transforms. They observe that in modern computer architectures the number of (floating point) operations for an algorithm is not necessarily indicative of how fast an algorithm will run due to pipelining and variations in specialized hardware on different target machines (e.g. SIMD instructions).

In their self-optimizing approach an executor forms the general skeleton outline of the FFT algorithm handling recursive subdivision of the problem into smaller problems and recombination of the smaller solutions into a larger solution. To accomplish this, the executor selects from library of codelets, specialized chunks of highly optimized C code (generated by a higher-level
language). These codelets implement various parts of the recursive FFT algorithm in different ways for different problem sizes as well as offering different strategies for decomposing and composing the result. The executor tests various inputs and uses dynamic programming to compute the optimal plan for problems of a given size. Later program runs use this optimal plan without timing or exploring further plans much like an optimal policy where the state is the problem size.

This approach is a very specialized instance of partial programming. The executor is the partial program since it contains the recursive FFT algorithm skeleton. Sub-parts of the algorithm are implemented via codelets, which correspond to actions in a way. The problem size can be viewed as a recognition context of sorts. The goal of minimizing execution time can be considered a reward signal, and the final optimal resulting plan for each problem size can be viewed as a policy (in RL terms).

The recursive divide-and-conquer nature of the FFT algorithm employed [20] allows FFTW to use dynamic programming instead of RL. Specifically, FFTW assumes an optimal substructure [21] on their problems meaning that once they find the optimal solution for a given $N$, they can use that optimal solution as a sub-solution for problems of a larger size.

### 3.4.2 PetaBricks

The PetaBricks project [5] addresses the static configuration problem through the development of a new declarative programming language. In PetaBricks programmers specify algorithms in a non-deterministic way. The approach excels for problems where different strategies perform better in different contexts (specifically on different hardware configurations).
Recall that in FFTW a similar judgement was being made about the recursive decompositions at each step (depending on their size). This approach can be seen as a generalization of that approach by formalizing it as a language.

In PetaBricks, algorithms are called transforms and consist of various rules which may non-deterministically be selected. That is, a compiler may choose from some or all of the rules for each transform. Each rule operates over some region of the problem.

To illustrate these concepts, we reproduce their example of a matrix multiplication algorithm in their language.
transform MatrixMultiply from A[c,h], B[w,c] to AB[w,h]
{
    // Base case, compute a single element
    to(AB.cell(x,y) out) from(A.row(y) a, B.column(x) b) {
        out = dot(a,b);
    }

    // Recursive decomposition in c
    to(AB ab) from(A.region(0, 0, c/2, h) a1,
      A.region(c/2, 0, c, h) a2,
      B.region(0, 0, w, c/2) b1,
      B.region(0, c/2, w, c) b2) {
      ab = MatrixAdd(MatrixMultiply(a1, b1),
                      MatrixMultiply(a2, b2));
    }

    // Recursive decomposition in w
    to(AB.region(0, 0, w/2, h) ab1,
      AB.region(w/2, 0, w, h) ab2) from(A a,
       B.region(0, 0, w/2, c) b1,
       B.region(w/2, 0, w, c) b2) {
      ab1 = MatrixMultiply(a, b1);
      ab2 = MatrixMultiply(a, b2);
    }

    // Recursive decomposition in h
    // ... elided
}
// MatrixAdd, elided
Here, the MatrixMultiply rule has four rules (one elided for space). Each rule represents a possible strategy for multiplying parts of matrices \( A \) and \( B \) of the given dimension. For instance, the second rule ("decomposition in \( c \") represents a split along the shared dimension of the two matrices being multiplied (i.e. \( c \)). The compiler can decide, based on the size of the input matrices \( A \) and \( B \), which rules to select. Notice that each rule specifies the region it operates on within the matrix. The corresponding dimension arguments (arguments to \( A \) and \( B \)) allow the tuning algorithm of the compiler to decide how to subdivide the problem. The compiler computes a dependency graph for each rule and can safely decide what can be computed in parallel and various strategies for each. The programmer can specify a generator for data to tune the algorithm with, and additionally threshold constants can be tagged with a tunable keyword to indicate the compiler should test different values there.

The PetaBricks compiler generates a self-tuning program that tests various options in a disciplined manner, starting with smaller problems and tuning them before progressing to larger ones. In this way it is similar to FFTW, but potentially less fine-grained as it doubles the problem size each step (where FFTW learned over all sizes). The output of all this is the most efficient algorithm the process explored.

The notion of regions and dimensions on the arguments (implying arrays or matrices) illustrates the distinct bias this problem has towards such numeric problems. Furthermore, this work showcases itself on algorithms such as adaptive sorting, an "eigenvector solve" problem, as well as the matrix multiply shown above, and a few other numerical problems that operate on large arrays or matrices. Moreover, execution time is the built-in feedback signal that limits this language to performance-critical problems. As a result, PetaBricks could not evaluate a program completion based on other criteria such as numerical stability or space usage which are both relevant concerns in numeric computing.
In this language the transform is the partial program since it contains the choice (rules to select from). The auto-tuning system uses time as the algorithm fitness metric, and the context is problem’s size.

3.5 Summary

This section summarizes the most prominent approaches discussed in this chapter. Table 3.1 provides a compact overview of these approaches. The ALISP work (and HAMs) started as a method of constraining RL problems. The ALISP programs (or machine hierarchies) are always forcibly decoupled from the MDP representing the problem domain; state and reward information are managed by an external MDP. However, that abstraction leaks and details of the problem that should remain hidden must be shared. This leak is visible through the addition of the state

<table>
<thead>
<tr>
<th>Approach</th>
<th>Choice Construct</th>
<th>Adaptation Method</th>
<th>Feedback/Reward</th>
<th>State Abstraction</th>
<th>Problem Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABP</td>
<td>Run-time library calls</td>
<td>RL</td>
<td>reward functions</td>
<td>Context argument to choose</td>
<td>General purpose</td>
</tr>
<tr>
<td>HAM</td>
<td>Choice states</td>
<td>Hierarchic RL</td>
<td>External (not part of the language)</td>
<td>No support</td>
<td>Hierarchic problems</td>
</tr>
<tr>
<td>ALISP</td>
<td>choice macro</td>
<td>Hierarchic RL</td>
<td>External (not part of the language)</td>
<td>Optional argument to action macro</td>
<td>Hierarchic / general</td>
</tr>
<tr>
<td>A²BL</td>
<td>adaptive key- word applied to behavior definitions</td>
<td>RL and Modular RL (for parallel behaviors)</td>
<td>reward clauses</td>
<td>state clauses</td>
<td>Behavior systems</td>
</tr>
<tr>
<td>Program Sketching</td>
<td>#define syntactic grammars</td>
<td>CEGIS (inductive synthesis)</td>
<td>Correctness assertions</td>
<td>N/A (stateless)</td>
<td>General purpose</td>
</tr>
<tr>
<td>FFTW</td>
<td>The executor picks different problem decompositions</td>
<td>Dynamic programming</td>
<td>Time</td>
<td>Implicit (problem size)</td>
<td>The Cooley-Tukey FFT [20]</td>
</tr>
<tr>
<td>PetaBricks</td>
<td>rules within the transform</td>
<td>Hybrid of dynamic programming and correctness assertions</td>
<td>Implicit (sub-problem dimensions)</td>
<td>Numerical/scientific recursively decomposable algorithms</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of Partial Programming Approaches
abstraction mechanism and the notable absence of a reward statement in the ALISP language. Moreover, to use the state abstraction mechanism one must understand details of the learning algorithm. Follow-on languages to ALISP such as concurrent ALISP deal with the problem more explicitly, for example, by adding functions to fetch the current environment from the MDP.

The $A^2$BL language is a more specialized partial programming language. Specifically, it extends a language for representing behaviors with adaptive constructs to define behaviors which are learned based on various reward annotations. Like ABP, and unlike ALISP, the approach explicitly combines the state and reward information into the program.

The Program Sketching approach covered is fairly different from all the others. Program instances are checked via correctness assertions, and failed assertions guides the synthesis procedure. Using assertions as a feedback mechanism are somewhat limiting in that the feedback type is boolean. Methods using RL typically use numeric values (rewards) and are a bit more expressive. There is no notion of context or environment in these systems, since the adaptation system has no use for them. Finally, programs in these systems often require fairly complicated assertions to actually find a proper algorithm.

Finally, we discussed a bit of the background and recent work in so called “autotuning systems” or systems for automatic configuration, the most notable of these being a language called PetaBricks. PetaBricks provides a system for users writing high-level skeletons for recursively decomposable numeric problems with different strategies possible at each recursive step (such as matrix multiplication). The PetaBricks compiler experiments with various strategies for decomposition on a given platform and hardware configuration and generates highly efficient program text for that environment. While PetaBricks is much more specialized system than ABP, the notion of state (context) and reward exist (at least implicitly) as problem size and execution time respectively.
Chapter 4: Functional Adaptation-Based Programming

This chapter presents a view of ABP from earlier work [11] where ABP is realized within the functional programming language Haskell [56]. In this approach to adaptive programming, the ABP implementation as a domain-specific language (DSL) embedded within Haskell.

DSLs [35, 72, 25] are sometimes referred to as “little languages” [13] or “mini-languages” [59] (specifically chapter 8). They are specific-purpose programming languages as opposed to general-purpose languages, and as such they trade away the ability to represent every problem for the ability to represent specific types of problems concisely (e.g. many are not Turing complete). Additionally, DSLs are frequently embedded within a general-purpose programming language and operate much like a sophisticated programming library with a coherent set of semantics.

DSLs have been used in many domains including build systems [52], game theory simulation [74], system scripting [9], and a host of other applications. The goal of the work within this chapter is to define the abstractions specific to adaptive programming, explore their interactions, and develop a firm understanding of these ideas so that they can be applied to other implementations of ABP.

Haskell is a popular host language for embedded DSLs due to its flexible syntax and powerful type system [34, 36]. Additionally, Haskell provides abstractions that facilitate fast experimentation of ideas for DSLs. Its type system forces language developers to be precise in the description of the DSL constructs while offering enough flexibility to describe elements in their most general form. In particular, type classes together with type functions [63] provide an elegant way of formulating the notion of compositional adaptive programs.
The adaptive DSL is defined around a type class and multiple functions that transform and operate on instances of it. Programs of the DSL consist of instances of this type class and allow the user to specify uncertainty. The language also provides template DSL programs for common patterns in the form of generic instances such as adaptive pairs and functions as well as operations supporting various patterns of evolution and adaptation.

Some of the topics addressed in this chapter include:

- Identification of adaptive values as a foundation for adaptation-based programming and their formalization through a corresponding Haskell type class.

- The definition of specific instances of adaptive values, with intuitive interpretations, to be used as building blocks for adaptive programs. In many cases these building blocks draw on machine learning theory to provide formal guarantees regarding their adaptation behavior.

- Identification and definition of adaptable computation patterns that are likely to arise in common practice.

- A report on some practical experiments that illustrate the potential utility of adaptive programming.

The remainder of this chapter is structured as follows. Section 4.1 introduces the notion of adaptive values and define the interface to adaptive values through type classes. The use of adaptive values to build adaptive computations is demonstrated in Section 4.2. This section identifies adaptive computation patterns that correspond to standard procedures in machine learning and those that are likely to arise in some typical uses of ABP. Section 4.3 presents functions to monitor and control adaptive computations. Finally, section 4.4 provides some empirical results for the application of this model of ABP.
4.1 Adaptive Values

The usual understanding of a value in the context of programming languages is that of a constant, unchanging object. In contrast, an adaptive value can change over time. Changes to an adaptive value are determined by feedback gathered from the context in which it is used.

To facilitate a meaningful, controlled adaptation an adaptive value of type \( v \) needs to be represented, in general, by a somewhat “richer” type \( a \), that is, type \( a \) allows the extraction of values of type \( v \), but also contains enough information to support interesting forms of adaptation.

We call \( a \) the representation type and \( v \) the value type of \( a \). The adaptation is controlled by values of another type \( f \), called the feedback type of \( a \). In the following we call an adaptive value an **adaptive** for short to avoid ambiguities between an adaptive value and the “value of an adaptive value”, that is, we simply say that \( x :: a \) is an adaptive and \( \text{value} x :: v \) is the value of (the adaptive) \( x \). (The function \( \text{value} \) will be defined in Section 4.1.1.)

In Section 4.1.1 we describe the definition and examples of basic adaptives, that is, adaptives defined directly on specific representation types. In Section 4.1.2 we discuss obvious ways of obtaining compound adaptives through derived instances for type constructors. A particularly useful instance of this is the derived instance for function types that leads to **contextual adaptives** to be discussed in Section 4.1.3. In Section 4.1.4 we describe how to construct new adaptives through nesting.

4.1.1 Defining Adaptives

The described concept of adaptives can be nicely captured by the following Haskell type class.
class Adaptive a where
  type Value a
  type Feedback a
  value :: a -> Value a
  adapt :: Feedback a -> a -> a

This class constitutes the core of our DSL: the operation value retrieves the current value from the representation, and the function adapt takes a feedback value and an adaptive and produces a new adaptive. We represent points of uncertainty in our program as instances of this class.

To define an adaptive representation type, a programmer has to provide an instance definition for the class Adaptive, which requires

- implementations for the functions value and adapt, and
- a definition of the corresponding value and feedback types

The value and feedback types are associated with the representation type a through the type functions Value and Feedback, which allows a large degree of flexibility in defining the adaptive behavior [63].

There are more things that we ultimately might want to store for adaptive values for practical purposes (for example, statistics about usage, feedback, and adaptation/adaptive behavior). We will consider this aspect later in Section 4.3.

As a simple example program we consider a form of incremental linear regression. In particular, we want to learn the equation of a line \( y = mx + b \) given a sequence of sample data points \((x_1,y_1), (x_2,y_2), \ldots\). The goal is to adjust \( m \) and \( b \) to values that minimize the squared error of predicting \( y_i \) given \( x_i \).

The adaptive for this example could be defined as follows. First, we define the slope/intercept representation of lines.
type Slope = Double
type Intercept = Double

data Line = L Slope Intercept

type Point = (Double, Double)

Based on this representation we can define the line adaptive as follows.

    instance Adaptive Line where
      type Value Line = Line
      type Feedback Line = Point
      value = id
      adapt (x,y) (L m b) = L m’ b’
        where m’ = m + eta*x*(y - y0)
              b’ = b + eta*(y - y0)
              y0 = m*x + b
      eta = 0.01

We can observe that the value of this particular adaptive is just the same as the representation. The feedback is provided in the form of individual points, each of which leads to an update of slope and intercept as defined by the expressions for \( m' \) and \( b' \). The value \( \eta \) represents the learning rate, which is how much new inputs influence the adaptation. Our chosen small constant will work for small (< 20) input points.

As another example, consider the game of Rock-Paper-Scissors, in which two players simultaneously choose one of three values Rock, Paper, or Scissors, trying to beat the opponent.

    data Move = Rock | Paper | Scissors

The winning move against each move is defined by the following function \( \text{win} \).

    win :: Move \to\ Move
    win Rock = Paper
    win Paper = Scissors
    win Scissors = Rock
It turns out that, given a fixed opponent, this game is a specific instance of a so-called “multi-armed bandit” problem. This is a classic problem, first described by Robbins [60], which captures the essential elements of many experimental design problems, among others. The problem can be viewed as modeling the process of playing a slot machine with multiple arms, where each arm has an unknown distribution of random payoffs. At each time step the player must select an arm to pull based on information gathered from previous pulls, upon which a randomized return from the selected arm is received. The goal is to develop an arm-pull strategy that maximizes some measure of the expected payoff sequence over time, e.g. maximizing the expected temporally-averaged payoff. In the case of Rock-Paper-Scissors with a given opponent strategy, the arms correspond to the selection of either rock, paper, or scissors, and the payoff reflects whether the selected move won or lost against the selection of the opponent at that time step.

A good bandit strategy must balance the exploitation-exploration trade-off, which involves deciding whether to exploit the current knowledge and pull the arm that currently looks best, or to explore other arms that have been tried fewer times in the hope of discovering higher payoffs.

There are well known lower bounds on the performance of the best possible strategy and bandit strategies that achieve those bounds asymptotically [42]. More recent work [7] has developed an upper confidence bound (UCB) strategy, which was shown to achieve the lower bound uniformly over all finite time periods. Below, we describe a multi-armed bandit adaptive based on UCB.

In our representation of a multi-armed bandit we store a map that gives for each arm how often it was pulled and the total rewards collected with it. The representation is parameterized by the type used to represent the bandit’s arms.

type Reward = Float
type Pulls = Int
data Bandit a = Bandit (PlayMap a)
type PlayMap a = [(a,Pulls,Reward)]
The definition of the bandit adaptive has to return arm values (of type \( a \)) as values. The feedback is the arm that was pulled last together with a reward that will be added to the total reward of that arm in the map.

We define the helper function `updPM` to update the play map for a given arm in some generic way.

\[
\text{updPM} :: \text{Eq} \ a \Rightarrow (\text{ArmInfo} \ a \rightarrow \text{ArmInfo} \ a) \rightarrow a \rightarrow \text{PlayMap} \ a \\
\text{updPM} \ f \ x \ (a:as) = \begin{cases} 
    f \ a:as & \text{if} \ \text{fst3} \ a == x \\
    a:\text{updPM} \ f \ x \ as & \text{otherwise}
\end{cases}
\]

\[
\text{fst3} \ (x,_,_) = x
\]

With these definitions we can define a multi-armed bandit as an instance of an adaptive.

\[
\text{instance} \ \text{Eq} \ a \Rightarrow \text{Adaptive} \ (\text{Bandit} \ a) \text{ where}
\text{type Value} \ (\text{Bandit} \ a) = a
\text{type Feedback} \ (\text{Bandit} \ a) = (a, \text{Reward})
\text{adapt} \ (a,r) \ (\text{Bandit} \ m) = \text{Bandit} \ (\text{addReward} \ r \ a \ m)
\text{where} \ \text{addReward} :: \text{Eq} \ a \Rightarrow \text{Reward} \rightarrow a \rightarrow \text{PlayMap} \ a \rightarrow \text{PlayMap} \ a
\text{addReward} \ x = \text{updPM} \ (\lambda (a,p,r) \rightarrow (a,p+1,r+x))
\]

What remains to be defined is the value method, for which we use the UCB bandit algorithm. This approach first selects any arm that has not been pulled before, which is achieved by the function `zeroPulls`, and otherwise selects the arm with the highest upper confidence bound. This measure is defined for an arm \( i \) that has been pulled \( n_i \) times and has a reward sum of \( r_i \) as

\[
r_i/n_i + \sqrt{\log n/n_i} \text{ where } n = \sum_n n_i.
\]

\[
\text{value} \ (\text{Bandit} \ m) = a
\text{where} \ ((a,_,_,):_) = \text{zeroPulls} \ ++ \ \text{sortDesc ucb} \ m
\text{zeroPulls} = \text{filter} \ ((==0).\ \text{pulls}) \ m
n = \text{fromIntegral} \ (\text{sum} \ (\text{map} \ \text{pulls} \ m))
\text{ucb} \ (_,p,r) = r/n_i + \sqrt{\log n/n_i}
\text{where} \ n_i = \text{fromIntegral} \ p
\text{pulls} \ (_,p,_) = p
The above function extracts arm a by first choosing any arm that has not been pulled (from zeroPulls). If all arms have been pulled, then it chooses the maximum value according to the UCB computation given above. The function sortDesc sorts a list in descending order of values as obtained by the parameter function ucb.

It is illustrative to note how the above UCB-based implementation of value manages the exploration-exploitation trade-off. Assuming that all arms have been pulled at least once, the decision is based on the upper confidence bound, which is composed of two terms. The first term \( r_i/n_i \) can be viewed as encouraging exploitation since it will be larger for arms that have been observed to be more profitable on average. Conversely, the second term encourages exploration since it grows with the total number of arm pulls, causing it to overwhelm the first term if an arm has not been pulled very often. However, the exploration term vanishes very quickly for an arm as its number of pulls increases causing its evaluation to be based solely on its observed returns. The result is that low-payoff arms tend to get fewer pulls than those with higher payoffs over time, as desired.

The Bandit instance is a generic operation in our DSL, it can be utilized by many consumer programs. We illustrate one such use by coming back to our Rock-Paper-Scissors example and instantiating the bandit as an adaptive strategy for playing the game.

```haskell
type Strategy = Bandit Move

initStrat :: Strategy
initStrat = Bandit [(m,0,0) | m <- [Rock, Paper, Scissors]]
```

We can use the following function `score` to translate wins and losses into numerical feedback.

```haskell
score :: Move -> Move -> Int
score m m' | win m == m' = -1  
            | win m' == m   = 1  
            | otherwise     = 0
```
We can then pair \texttt{initStrat} with other strategies and observe how it adapts guided by the feedback values produced from \texttt{score} applied to the moves produced by \texttt{value} and the opponent's move. We will do this in Section 4.2 where we will identify and formalize adaptation computation patterns that allow us to define applications (such as, line regression or Rock-Paper-Scissors tournaments) that employ the defined adaptives.

One final note regarding the feedback employed for the multi-armed bandit: the theoretical optimality result assumes the rewards are in the range \([0..1]\). To adjust the \texttt{Bandit} adaptive to the feedback produced by \texttt{score} we just needed to multiply the \texttt{sqrt} term by 2. However, in this example the optimal behavior is not affected even if we do not scale the rewards since all we are interested in is average reward.

### 4.1.2 Derived Adaptives

We define adaptation of generic structures in this DSL by defining derived instances of \texttt{Adaptive}. This gives us instances of adaptives for many common patterns in adaptive programs.

As an initial example, we define a derived instance of \texttt{Adaptive} for pairs, which realizes the parallel adaptation of two values in a synchronized fashion.

\[
\text{instance } (\text{Adaptive } a, \text{Adaptive } b) \Rightarrow \text{Adaptive } (a,b) \text{ where }
\]

\[
\begin{align*}
\text{type Value } (a,b) &= (\text{Value } a, \text{Value } b) \\
\text{type Feedback } (a,b) &= (\text{Feedback } a, \text{Feedback } b) \\
\text{value } (x,y) &= (\text{value } x, \text{value } y) \\
\text{adapt } (u,v) (x,y) &= (\text{adapt } u x, \text{adapt } v y)
\end{align*}
\]

One example use of this is the parallel adaptation of two competing or even cooperating adaptive strategies in a game. For instance, an agent might have two goals that need to be satisfied concomitantly. Then two \texttt{Bandits}, one adapting to each goal automatically, form a more complex agent that addresses both with no additional programming.
Another example use of this particular construct will be given in Section 4.2 where we can derive a co-evolution computational pattern from a simple evolution pattern by using this class instance definition.

We can also obtain an Adaptive definition for lists. In this definition, each adaptive’s feedback value is used exclusively for that adaptive.

```haskell
instance Adaptive a => Adaptive [a] where
  type Value [a] = [Value a]
  type Feedback [a] = [Feedback a]
  value = map value
  adapt = zipWith adapt
```

This definition can be generalized to any Functor type constructor, because we can easily define a corresponding fzipWith function.

### 4.1.3 Contextual Adaptives

A frequent scenario is to extend a given adaptive by context. For example, the best arm to pull for a multi-armed bandit may depend on the time of day. Such a context extension can be very conveniently achieved through the derived Adaptive instance for function types. The idea is to turn an adaptive for some type `a` into an adaptive for functions from some context `c` into `a`. The value type of such an adaptive function is a function from context into values of the original adaptive `a`, and feedback is given by feedback for `a` enriched by context information. Contextual adaptive values are obtained in two steps. First, apply the function to contextual information `x`, and then extract the value of that result. Adaptation based on a feedback `(x, v)` constructs an updated function that overrides input `x` to map to the adapted result of `(f x)` with feedback `v`. All other inputs are delegated to the old function. This definition illustrates that the functional adaptive essentially maintains a number of separate copies of the original adaptive.
instance (Eq c, Adaptive a) => Adaptive (c -> a) where
  type Value (c -> a) = c -> Value a
  type Feedback (c -> a) = (c, Feedback a)
  value f = \x -> value (f x)
  adapt (x, v) f = \y -> if x==y then adapt v (f x) else f y

The definition for value could be given more succinctly as (value .), but we think the
above definition is easier to understand and explains better what is going on.

This derived instance effective expands our DSL to support function types transparently.

Note that this Adaptive instance definition can be easily generalized to a whole class of
context type constructors, of which -> is one example. A mapping type is another example,
which might be preferable for efficiency reasons.

As a concrete example we can add a player context to the multi-armed bandit representing the
Rock-Paper-Scissors player, which then allows the adaptive to learn different strategies against
different players.

data Opponent = Jack | Jill deriving Eq

  flexible :: Opponent -> Strategy
  flexible = \_ -> initStrat

Note that this context-dependent strategy is obtained for free since it is based on the automati-
cally derived instance of Adaptive for function types. For either player, the initial strategy is
used, but as the function receives feedbacks it will adapt more specialized strategies for each
player (input).

4.1.4 Nested and Recursive Adaptives

Another way in which adaptives can be combined into more complex adaptives is through nest-
ing, that is, the value of one adaptive is another adaptive. In such a nested adaptive, value
selection and adaptation happens on two levels. While an “ordinary” adaptive represents an evolving decision, a nested adaptive represents a sequence of such decisions.

To work effectively with nested adaptives it is not sufficient to simply place one adaptive as a value into another one, because adaptation of the nested adaptives would be impossible. The adapt function for the outer adaptive would simply adjust the selection of the nested adaptive. Although a nested adaptive that is obtained by the value function of the outer adaptive can be adapted, there is no mechanism to put this changed adaptive back into the outer one.

Therefore, we define a subclass of Adaptive, called Dedaptive, to represent dependent adaptives. These contain an extended value function valueCtx, which returns the value plus the context where it was found. This context is a function that allows the value, or an adapted version of it for that matter, to be put back into the containing adaptive. The class also contains a function propagate that allows the derivation of feedback for the outer adaptive from feedback for the nested one. The additional first parameter of type a serves two purposes: First, it is needed to resolve the overloading of propagate, and second it provides a context of values to properly derive feedback, because in some situations, the feedback type contains more than just an external value, but also information related to the adaptive type.

```haskell
class (Adaptive a, Adaptive (Value a)) => Dedaptive a where
    valueCtx :: a -> (Value a, Value a -> a)
    propagate :: a -> Feedback (Value a) -> Feedback a
```

Note that the dependency in nested adaptives goes both ways: the nested adaptive depends as a value on the outer adaptive, while the outer adaptive’s adaptation is in part controlled, through propagate, by the nested adaptive.

As an example we can consider a nested multi-armed bandit. The nested bandit could be a Rock-Paper-Scissors game or actually a gambling machine, while the outer bandit might represent, for example, the decision at which time to play.
In the instance definition of Dedaptive, the function `valueCtx` is based on the outer `value` function to find the value. The context is then simply obtained by isolating that value in a list and producing a function that can insert an element in its place. Since the feedback for a bandit of type `a` is given by values of type `(a,Reward)`, we can produce feedback for the outer bandit simply by pairing the reward provided for the nested one with the current value of the outer one.

\[
\text{instance } (\text{Eq } a, \text{Eq } (\text{Bandit } a)) \Rightarrow \text{Dedaptive } (\text{Bandit } (\text{Bandit } a)) \text{ where }
\]
\[
\text{valueCtx } b@(\text{Bandit } m) = (a, \lambda y \rightarrow \text{Bandit } (xs++(y,p,r):ys))
\]
\[
\text{where } a = \text{value } b
\]
\[
(xs, (\_, p, r):ys) = \text{break } ((==a).\text{fst3}) m
\]
\[
\text{fst3 } (x, (\_, \_), \_) = x
\]
\[
\text{propagate } b (\_, r) = (\text{value } b, r)
\]

We can now create a nested adaptive as follows.

\[
\text{dependent } :: \text{Bandit } \text{Strategy}
\]
\[
\text{dependent } = \text{Bandit } [(\text{initStrat}, 0, 0), (\text{initStrat}, 0, 0)]
\]

It seems that `dependent` is very similar to `flexible`, and in fact, we can simulate contextual adaptives by nested adaptives. However, nested adaptives are more general since we can nest different adaptives (of the same type) if we want, which is not possible for contextual adaptives. This situation is reminiscent of the relationship between dependent and independent products in type theory [71].

Nested adaptives also raise the question of "nested values", that is, when we want to get the value of a dedaptive, we in many cases do not want to have the immediate value, which is itself an adaptive, but rather the "ultimate" value, that is, the value of the nested adaptive. This can be easily computed by the function `nestedValue`.

\[
\text{nestedValue } :: \text{Dedaptive } a =\Rightarrow a \Rightarrow \text{Value } (\text{Value } a)
\]
\[
\text{nestedValue } = \text{value } . \text{value}
\]
4.2 Programs for Adaptive Computation

The idea behind this adaptation DSL is the gradual evolution of values to improve a programmatic solution to a problem. This view requires that an adaptive computation, that is, a computation that contains adaptive values, is performed repeatedly so that feedback, often obtained from the results of the computation, is used to evolve the adaptives employed in the computation.

Under this view, an adaptive computation has to contain (repeated) calls to adapt functions, and we can distinguish different adaptive computation patterns based on the relationship of these adaptation steps with other computations.

One of the most basic adaptation operations in our DSL is given by the adapt function itself, namely the one-step adaptation of an adaptive. More complex patterns can be obtained by considering different forms of repeated adaptation.

What is the result of an adaptive computation? Is it the final adaptive or the trace of values that can be obtained from the list of all intermediate adaptives, or both, or something else entirely? For generality we define combinators for adaptive computation patterns to return the list of all adaptives produced during the adaptation. From this list we can easily obtain the final adaptive through the list function last or the trace of represented values through the function valuesOf, which is defined as follows.

\[
\text{valuesOf :: Adaptive a => [a] -> [Value a]} \\
\text{valuesOf = map value}
\]

Other inspection and debugging functions for sampling or aggregating can be added quite easily through ordinary list processing functions.
4.2.1 Adaptation Combinators

One of the most basic adaptation patterns is to train an adaptive by a list of training values analogous to supervised learning [15]. This is realized by the function `trainBy` below.

\[
\text{trainBy} :: \text{Adaptive } a \rightarrow a \rightarrow [\text{Feedback } a] \rightarrow [a]
\]
\[
\text{trainBy} = \text{scanl} \ \text{adaptBy}
\]
\[
\text{adaptBy} :: \text{Adaptive } a \rightarrow a \rightarrow \text{Feedback } a \rightarrow a
\]
\[
\text{adaptBy} = \text{flip} \ \text{adapt}
\]

The `scanl` function returns a list of all intermediate results as a leftward fold is applied to a list. Here it will adapt an initial adaptive in sequence and return the list (stream) of all intermediate adaptives.

A more dynamic scenario is captured by the function `evolve` that uses its function parameter to compute feedback from the values of an adaptive.

\[
\text{evolve} :: \text{Adaptive } a \rightarrow (\text{Value } a \rightarrow \text{Feedback } a) \rightarrow a \rightarrow [a]
\]
\[
\text{evolve} \ f \ x = x : \text{evolve} \ f \ (x \ \text{‘adaptBy’} \ (f \ (\text{value } x)))
\]

The function `evolve` represents a form of online learning [15] where the adaptive can be viewed as alternating between making a decision (producing a value), getting feedback, and then adapting. The bandit problem is a classic example of online learning, though there are many other instances in the literature.

A generalization of `evolve` is obtained by evolving two adaptives in parallel where the values of both adaptives are the basis for feedback to either one of the adaptives. This definition makes use of the `Adaptive` instance for pairs shown in Section 4.1.2. The function `distr` makes the values of both adaptives available to compute feedback.
The adaptation pattern defined by `coevolve` corresponds to the structure of multi-agent reinforcement learning [45], an area of reinforcement learning that studies situations where multiple agents are learning simultaneously, possibly interacting with one another either cooperatively or as adversaries.

As an example we consider the implementation of a Rock-Paper-Scissors tournament. In addition to players, such as `initStrat` described in Section 4.1.1, we need functions to produce feedback values from the values of two players. One such function is `myScore`.

```haskell
myScore :: Move -> Move -> (Move,Reward)
myScore x y = (x,score x y)
```

Since different player adaptives might have other feedback types, we generally need other functions as well. For example, a simple Rock-Paper-Scissors strategy is to always play the move that wins against the last move of the opponent.

```haskell
data BeatLast = BL Move

instance Adaptive BeatLast where
    type Value BeatLast = Move
    type Feedback BeatLast = Move
    value (BeatLast m) = m
    adapt m (BeatLast _) = BL (win m)
```
Recall coevolve uses the value of both adaptives to produce the corresponding feedback value for the adaptive. The function below can be used to select the opponent’s move from the previous round and fits nicely with the above strategy.

```haskell
opponent'sMove :: Move -> Move -> Move
opponent'sMove _ y = y
```

Or consider a smarter strategy that plays the move that beats its opponent’s most frequently played move. This player maintains a count that each move has been played.

```haskell
data Max = MP [(Move,Int)]

instance Adaptive Max where
  type Value Max = Move
  type Feedback Max = Move
  value (MP ms) = win (fst (maxWrt snd ms))
  adapt m (MP ms) = MP (updF m (+1) ms)
```

The function `updF` updates a mapping in a list of pairs and `maxWrt` selects the maximum element from a list with regards to some specific criteria (``snd`` meaning the second element).

```haskell
updF :: Eq a => a -> (b -> b) -> [(a,b)] -> [(a,b)]
updF x f [] = []
updF x f ((y,w):as) | x==y = (x,f w):as
                  | otherwise = (y,w):updF x f as
```

We can now define players as pairs of adaptive values plus their corresponding feedback-producing functions.

```haskell
bandit = (initStrat, myScore)
beatLast = (BL Rock, opponent’sMove)
maxMv = (MP [(m,0) | m<-rps], opponent’sMove)
```

To be able to play strategies with their corresponding feedback function against one another, we introduce the following tournament function.
Tournaments can then be played using \texttt{vs} in the obvious way, for example:

\begin{verbatim}
beatLast \textquote{vs} maxMv
\end{verbatim}

This example leads as expected to an overall victory for the \texttt{maxMv} player.

### 4.2.2 Recursive Adaptation

In Section 4.1.4 we have considered nested adaptives, in which value selection and adaptation happens on two or more levels. While an “ordinary” adaptive represents an evolving decision, a nested adaptive represents a sequence of such decisions.

When the number of nesting levels is not fixed and not known in advance, it is difficult to capture this computational pattern in a single combinator. In that case, adaptation and value retrieval must be performed by individual function calls that are integrated into the recursive structure of an adaptive algorithm.

As an example we consider the problem of learning a combination of sorting methods. The idea is based on the observation that for specific kinds of lists, one sorting method performs better than others.

To learn a combination of sorting algorithms we have to abstract some property of lists and store costs or rewards for each sorting method under consideration in a table indexed by that property. Since some sorting methods are recursive, this will lead to a recursive adaptation process in which potentially different sorting methods can be chosen based on the respective properties of lists decomposed during the sorting recursion.
For simplicity we consider here the length of the list as a property.\footnote{We actually use the square root of the list length to keep the size of the table reasonably small.} We can build this adaptive table in two steps. First, we define an adaptive for sorting methods, from which we can then create a table by adding the list size as context, as demonstrated in Section 4.1.3.

```haskell
data SortAlg = MSort | ISort
type Cost = Double
data Action = Action [(SortAlg,Int,Cost)]
```

The base adaptive for sorting algorithms has essentially the same structure as a multi-armed bandit (see Section 4.1.1): It stores the number of times each method was chosen together with the cost (representing running time). Here we consider two methods, namely insertion sort and mergesort.

The `Adaptive` instance definition for `Action` is also very similar to that of `Bandit`. The only differences are that `value` selects the smallest entry (that is, the on average fastest sorting method) and that `adapt` updates a running average of costs via the `updAvg` function. We also choose any action not sufficiently explored (8 is used as an arbitrary cutoff to decide this).

```haskell
instance Adaptive Action where
type Value Action = SortAlg
type Feedback Action = (SortAlg,Cost)

value (Action as)
  | null unexplored = fst3 $ minWrt thd3 as
  | otherwise = fst3 $ head unexplored
  where unexplored = filter (\a -> snd3 a < 8) as

adapt (a,c) (Action as) = Action (updF3 g)
  where g (a',f',c') = (a', f' + 1, runAvg f' c' c)) as
```

```
The function \texttt{runAvg} updates a running average, \texttt{minWrt} selects the minimum element with regard to some criteria in our case the average time a sorting method takes, and \texttt{updF3} remaps a specific triple in a list.

\begin{align*}
\text{runAvg } f \ c' \ c &= \frac{fd \ast c' + c}{fd + 1} \\
\text{where } fd &= \text{fromIntegral } f
\end{align*}

\begin{align*}
\text{minWrt :: Ord b => } (a -> b) -> [a] -> a \\
\text{minWrt } f &= \text{head . sortBy } (\backslash x \ y -> \text{compare } (f x) (f y))
\end{align*}

\begin{align*}
\text{updF3 :: Eq a => a -> ((a,b,c) -> (a,b,c)) -> [(a,b,c)] -> [(a,b,c)]} \\
\text{updF3 } x \ f \ [] &= [] \\
\text{updF3 } x \ f \ (a:\text{as}) | x == \text{fst3 } a &= f \ a : \text{as} \\
| \text{otherwise} &= a : \text{updF3 } x \ f \ \text{as}
\end{align*}

To support unlimited recursive adaptives, we use the adaptive as the state of a state monad, which can then be used to thread adaptives through arbitrary computations. To facilitate the computation of actual timings for the given application, we use a state monad transformer that encapsulates the \texttt{IO} monad. The following general definition of a Q-table \cite{70} abstracts from the concrete types for state/context (s) and actions (a).

\begin{align*}
\text{type QTable } s \ a \ r &= \text{StateT } (s -> a) \ \text{IO } r
\end{align*}

Note that the state of the state transformer monad is a function that represents a contextual adaptive. For our example we have as an adaptive a function from list sizes to sorting method adaptives.
type Size = Int

type ASort r = QTable Size Action r

asort :: Size -> [Int] -> ASort [Int]
asort n xs =
do let s = isqrt n
  q <- readTable
  let m = value q s
  t <- readTime
  ys <- case m of
    ISort -> isort n xs
    MSort -> msort n xs
  forceEval ys
  t' <- readTime
  modify ('adaptBy' (isqrt n,(m,t-t')))
  return ys

Adaptation sort takes as input a list xs and its size n, which is used to select the best sorting method for the list. First, the Q-table is read from the state using the function readTable, which is simply another name for the state monad function get that retrieves the state of the monad. The value of the adaptive Q-table is the function that maps sizes to sorting methods. Based on the selected sorting method m, which is obtained by applying the function value q to the integer square root of s, we either sort using insertion sort or mergesort. After forcing the evaluation of the result list ys, we adapt the Q-table using the monadic state updating function modify before returning the sorted list.

The recursively called sorting functions are also defined within the context of the monadic adaptive ASort since, at least msort, has to recursively sort sublists (of smaller size). That sorting task should be performed using the currently best method for those lists, and it should also adapt the information stored in the Q-table.
isort :: Size -> [Int] -> ASort [Int]
isort _ xs = return (foldr insert [] xs)

msort :: Size -> [Int] -> ASort [Int]
msort n xs =
  if n<2 then
    return xs
  else
    do let k = n \div\ 2
     us' <- asort k us
     vs' <- asort (n-k) vs
    return (merge compare us' vs')

In Section 4.4 we report some concrete timing results for this application, and we will present another application that is also based on recursive adaptation.

4.2.3 Transactional Adaptations

The adaptive pattern operations considered so far all progressed in a very fine-grained fashion, by tightly interwoven calls of \texttt{value} and \texttt{adapt}. Although these patterns seem natural there might be cases in which adaptation is less tightly controlled. For instance, it is often convenient for a multi-armed bandit to have several arm pulls per reward (\texttt{adapt}) call.

To illustrate this consider the following alternative representation of our multi-armed bandit, which stores in addition to the map the last pulled arm.

type ArmInfo a = (a,Pulls,Reward)
type PlayMap a = [ArmInfo a]

data Bandit a = Bandit a (PlayMap a)
In order to maintain this representation we have to use a different feedback type that distinguishes two kinds of feedback: either (a) an arm was pulled, in which case the corresponding pull counter is increased and the arm is remembered as the last one pulled, or (b) a reward for the last pulled arm is delivered, which will be added to the total reward of that arm in the map. These two different forms of feedback are captured in the following type.

```haskell
data Play a = Pull a | Reward Reward
```

This leads to a slightly different `Adaptive` instance definition than the one shown in Section 4.1.1.

```haskell
instance Eq a => Adaptive (Bandit a) where
    type Value (Bandit a) = a
    type Feedback (Bandit a) = Play a
    adapt (Pull a) (Bandit _ m) = Bandit a (incPulls a m)
    adapt (Reward r) (Bandit a m) = Bandit a (addReward r a m)
```

The function `incPulls` increments the number of pulls of the given arm in the map, `addReward` adds reward for a given arm. The definition of `value` remains unchanged and still uses the UCB algorithm previously described.

Now consider what happens if we want to implement a Rock-Paper-Scissors strategy on the basis of this representation and play it against some other strategy. The problem is that it now takes two adaptation steps, a Pull of an arm and a Reward for it, to make a meaningful adaptation transition in the sense of machine learning. Therefore, we need some form of “big-step” adaptation that can for this example be derived from the adaptive’s feedback as follows.

```haskell
bigStep :: Eq a => (a,Reward) -> Bandit a -> Bandit a
bigStep (x,r) b = b 'transBy' [Pull x,Reward r]

transBy :: Adaptive a => a -> [Feedback a] -> a
transBy = foldl adaptBy
```
The point to observe is that we have converted a value of type `Feedback a` into a function of type `a -> a`, which means that the big-step adaptation pattern that corresponds to `trainBy` takes a list of such functions instead of feedback values.

```
transformBy :: a -> [a -> a] -> [a]
transformBy = scanl (flip ($))
```

Consider, for example, an adaptation of the following form.

```
initStrat 'trainBy' xs
```

The corresponding adaptation for the changed adaptive could be implemented using `transformBy` in the following way. Here `stratB` is the initial bandit value, defined in the same way as `initStrat` for the new `Bandit` type.

```
stratB 'transformBy' map bigStep xs
```

As for `trainBy` we can also produce a big-step version of `coevolve` by generalizing the type of the argument functions. The result is a function that adapts two adaptives based on big-step adaptation parameter functions that have access to both current adaptives.

```
cotransform :: Adaptive (a,b) =>
    (a -> b -> a,b -> a -> b) -> (a,b) -> [(a,b)]
cotransform (f,g) (x,y) = (x,y) : cotransform (f,g) (f x y,g y x)
```

An example would be the definition of a Rock-Paper-Scissors tournament for adaptives as defined at the beginning of this section.

### 4.3 Monitoring Adaptation Behavior

The lifetime of adaptive programs can often be split into two major phases: (i) a *learning* or *adaptation phase* in which adaptives adapt (significantly) and (ii) a *stable phase* in which no or
only minor adaptations occur. It might be desirable, for example if we are training an adaptive
with predefined feedback, to be able to detect this transition.

To determine whether an adaptive program is stable requires us to monitor the adaptives. To
this end, we define a type `Monitor` and a corresponding function `monitor` to produce observa-
tions about the adaptation behavior.

```haskell
type Monitor a b = [a] -> b

monitor :: Adaptive a => Monitor a b -> [a] -> [b]
monitor m = map m . inits
```

The function `inits` produces the list of all prefixes of a given list.

Here is an example monitor that ensures that a particular property holds for the values of the
`k` last adaptives produced in an adaptation.

```haskell
ensureLast :: Adaptive a =>
            Int -> ([Value a] -> Bool) -> Monitor a Bool
ensureLast n p xs = length xs >= n &&
                    p . map value . take n . reverse $ xs
```

A very simple example property to monitor is whether all the values in a list are the same.

```haskell
allEq :: Eq a => [a] -> Bool
allEq [] = True
allEq (x:xs) = all (==x) xs
```

This property can be used to define a simple convergence criterion as follows.

```haskell
convergence :: (Adaptive a, Eq (Value a)) => Monitor a Bool
convergence = last 3 allEq
```

Using monitors we can define adaptation combinators that are controlled by the monitors.
until :: Adaptive a => [a] -> Monitor a Bool -> [a]
until xs = shiftMonitor ([],xs)

shiftMonitor :: ([a],[a]) -> Monitor a Bool -> [a]
shiftMonitor (xs,[]) m = if m xs then xs else []
shiftMonitor (xs,y:ys) m | m xs = xs |
                      otherwise = shiftMonitor (xs++[y],ys) m

With until we can now define self-controlling adaptations that adapt until a certain criterion,
such as convergence, is met.

As a concrete example, consider again the linear regression scenario. We can adapt a line \( l \)
using a list of points \( ps \) until the last two lines in the approximation sequence are close enough
together, that is, their difference in slope and intercept is smaller than a specific threshold.

\[(l \text{ 'trainBy' ps}) \text{ 'until' ensureLast 2 areClose}\]

areClose :: [Line] -> Bool
areClose [L m b,L n c] = max (abs (m-n)) (abs (b-c)) <= 0.001

4.4 Empirical Results

Here we present empirical results for the application of ABP to two well-known problems, RL
has been previously applied to: sorting [41] and budgeted optimization [62]. Our framework is
able to naturally capture both problems, allowing for most of the details of the adaptation process
to be hidden from the programmer.

4.4.1 Adaptive Sorting

Prior work [41] on adaptive sorting used RL to learn to choose between quicksort and insertion sort at each recursion point based on the length of the list. The learned program showed
small gains in average running time over pure quicksort and insertion sort. We implemented an adaptive sort using the structure shown in Section 4.2.2 to learn a mixed strategy of insertion sort and mergesort. We trained the algorithm on lists of integers of lengths up to 10000. The learned policy found a cutoff of just above 300: For lists smaller than that, insertion sort was faster, whereas for lists longer than the cutoff, mergesort was faster. Next, we tested our learned algorithm policies of just mergesort with no cutoff and with cutoffs off 10 and 1000. The learned algorithm was considerably faster than just mergesort with the other cutoffs we tested. For lists of size 10000, we see a speedup of between 1.6 and 2.6. Against mergesort with no cutoff, the learned hybrid algorithm is 20 times faster.

An important observation was that the cutoff learned only applies in the environment it was learned. That is, when we were learning the cutoff we were accessing the system timer and modifying our adaptives as we sorted lists. This overhead is necessarily included in the time we record to sort a sublist (in sort). But if we sort in an environment without this overhead, the learned cutoff was not as efficient as others. In fact, tests showed a very low cutoff (perhaps none) was fastest if there is no sampling overhead.

Whenever using time as a cost or reward, one must consider the fact that the timing observations influence the results. Although our adaptive framework is fast and efficient, the action being timed (sorting in our case) must be significant compared to this overhead. In this sorting domain, the time to sort a list was only significant for larger lists.

4.4.2 Adaptive Budgeted Optimization

Another application we now consider for ABP is in budgeted optimization. In this domain we have a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ and wish to find the value of $x$ that minimizes the “squared loss”

\[^2\text{We used a tree-based map as a contextual adaptive instead of functions for performance reasons.}\]
function $L(x) = |f(x)|^2$ (the square of the magnitude of $f(x)$). Furthermore, we are given a budget $b$ on the maximum number of times that we are allowed to evaluate $f$ during the optimization process. This situation of budgeted (or time-constrained) optimization occurs mostly due to real-time performance requirements (for example in computer vision and control problems).

A very common algorithm used for this problem is the Levenberg-Marquardt (LM) algorithm [44]. LM is an iterative optimization algorithm that starts at a random location $x_0$, and on each iteration evaluates the function at the current $x_i$ and computes a new $x_{i+1}$. LM uses a mixture of gradient descent and Gauss-Newton optimization to compute $x_{i+1}$. The details of this computation are not particularly important other than the fact that a key component of the algorithm is that each iteration must decide how to best blend gradient descent and Gauss-Newton, which is done by specifying a blending parameter $\lambda$. Marquardt [47] proposed a simple way to modify $\lambda$ by increasing $\lambda$ by a particular factor $\eta$ (larger values of $\lambda$ put more weight on gradient descent), when the previous iteration increased the loss, and decreasing $\lambda$ otherwise by dividing $\lambda$ by $\eta$ (giving more weight to Gauss-Newton). This strategy works well and can be found in most implementations.

Figure 4.1 illustrates an example of the LM algorithm on an example function called the Rosenbrock function.

In [62] the authors apply reinforcement learning (RL) methods to learn a controller for $\lambda$ and showed that it is possible to obtain a small improvement in the minimization of loss compared to the standard $\lambda$ control from LM. In [11] we presented an adaptive program similar to the RL described approach to illustrate our ABP framework. Below we discuss some details about a variation of that adaptive program below.

First, we represent the algorithm explicitly as a trace of gradient descent steps through a set of points.
Figure 4.1: An example of the LM algorithm using the Rosenbrock function (the true minimum value is \((1, 1)\)) as an example function. The algorithm starts at any point \(((0.5, -0.5)\) in this example). Then at each point it computes the gradient and steps in that direction by a function of \(\lambda\). If a step “overshoots”, it backs up and adjusts the \(\lambda\) parameter as in the 4\(^{th}\) sample in this example.

```
type Vector = [Double]  -- x
type Loss = Double      -- |f(x)|^2
type Point = (Vector, Loss) -- (x,|f(x)|^2)
type Lambda = Double

type Trace = [TraceElem]
data TraceElem = TE {
  teIn :: Point,  teStep :: Point,
  ...
  teOut :: Point,  teLamOut :: Lambda }
```

A Point groups an input vector \(x\) with the computed loss at that point \(|f(x)|^2\). A Trace is a list of descent steps the LM algorithm takes. Some of the following elements included in a trace are the following.

- The point the algorithm starts at \(teIn\).
- The point the gradient descent steps to \(teStep\).
• The output point \( teOut \) chosen and adjusted \( \lambda \) parameter \( tcLamOut \). This will be some function of the input point and step point (one or the other). For instance, in standard LM, if the new estimate is an improvement (a smaller loss), this will be \( teStep \), and \( teIn \) otherwise.

• The adjusted value for \( \lambda \), represented as \( tcLamOut \).

Using these trace data one might start the implementation of the LM algorithm with the program below.

\[
\text{lma :: Function -> Point -> Lambda -> Trace}
\]

\[
lma f (x,fx) \lambda = te : lma' f (x',fx') \lambda'
\]

\[
\quad \text{where } (xStp,fxStp) = \text{step } f \ x \ \lambda
\]

\[
\quad \begin{array}{l}
\quad (x',\lambda') \quad -- \text{Fixed LM logic} \\
\quad | fxStp < fx = (xStp, \lambda/\eta) \\
\quad | \text{otherwise} = (x, \lambda/\eta)
\end{array}
\]

\[
f'at'x' = f'at'x'
\]

\[
te = \text{TE } (x,fx) \lambda (xStp,fxStp) \ldots \ldots (x',f'at'x') \lambda'
\]

The \text{step} function steps in the direction of the gradient using the \( \lambda \) parameter. The LM logic is the selection of the next \( x \) value and \( \lambda \) for the next iteration, represented by \( x' \) and \( \lambda' \). The \text{Function} data type is just an abstract type representing the function we are minimizing. The \text{at} function will evaluate the function at an given point and return loss of the output vector.

This \text{lma} algorithm lazily generates an infinite list of trace elements (we will see why later). Hence, to implement a budgeted optimization with \( b = 5 \) and with an initial \( \lambda = 2 \) one must generate a list of \( b \) elements and access the last \( x \) value of that prefix as in the code below.

\[
\text{lma5 :: Function -> Vector -> Point}
\]

\[
lma5 f \ x0 = teOut . \text{head} . \text{drop 5 } \$ \ lma f (x0,f'at'x0) \ 2
\]

As stated before, our goal is to replace the fixed logic in selecting the new \( \lambda \) and \( x \) values. For instance, if the last steps have all been successful, maybe we should retain the bad sample instead of discarding it. To make that logic programmable we define the following data types.
type Strategy = History -> Move
type History = [Estimate]
data Estimate = G | B

da strategy is any function that selects a method (we call it a Move) for adjusting $\lambda$ and the new $x$ value based on whether the loss decreased over the last few iterations (hence a list of estimates). The value $G$ means the previous step was “good” and decreased the loss, while the value $B$ means the loss increased in the last step. We encode various strategies for making this change as the Move data type as shown below.

data Move = KeepKeep | KeepDiv | KeepMul
           | DiscDiv | DiscMul | ...

The meaning of these values indicate how we select $x$ and adjust $\lambda$ respectively. A formal definition is given below with the adjust function.\(^3\)

\[
\text{adjust} :: \text{Move} \rightarrow \text{Lambda} \rightarrow \text{Vector} \rightarrow \text{Vector} \rightarrow (\text{Vector},\text{Lambda})
\]

\[
\text{adjust} \ m \ \text{lam} \ x \ \text{xs} =
\begin{align*}
\text{case} \ m \ \text{of} \\
\text{DiscDiv} & \rightarrow (x,\text{lamb}/\eta) \\
\text{DiscMul} & \rightarrow (x,\text{lamb}*\eta) \\
\text{KeepKeep} & \rightarrow (x,\text{lamb}) \\
\text{KeepDiv} & \rightarrow (x,\text{lamb}/\eta) \\
\text{KeepMul} & \rightarrow (x,\text{lamb}*\eta)
\end{align*}
\]

With these definitions, one could represent the standard LM strategy as follows.

\[
\text{stdLMA :: Strategy} \\
\text{stdLMA} \ (G:_:_) = \text{KeepDiv} \\
\text{stdLMA} \ (B:_:_) = \text{DiscMul}
\]

Next, we must extend the lma function to make it accept this programmable strategy and keep track of the history of previous estimates with the History data type.

\(^3\)The action DiscKeep is notably absent since it does not really make sense to discard the new $x$ value and retain the old $\lambda$ since that leads to an immediate loop (i.e. we would reach the exact same $x$ the next iteration).
alma :: Strategy -> Function -> Point -> Lambda -> History -> Trace

alma strat f (x,fx) lam es = te : alma strat f (x',fx') lam' ctx
  where (xStp,fxStp) = step f x lam
       e = if fxStp < fx then G else B : es
       ctx = e : es
       a = strat ctx
       -- LM step: choose new (x,lam)
       (x',lam') = adjust a lam x xStp
       te = TE (x,fx) lam (xStp,fxStp) (e:es) a (x',f'at'x') lam'

The above function represents the natural extension to make the LM strategy “programmable” and adaptive. Besides the standard LM stdLMA strategy, we can easily construct other functions and test various strategies very quickly. Most importantly, we can now define an adaptive function out of ABP primitives. We can use a contextual multi-armed bandit with context type History to define an adaptive strategy as follows.

  type AdStrat = History -> AdVal Move

An AdVal is just a more sophisticated Bandit (multi-armed bandit).

We can covert an adaptive strategy into a fixed strategy with the function shown below.

  toStrategy :: AdStrat -> Strategy
toStrategy as = \c -> value (as c)

Much like in the previous section where we introduced contextual adaptives, one can even simplify this further to (value _).

The final piece missing in this puzzle is a method of feedback to allow our adaptive function to improve. We desire to minimize the loss; hence, that feature is an appropriate starting point for the feedback. However, in general we do not know anything about the function being minimized, specifically its optimal x-value and the scale of its range are unknown. To address this, we normalize the reduction in loss by the last value in the sequence.\footnote{With this scheme all test functions must have a similar minimum loss value (typically 0).} Formally, we define the scaled
reduction in loss (SLR) as the reduction in loss divided by the initial loss $L_0$.

$$SRL = (L_0 - L_b)/L_0$$

If the optimal loss is 0, then this value gives a nice relative estimate of the percentage improvement after taking $b$ steps of the LM algorithm.

Since the adaptive LM algorithm returns a Trace of elements, we can use this information to reconstruct the History and update the AdStrat separate from outside the alma function. To make the adaptive strategy learn over one problem for a given input vector we might write the following.

```haskell
learnOnce :: Function -> Budget -> AdStrat -> Vector -> AdStrat
learnOnce f b as x0 = as'
  where trc = take b (alma (toStrategy as) f x0)
       fxB = snd (teOut (last trc))
       srl fx = (fx - fxB) / fx
       as' = adaptStrat srl trc as
```

The learning cycle converts the adaptive strategy to a fixed strategy and calls the adaptive LM algorithm. From this we extract the last loss value found $fxB$. Given this we can define the $srl$ function for this sequence. For any element in the sequence $srl$ will returned the scaled loss in reduction as defined above for that. The adaptStrat function walks through the trace updating each adaptive.

Our specific learning method for this problem is a “principled update” method that assumes a (loose) optimal subproblem property. Specifically, we only update the adaptive values corresponding to histories of this problem for that smallest $b$ value (the leaf-level histories) first. After a set number of steps adaptives that depend on those “smaller” ones adapt. For example, if our sequence of gradient descents contains a bad estimate followed by a good estimate ($[B, G]$), we only adapt the $[B]$ estimate after $[B, G]$ has had sufficient chance at being adapted.
This scheme gives us an update strategy similar to a dynamic programming approach in that we solve smaller problems first and use those solutions to solve larger problems, but different in that smaller values can keep adapting as higher-level ones attempt different strategies.

This principled strategy is implemented adaptStrat as shown below.

```haskell
adaptStrat :: (Loss -> Double) -> Trace -> AdStrat -> AdStrat
adaptStrat srl t0 as0 = adStrat t0 [] as0
    where adStrat :: Trace -> History -> AdStrat -> AdStrat
        adStrat [] _ as = as
        adStrat (te:trc) pvctx as
            | null trc = as'
            | avFullyExplored (as nctx) = adStrat tes ctx as'
            | otherwise = adStrat tes ctx as
            where fx = snd (teIn te)
              ctx = est te : pvctx
              est te
                | snd (teStep te) < snd (teIn te) = G
                | otherwise = B
              nctx = est (head tes) : ctx
              as' = adapt (ctx,-(srl fx)) as
```

The main function calls a recursive helper function to walk through the trace elements. For each element in a history, there are three cases, which we discuss (in respective order as defined above).

- If we are at the end of the trace (e.g. the history is [G] or [B]), then this is a leaf case, we can update the adaptive value.

- If the successor context nctx is known (avFullyExplored determines this), then we can update this adaptive. Furthermore, we adapt the rest of the histories in the subproblems as well.
Finally, if the child state is not known, we leave the adaptive unchanged, but continue adapting the smaller adaptives.

This manual updating scheme illustrates an interesting approach that seems to work fairly well on this problem.

The adaptive values AdVals know nothing of each other or this principled learning ordering relation imposed on them by the larger program. We trade away automation in updates for increased control over the feedback mechanisms. This allows us great flexibility in testing and programming specialized learning strategies. However, this customization step is required of us for all cases.

In this version of ABP, an AdVal has no way of being able to see or update itself since it does not have access to the future state information. In fact, AdVals (or Bandits) lack any notion of future state. Any chain of updates has to be handled by an external (parent) process that manages the learning process globally. For example, the adaptive corresponding to context \([G]\) has no innate way of accessing the adaptive from context \([G,B]\). Hence, the adaptive program using ABP has to explicitly provide that logic.

4.4.2.1 Results

We tested this approach on three different functions: (1) Rosenbrock, (2) The Helical Valley, and (3) Brown and Dennis functions. These functions are all popular choices in function optimization benchmarks [53]. One problem instance consists of a function from the above set and an appropriate random start vector from that function’s domain. Over a long sequence of random problem \((10^5)\) instances we adapted an initially empty random strategy. At regular intervals we evaluated the strategy on a different set of problem instances and compared it to the standard LM algorithm on the same problems. Our metric for success is the average scaled reduction in loss.
(ASRL).

\[ ASRL = \frac{1}{N} \sum_{i=1}^{N} \frac{(L_{0i} - L_{Ri})}{L_{0i}} \]

On the Rosenbrock and Helical Valley functions, the learned strategy achieved about a 2 and 4 percent improvement over the standard LM. With initial good estimates, the policy was very similar to standard LM and the best option was to keep the new estimate and divide \( \lambda \) (the `KeepDiv` action). However, the learned policy also found utility in keeping bad estimates and dividing \( \lambda \) as well. This possibly implies that the initial \( \lambda \) value of 2 was too small for such a small budget. However, there were some cases where the adaptive strategy also chose to keep bad estimates (while adjusting \( \lambda \)).

On the Brown and Dennis function the adaptive strategy did not perform as well and saw a 3 percent decrease in performance. The adaptive strategy tended to make \( \lambda \) larger (multiplying it) or keeping it unchanged after taking a good descent step where the standard LM strategy would divide it. Probably the principled adaptation approach used did not allow the learning algorithm to sufficiently explore the space of strategies.

### 4.5 Summary

This chapter represents initial work in understanding adaptive programs and exploring the ideas behind them. We presented a generic embedded DSL in Haskell for describing adaptive computations based around the concept of adaptive values (“adaptives”). This work sets the foundation for our understanding of adaptive programs and adaptive values and how they are expressed and used.
The approach is centered around an \texttt{Adaptive} type class, which represents a generic interface for adaptive values. It abstracts the underlying representation of different adaptive values by providing access to the current value through a \texttt{value} method and defines an interface for adaptation through an \texttt{adapt} operation. With just those two simple operations, we are able to define generic adaptive patterns including adaptive instances for various predefined data structures "derived adaptives" (e.g. pairs of adaptives are an adaptive). One of the most important of these derived patterns is the contextual adaptive, which is an adaptive function that looks up an adaptive based on a recognition context argument. Finally, higher-level combinator functions such as \texttt{train}, \texttt{evolve}, and others allow us to operate on adaptives easily by abstracting common patterns in adaptive programs.

This combination of a few carefully chosen generic adaptive structures and combinators promotes a small, but compositional DSL. For instance, a pair (tuple) of adaptives is also an adaptive. Hence, one can easily nest pairs of pairs of adaptives and still have an adaptive. Combinators such as \texttt{coevolve} allow us to quickly define interactions between these compositions of adaptive values.

One very useful pattern we define is the multi-armed bandit. This pattern allows us to gracefully handle points of uncertainty in programs where we can select from a small set of options and indicate success with relative numeric \texttt{Reward} values. In our case studies in Section 4.4 we heavily use multi-armed bandits in contextual adaptives. In fact, this pattern is so valuable that later chapters focus on extensions of it.

One disadvantage this approach runs into is that rewards have to explicitly be calculated by the adaptive program and then manually fed back to all adaptives by the user. In general a user writing an adaptive program should not have to keep track of all these prior decisions made throughout the execution and explicitly dole out rewards or punishments to the various adaptives.
To illustrate the difficulty of feeding back rewards manually, consider an adaptive program in a grid world searching for some goal. Each time the program enters a square \((x, y)\), it chooses a move. Once the goal is found many steps later we must go back and explicitly reward each adaptive value along the way. This is certainly annoying extra work and might be intractable for long paths. A better approach would be to reward the system as a whole and let the learning algorithm filter the reward back to each decision automatically. The next chapter addresses this problem with a different approach.

The highly tunable reward scheme of this approach does hold an obvious advantage in cases where an expert desires to explore different learning algorithms. Recall the adaptive function minimization case study in Section 4.4.2. The programmable aspects of our learning algorithm proved quite useful; we were allowed to easily and rapidly explore different learning algorithms without changing the ABP library. It is quite reasonable that other learning algorithm experts might also find this useful. However, the disadvantage of this scheme is that the programmer must deal with that aspect explicitly. It would be nice for the programmer to be able to specify one reward for many choices and have that reward filter back automatically to the choices that influenced that reward.

This initial approach does not model adaptive programs as any specific formal system from reinforcement learning such as an MDP. Instead it allows one to combine smaller models such as multi-armed bandits however they see fit. This makes it difficult to provide guarantees of program convergence to optimal values. In contrast, our next chapter’s approach fixes this to a single model (and MDP) and uses a single adaptation strategy thus simplifying the library’s use without significantly reducing the class of problems that can be solved effectively with ABP.

In summary, this approach exposes the internals of the learning algorithms and feedback methodologies explicitly to the user. When the user needs to program those specific aspects, this is beneficial; otherwise, it can be a burden. Moreover, this does imply the utility of a higher-level
framework to fix a few complex update patterns. For example, it might make sense to define a set of functions and data types for problems that need Q-learning-type solutions and have a fixed number of choice points. Such a library could abstract away the work of tracing the algorithm execution and feeding back rewards internally.

Finally there are several other things this chapter’s work demonstrates.

- The notion of an adaptive value is very key. Adaptive programs can be represented simply as programs with adaptive values in them. (In this language the concept was modeled via the Adaptive type class.)

- One can draw on machine learning and reinforcement learning successfully to solve many very simple partial programs. We found this particularly the case in developing patterns for adapting adaptive values such as in the trainBy, evolve, and coevolve combinators. The highly customizable Adaptive type class favors programmers that might want to tune or tweak their learning algorithm for some specific case.

- Domains involving raw speed as the reward signal are very difficult to evaluate in high-level languages such as Haskell because of the level of abstraction between program representation and execution. Other work in this field always makes an effort to perform learning in a pre-processor step such as the systems for automated configuration we discussed in Section 3.4 (in the literature review), and those systems typically operate in very low-level languages to minimize the effects of abstractions such as garbage collection on the performance.

- Tried and true algorithms such as a tuned mergesort or the Levenberg-Marquardt algorithm leave little room for general improvement through reinforcement learning. Additionally, this conclusion is supported by similar results such as [62].
Chapter 5: Object-Oriented Adaptation-Based Programming

This chapter considers an object-oriented implementation of ABP as a library for the Java [6] programming language from some earlier work [10]. The previous chapter explored some possible meanings for adaptive programs and defined the notion of adaptive values and ways of composing them. Guided by some of the observations from that work, this chapter presents a slightly different form of adaptive programming tuned for a slightly different set of problems.

One thing that became clear from the work detailed in the previous chapter was that in all the large problem domains considered numeric feedback was appropriate. This suggests that we should specialize (and simplify) the Feedback component of adaptive variables to a numeric type. Moreover, those examples all used reinforcement learning algorithms in our large applications (adaptive sorting and adaptive function optimization) with considerable success. Hence, it makes sense to expand the role of those algorithms by introducing adaptive program constructs that support more sophisticated learning algorithms.

The previous chapter’s approach required the user to manually adapt each adaptive (i.e. link rewards back for each adaptive used in producing that reward). In contrast this chapter’s approach involves recording a trace of all the choices made (adaptive variables used) and rewards received, and then automatically adapting those adaptives. By taking this task out of the hands of the user, ABP is easier to use.

The decisions an adaptive program makes at any given moment greatly influence the future states it reaches. Hence, it makes sense to model the program trace as the interconnected sequence of choices that the adaptive program made. This allows us to treat the adaptive program
as Markov Decision Processes (recall MDPs from Chapter 2) and solve them with algorithms such as Q-learning [75] or SARSA [61].

Also recall that our previous approach forced all state management onto the shoulders of the user. For instance, in the adaptive sorting and adaptive function optimization programs, the Q-table of bandits had to be explicitly maintained by the user. However, in this approach we will encapsulate all that state information internally in the library’s representation of the adaptive values.

The rest of the chapter is outlined as follows. Section 5.1 provides some implementation details about the example scenario to set the stage for the application of ABP. Section 5.2 illustrates how to realize the application using our ABP library. In addition, it will also use the example to motivate the design of the library components. Section 5.3 discusses some additional aspects regarding the programmer’s control over the machine learning process. In particular, this section shows how a programmer can improve (that is, speed up) the adaptation process by coding insights about the domain. Finally, Section 5.4 presents some real-world uses of this ABP library, including example uses in strategy game controllers as well as automated randomized program testing.

5.1 Example Scenario

To illustrate this approach suppose we are implementing a simple hunting simulation (illustrated in Figure 5.1 involving two wolves hunting a rabbit. The goal is to write a program where the wolves work together to trap the rabbit.

The game world’s state is represented by a two-dimensional grid as in the class given below.
This class maintains the coordinate positions of each animal on the grid and implements game logic and rules. We omit some of the details that are not important for the following discussion. Changes to the state are made via three move methods `moveWolf1`, `moveWolf2`, and `moveRabbit`. Each of these methods returns a new `Grid` with the move applied. Additionally, a method `rabbitCaught` is given that checks to see if the rabbit has been captured in the current game grid. The static constant `INITIAL` corresponds to the instance of the world with the wolves at the top-left and the rabbit at the bottom right. This is used for the game’s initial configuration.

Permissible moves for each animal are described via a `Move` enumeration.

![Figure 5.1: This illustrates the wolf hunting problem on a 3 × 4 grid. The x-coordinate wraps, and the wolves must work together to trap the rabbit.](image)
enum Move {
    STAY, LEFT, RIGHT, UP, DOWN;
    static final Set<Move> SET = unmodifiableSet(EnumSet.allOf(Move.class));
}

The Rabbit class given below defines a basic interface for any number of fixed strategies that the rabbit may take.

abstract class Rabbit {
    abstract Move pickMove(Grid g);
    static Rabbit random();
}

The static method random returns an implementation of a rabbit strategy that moves randomly.

A game proceeds as follows. First, the rabbit may observe the location of the wolves and then move one square in any direction. Next, the first wolf gets to observe the rabbit’s move and move itself. After that the second wolf gets to observe both prior moves and then move itself. If at the end of a round either wolf has landed on the rabbit’s square, the rabbit is captured. Otherwise, the hunt continues.

To make the problem more interesting we allow the x-coordinate of the world to wrap. Hence an animal at the left end of the grid can wrap around to the right end and vice versa. In this way, the rabbit could always escape if the wolves close from the same direction. Hence, the wolves must be smart enough to cooperate and close from opposite directions.

A programmer solving this problem with the above definitions might initially sketch out pseudocode such as that given below.
Rabbit rabbit = Rabbit.random();
Grid g0 = Grid.INITIAL;
while (true) {
    Move rabbitMove = rabbit.pickMove(g0);
    Grid g1 = g0.moveRabbit(rabbitMove);

    // ... pick move for wolf 1
    Move wolf1Move = ???
    Grid g2 = g1.moveWolf1(wolf1Move);

    // ... pick move for wolf 2
    Move wolf2Move = ???
    Grid g3 = g2.moveWolf2(wolf2Move);

    if (g3.rabbitCaught()) {
        ... raise our score a large amount
        break;
    }

    g0 = g3;
    ... lower our score a small amount
}

The grid g0 represents the world state at the beginning of the loop each iteration. Again this includes the location of each animal. We move each animal as described before in the rules. The grids g1, g2, and g3 correspond to the game state after each animal’s move.

For now, we are uncertain about how to select the wolf moves, so we indicate that with question marks. Near the end of the loop, we check to see if the rabbit has been captured. If not, we reassign g0 and perform another round.

We can make some observations on this pseudocode.

- There is a natural sense of constrained uncertainty when our wolves select their moves.
  They have to pick one of a small finite set of moves.
The score is a reward or indicator of success or failure. Hence, it can be used as a metric telling us how successful our wolves are.

There is an inherent dependency between both wolves’ strategy. They must work together. Moreover, the reward or score applies to both as they share a common goal.

5.2 Adaptation Concepts

The coupling between choice and reward suggests that some sort of abstraction could automatically select sequences of moves and then evaluate those moves by checking the score. Over time, this abstraction could identify better and better sequences of choices so as to optimize their average reward. Implementing these notions is the goal of our ABP library, which we now describe.

In this library an adaptive variable (adaptive for short) as one of these points of uncertainty in a program where the programmer must make some decision amongst a small discrete set of choices. A value generated by one of these variables is called an adaptive value. The location of this uncertain selection a choice point.

An adaptive can suggest a potential action at a choice point. But in order to do so, it requires some unique descriptor that identifies the state of the world. In the above example, there are two points of uncertainty, namely the wolf move selection indicated with ??? in the code. In this library we represent adaptive variables via the Adaptive class shown below.

```java
public class Adaptive<C,A> {
    public A suggest(C context, Set<A> actions);
}
```

Adaptive values are parameterized by two type variables (using Java Generics [16]). The first (C) corresponds to the context or world state, and the second (A) corresponds to the type of
permissible actions that the adaptive can take. In the hunting example the context type argument \( C \) is instantiated to be the \texttt{Grid} class since that class neatly contains all the necessary information. The type parameter \( C \) is the subset of program state that an adaptive variable depends on. The process of converting that state into whatever type argument is bound to \( C \) helps make the state abstraction problem type-safe and explicit.

The \texttt{suggest} method is our way of asking the adaptive for an appropriate value for some context (instance of a \( C \)). (This is similar to the \texttt{value} method of an adaptive from the Haskell DSL presented in the previous chapter or the \texttt{choose} function in the simple imperative ABP implementation presented in Chapter 2.) Moreover, the caller passes a set of permissible actions for the adaptive to choose from. We are asking the adaptive value, “If the state of the world is context, what is a good move from the set of actions?” Explicitly passing a context argument for every adaptive value solves the state-abstraction problem by allowing the user to explicitly define it. Since this process is made explicit, this approach is similar to ALISP’s [4] optional argument to the \texttt{action} macro or the \texttt{state} clauses in \texttt{A^2BL} [64].

In the wolf hunt example, each of the wolves could be represented by an adaptive. The context type parameter would be the world state \texttt{Grid}, and the action type would simply be a \texttt{Move}. We illustrate this shortly.

The dependency between the moves that we select for our wolves elicits another important observation: multiple adaptives might share a common goal. Under this view our adaptive wolves must share a common reward stream. The score in the game (the reward) applies to both wolves, not just one. This sharing of rewards asks for a scoping mechanism that allows the grouping of multiple adaptive variables.

We define this common goal as an \textit{adaptive process}. It represents a goal that all its adaptives share, it distributes rewards (or penalties) to its adaptives, and it manages various history information that its adaptive variables learn from.
A new adaptive process is created via the class’s `init` method. The source file argument permits the `AdaptiveProcess` to automatically be persisted between runs. The first time the program is run, a new file is created to save all information about adaptives. When the program terminates, the process and all its adaptives are automatically saved. Successive program invocations will reload the learning process information from the given file. This persistence permits the adaptive variables contained in the process to evolve more effective strategies over multiple program runs.

Individual adaptives are created with the `initAdaptive` method. The context and action type parameters are passed in as arguments, typically as class literals. This permits the library to dynamically type check persisted data as it is loaded.

In our wolf example we would use the following code to initialize our process and the adaptives for each wolf.

```java
public class Hunt {
    public static void main(String[] args) {
        AdaptiveProcess
            hunt = AdaptiveProcess.init(new File(args[0]));
        Adaptive<Grid, Move>
            w1 = hunt.initAdaptive(Grid.class, Move.class),
            w2 = hunt.initAdaptive(Grid.class, Move.class);
        ...
    }
}
```
We specify the aforementioned rewards of an adaptive process with the `reward` method. Positive values indicate positive feedback and tell the process that good choices were recently made, negative values indicate bad choices were recently made.

Upon receiving a reward, the adaptive process will consider the previous actions it has taken and adjust its view of the world accordingly. This permits later calls to `suggest` to generate better decisions. We discuss the final method of `AdaptiveProcess.disableLearning`, later.

Continuing with the previous block of code, the body of this game sketched out earlier in pseudo code could be implemented as follows.
Rabbit rabbit = Rabbit.random();
Grid g0 = Grid.INITIAL;
while (true) {
    Move rabbitMove = rabbit.pickMove(g0);
    Grid g1 = g0.moveRabbit(rabbitMove);

    // ... pick move for wolf 1
    Move wolf1Move = w1.suggest(g1,Move.SET);
    Grid g2 = g1.moveWolf1(wolf1Move);

    // ... pick move for wolf 2
    Move wolf2Move = w2.suggest(g2,Move.SET);
    Grid g3 = g2.moveWolf2(wolf2Move);

    if (g3.rabbitCaught()) {
        // ... raise our score a large amount
        hunt.reward(CATCH_REWARD);
        break;
    }

    g0 = g3;
    // ... lower our score a small amount
    hunt.reward(MOVE_PENALTY);
}

The interesting pieces of this program are those near the comments where we filled in adaptive
code. We now discuss those parts here. First, the wolf movement strategies are as simple as calls
to the adaptive variables’ suggest methods. In each case, we pass in the current game state
(the Grid) and the set of all moves to choose from Move.SET. Note that every move is legal; we
translate moves into walls (the y-axis does not wrap) into Move.STAY for simplicity.
Second, wherever we referred to scores earlier, we now place calls to the `reward` method of the adaptive process representing our hunting goal. After each unsuccessful round we assess a small penalty `MOVE_PENALTY`. Once the rabbit is captured we reward a large amount `CATCH_REWARD`. In our example we use the values $-1$ and $1000$, respectively.

If we run the game over multiple iterations, the average number of moves for the wolves to catch the rabbit drops fairly quickly. After a few thousand runs the average is between 4 and 6 moves. With grid dimensions of $3 \times 4$ from the initial state, the worst-case optimal solution is 4 moves. That is, if the rabbit stays as far away from the wolves as possible each round, it will take up to 4 moves to capture it. With a random rabbit, we should expect smaller averages.

The reason for this initial discrepancy is two-fold. First, a few thousand iterations is not a long time for the adaptive process to learn an optimal strategy; indeed after a few thousand more games, the average drops lower. Second, and more importantly, even once an optimal strategy is found, the adaptive process will continue to search for better ones and may try suboptimal strategies. An initially bad looking move might lead to a better overall solution. Hence, it is necessary for the adaptive process to operate suboptimally as its adaptive variables try various move sequences.

Recall the `disableLearning` method of `AdaptiveProcess` whose description we deferred earlier. This method tells the adaptive process to suspend its search through suboptimal values and use only the best moves for every given world state. This is a means for the programmer to indicate to the adaptive process that it should stop searching and work deterministically with the knowledge it currently has. During practice an ABP program explores new strategies, but during the big game it uses what it knows works best. If we transition to this optimal mode and run the wolf hunt program shown before, the average number of moves to catch the rabbit drops to around five after just a few thousand rounds.
In Figure 5.2 the plot titled “Full Move Set” shows the average number of rounds necessary to catch the rabbit as a function of the number of games learned when playing in this optimal mode. The second plot is discussed below.

5.3 Constraining Adaptation Behavior

In some contexts where an adaptive variable must suggest a value only some actions are permissible or desirable, and the others are invalid and may result in an unsound or inefficient program. For instance, if we are in a maze, moving into a wall may not be a permissible action. Passing a
constant set of actions to suggest would require the algorithm to explicitly define some meaning for those invalid actions. ABP solves this problem by allowing the programmer to vary the action set at run-time by passing it into the suggest method (the second argument).

Moreover, there may be situations where an ABP user can make observations statically (at development time), which simplify the learning problem in some way. For example, suppose that a programmer implementing the wolf hunt game observed that any successful strategy would require the two wolves to move in opposite directions, that is, close in on the rabbit from each side. In that case, we could easily specify this additional knowledge in the partial program by passing in a subset of the permissible moves. We illustrate this by showing a slice of the earlier code inside the game loop.

```java
// ... pick move for wolf 1
Move wolf1Mv = w1.suggest(g1,setOf(LEFT,DOWN,STAY));
Grid g2 = g1.moveWolf1(wolf1Mv);
// ... pick move for wolf 2
Move wolf2Mv = w2.suggest(g2,setOf(RIGHT,DOWN,STAY));
Grid g3 = g2.moveWolf2(wolf2Mv);
```

The above code only allows one wolf to move left and one to move right.

Constraining the action set for each wolf gives the more efficient “Constrained Move Set” plot shown in Figure 5.2. (The setOf function just wraps a list into a Set.) This constrained learning simplifies the learning algorithm’s since it must consider fewer possible solutions and thus results in faster adaptation.

5.4 Evaluation

Section 5.1 demonstrated positive experimental results on our example Wolf-Rabbit program. It was shown that through the use of this ABP library the wolves could effectively learn to capture
the rabbit. However, this ABP library makes has been used successfully by researchers in different domains such as games [10], code coverage testing [29, 31, 30], and in optimizing network controllers [79]. This section describes several of these uses in more detail, and illustrates how the authors made use of ABP in those non-trivial problems.

5.4.1 Yahtzee

![Yahtzee game screenshot](image)

Figure 5.3: An example Yahtzee game. The possible categories are on the left showing only two filled. The current dice are on the right. Here the player is choosing to re-roll one of the 5 dice with the hope of getting a 4 and getting a large straight (1, 2, 3, 4, 5).
Yahtzee is a well-known dice game where players roll five dice, select a subset to re-roll twice, and then apply the final combination to some specific category that those numbers suffice. For instance, if we initially roll (4, 5, 6, 6, 6), we may apply this roll to the three-of-a-kind category since there are three of the same number. However, we can also apply this roll to the sixes category, which generates a score proportional to the number of sixes. Finally, if we have any re-rolls left over, we might choose to re-roll the 4 and 5 in hopes of getting all five 6’s and filling the yahtzee category, which generates a larger score. Figure 5.3 illustrates an example Yahtzee game a Yahtzee clone called Jahtzee by James D. Gutholm.

Categories may only be filled once, so players must be careful deciding when to fill them. Filling a category in an early round runs the risk of missing a better opportunity later. For instance, filling the three-of-a-kind category with a roll containing three 1’s generates a lower score than filling it with three 6’s. However, waiting too late risks being unable to fill a category and receiving no score for it.

The decision of which category to try for when selecting dice to roll given the open categories and current dice is the deepest and most mentally challenging task a Yahtzee player must make. Once we make that decision, choosing the dice to re-roll is rather simple.

We illustrate below how ABP elegantly encodes this problem.
void yahtzeeRound(AdaptiveProcess player,
    Adaptive<GameCtx,Category> c1,
    Adaptive<GameCtx,Category> c2,
    GameState s1)
{
    Set<Category> categories = s1.getEmptyCategories();
    Category cat1 = c1.suggest(getCtx(s1),categories);
    GameState s2 = rollFor(cat1,s1);
    Category cat2 = c2.suggest(getCtx(s2),categories);
    GameState s3 = rollFor(cat2,s2);
    //out of rolls here
    GameState s4 = assignBest(s3);
    player.reward(s4.score - s3.score);
    return s4;
}

For each of the 13 rounds the Yahtzee player asks ABP to suggest a category given the current roll and empty categories getCtx(s1). This category is passed as an argument to rollFor, which selects the best dice to re-roll given the our goal. This illustrates part of the beauty of partial programming.

This contextual information for each choice is encoded in a GameCtx class which is a subset of the current game state GameState. Categories are represented via a Category enumeration.

The reward at the end of this is simply the number of points we receive in this round by applying the current dice to the best empty category, a simple greedy algorithm implemented by assignBest.

In [10] Pinto developed the above program and trained it by letting it play several million games over the course of about half an hour. The results are shown in Table 5.1 and demonstrate the reasonably quick improvement. Before any learning the adaptive program will randomly
guess and achieve about 119. However, after learning, the adaptive program achieved about a score of 195. Since the game is random, we average these numbers over 1000 trials.

Additionally, the table also shows our program’s performance compared to a state-of-the-art Monte-Carlo planning algorithm called UCT [39]. UCT is an “anytime algorithm” meaning it can be run as long as one desires generally improving results with more time. Pinto tested this against a fast and slower variant of this algorithm and illustrated that the trained ABP program almost reaches score-partity with the more complicated UCT algorithm albeit at a fraction of the running time.

More important to us than the timing comparison though is the score comparison. With very little programming effort we were able to use our adaptive library to achieve competitive results compared to a state-of-the-art planning algorithm. Moreover, our approach uses little more insight than the game rules and is more akin as to something performed by a casual programmer. Using the insights available to domain experts perhaps a slightly more complicated adaptive program could surpass the expert human and near the optimal value.

<table>
<thead>
<tr>
<th>Program</th>
<th>Average Score</th>
<th>Avg. Game Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABP (before learning)</td>
<td>119</td>
<td>0.001</td>
</tr>
<tr>
<td>ABP (after learning)</td>
<td>195</td>
<td>0.001</td>
</tr>
<tr>
<td>UCT-fast</td>
<td>161</td>
<td>0.8</td>
</tr>
<tr>
<td>UCT-slow</td>
<td>208</td>
<td>152.0</td>
</tr>
<tr>
<td>Optimal [28]</td>
<td>254</td>
<td>N/A</td>
</tr>
<tr>
<td>Average Human</td>
<td>~ 220</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 5.1: Performance on Yahtzee. All results are averaged over 1000 games.

Throughout the learning process Pinto sampled the rate of improvement of the program to illustrate how it was learning at various intervals. Figure 5.4 shows this learning curve and illustrates how quickly it improved (after only a few games).
The relevant state information for a context includes the set of open categories and the current dice information. However, since the learning algorithm must experiment with and record all possible contexts, this state space is intractably large ($2^{13}$ possible sets of open categories times the 252 of possible rolls of five dice results in over $2 \times 10^6$ unique contexts). To solve this we simplified the state space by using the number of open categories instead of the actual set (13 possible values times the set of rolls just under 3300 unique contexts).
These large state spaces illustrate a potential limitation with this approach to ABP. However, more advanced RL techniques might be suitable for this problem, and we discuss some of these at the end of this chapter and the trade-offs of using them.

5.4.2 Java ABP in Automated Testing

One of the most interesting applications of this ABP library was work performed by Groce et al. in the field of automated testing [29, 31, 30] for container classes (e.g. trees, hash tables, etc.) in the Java programming language.

The testing goal was to cover the maximum number of lines of code by selecting operations from \( \text{ADD}, \text{REMOVE}, \text{UPDATE} \) and selecting from a set of discrete values (say 0 to 10) to insert into the containers. For example, a potential test sequence for a set container might be: \([(\text{ADD}, 3), (\text{ADD}, 4), (\text{UPDATE}, 2), (\text{REMOVE}, 4)]\). Executing this sequence of steps would cause a certain number of lines to be covered. Of course, the more lines covered, the better.

Randomized testing in this context means to randomly select the API to call and the values to pass in. This approach is desirable since it requires little or no work from the API developer to get running and can still prove useful. QuickCheck for Haskell [18] is an example of a randomized tester that is quite popular.

Groce modified the random approach by using ABP to select the operation and value to apply to the data structure. A string representation of the current data structure was used as the context (state) and a reward was given when a new context was reached. The following is an excerpt from [29] sketching how the algorithm used ABP.
AdaptiveProcess test = AdaptiveProcess.init();
Adaptive<String, TestOp> opChoice
    = test.initAdaptive(String.class, TestOp.class);
Adaptive<String, TestVal> valChoice
    = test.initAdaptive(String.class, TestVal.class);
for (int i = 0; i < NUM_ITERATIONS; i++) {
    // Empty test and reg objects
    SUT = new SplayTree(); Oracle = new BinarySearchTree();
    // The state is simply a linearization of the SplayTree
    String context = SUT.toString();
    for (int j = 0; j < M; j++) {
        // Used just like pseudo-random number generator
        // All_vals fields contain a set of all values of the type
        TestOp o = opChoice.suggest(context, TestOp.AllVals);
        TestVal v = valChoice.suggest(context, TestVal.AllVals);
        switch (o) {
            case INSERT: r1 = SUT.insert(v); r2 = Oracle.insert(v); break;
            case REMOVE: r1 = SUT.remove(v); r2 = Oracle.remove(v); break;
            case FIND: r1 = SUT.find(v); r2 = Oracle.find(v); break;
        }
        assert ((r1 == null && r2 == null) || r1.equals(r2));
        context = SUT.toString(); // Update the context
        if (!states.contains(context)) { // Is this a new state?
            states.add(context); test.reward(1000);
        }
    }
}
test.endEpisode();

Effectively this work converted a problem using random guessing into a guided optimization problem, by asking ABP to learn the following: “If the data structure looks like such, what operation and value should we choose to maximize line coverage (reward)?” For instance, if a container already contains a value, then ABP would learn that attempting to reinsert that value would not cover very many lines.
Groce compared the above adaptive algorithm with random testing [33] as well as more sophisticated shape-abstraction testing [73] over 15 different container types such as AVL trees, binomial heaps, separate-chained hash tables and so forth. Since each testing performs at a different rate, they capped the running time for each method at a fixed value. Hence, the question became: how much coverage can you achieve in $t$ minutes?

This work found that an ABP-based testing approach is generally on par with randomized testing, but generally superior to the shape-abstraction approach. That is, the ABP approach and random testing both outperformed each other for different cases (but both did generally better than the third method).

This approach showed how ABP could take a problem using a randomized approach and guide it to more effective exploration of the solution space. A quote from [29]:

“'It would be of benefit to programmers to have access to other approaches that strike a similar trade-off between effectiveness and ease-of-use as random testing but work well in cases where random testing does not do well.”

ABP provides a simple and intuitive interface to machine learning algorithms, and this work in the testing domain demonstrated a very creative uses of this adaptive model of programming.

5.5 Summary

This chapter has presented an implementation of Adaptation-Based Programming in an object-oriented language. Unlike our previous approach, this approach exposed adaptive value constructs in such a way that the ABP library can trace the evolution of the program state and automatically propagate a reward back to earlier choice decisions. This approach is in stark contrast
to the manual-update approach from Chapter 4 where the specifics of the adaptation algorithms were left in the hanks of the user.

This chapter’s approach forces the adaptive programmer to represent their adaptive program as an MDP. The context argument passed to `suggest` represents the state in the MDP. The ABP library can then treat an adaptive program as an RL simulator and solve it with standard reinforcement learning algorithms. This approach exploits the sequential nature of programs and works well in many real-world cases.

In our case studies in Section 5.4, specifically in the Yahtzee example, we encountered the problem of massive state spaces. Later work by Pinto [58] experiments with modifying this library to support a policy gradient approach. Policy gradient algorithms use a much more compact state representation; instead of representing each state explicitly in a table as we do here, it expresses the state as a feature vector of continuous real values. The reduction in storage is from linear (in the number of contexts) to constant (the number of components in the context). However, one possible downside of this approach is that it burdens the programmer with the explicit construction of that feature vector and expresses various features as continuous values (perhaps not an intuitive process).
Chapter 6: Automatic Reward Attribution Through Data Flow Analysis

The previous chapter presented a Java library for Adaptation-Based Programming. That library
treated contextual arguments as a representation of an MDP’s current world state. By making this
assumption an adaptive program’s adaptive values can be independent of the reward statements.
The system can automatically determine which adaptives were responsible for a given reward,
and the user does not need to manually update and manage the state of each adaptive for a single
reward. The learning algorithm does this by assuming that all choices it makes influence future
rewards, and this assumption can cause subtle problems.

\[
\begin{align*}
    a &= A.suggest(s, ...) \quad \text{reward}(f(a)); \\
    b &= B.suggest(t, ...) \quad \text{reward}(g(b));
\end{align*}
\]

\[
\begin{align*}
    a &= A.suggest(s, ...) \quad \text{reward}(f(a)); \\
    b &= B.suggest(t, ...) \quad \text{reward}(g(b)); \\
\end{align*}
\]

Figure 6.1 illustrates a sample problem that can occur when a programmer reorders a reward
operation across an independent choice (suggest). The reordering “artificially” changes the
underlying MDP, that is, when we map adaptive programs to MDPs, there is an unenforceable hidden restriction that rewards associated with a state should be indicated to the learning algorithm before any future choice (suggest) is made. Failure to do this leads to noise in the reward signal and thus inefficiency, but typically not failure to learn the optimal solution.

The goal of the approach outlined in this chapter is to reduce some of these implicit ordering restrictions such as the example above through an automated data-flow analysis approach using techniques from our work in [12]. Specifically, this chapter defines a static may-influence relation between choices and rewards using a static analysis technique similar to program slicing [78].

The rest of this chapter is organized as follows. Section 6.1 presents the formal syntax for a simple imperative adaptive programming language much like that from Chapter 2. Section 6.2 details a formal algorithm over this syntax to compute a data-flow relation linking rewards to choices that may influence them. Section 6.3 demonstrates how this information can detect the “live range” of a choice based on the data-flow relation. Section 6.4 describes the reinforcement learning algorithm used and describes how it can be modified to use the data-flow relation to more quickly improve adaptive program performance as well as detect misuses of the adaptive programming library. Finally, Section 6.5 shows the improvement this approach permits on various example programs, and Section 6.6 concludes.

6.1 A Simple Adaptation Programming Language

To study the effects of data-flow analysis on adaptive programs we return to the simple imperative language presented in Chapter 2.

As shown in the Figure 6.2, the syntax provides basic language constructs for a typical imperative language, extended by two adaptive statements for describing choice and providing rewards. The syntax also implicitly refers to expressions as $e$, variables as $v$, and labels as $l$. 
Statements

\[ s ::= v = e \quad \text{assignment} \]
\[ \mid \text{if } e \text{ then } s \text{ else } s \quad \text{alternation} \]
\[ \mid \text{while } e \text{ do } s \quad \text{iteration} \]
\[ \mid s; s \quad \text{sequence} \]
\[ \mid l:a \quad \text{adaptivity} \]

Adaptive statements

\[ a ::= \text{reward}(e) \quad \text{reward} \]
\[ \mid \text{choose}(v, e, \tilde{e}) \quad \text{choice} \]
\[ \tilde{e} ::= [e, \ldots, e] \quad \text{alternatives} \]

Figure 6.2: Syntax for a Simple Language

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 the implementation the labels are automatically added by the compiler.)

The choice construct \text{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 \text{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.

One can view the language above as a simpler form of the object-oriented version of ABP presented in the previous chapter. There is implicitly one adaptive process and one adaptive variable. The \text{choose} operation is then just a \text{suggest} to that single adaptive, and the \text{reward} construct here corresponds to a call to \text{reward} of the single adaptive process.
We start by outlining the basic ideas of ABP at a high level with the following simple example adaptive program.

\[
\begin{align*}
  r &= 0 \\
  c1 &\text{choose}(x, [0,1]) \\
  c2 &\text{choose}(y, [0,1]) \\
  \text{if } x \text{ then} & \\
  r1 &\text{reward}(1) \\
  \text{else} & \\
  r &= y + 2 \\
  r2 &\text{reward}(r)
\end{align*}
\]

Intuitively, we can see that the reward statement labeled \(r_1\) depends only on the adaptive value chosen at \(c_1\) (stored in \(x\)). However, the reward at \(r_2\) depends on both \(c_1\) and \(c_2\), which can be seen as follows. The value of the reward passed in \(r_1\) depends on the value of \(x\), hence it depends on \(c_1\). The influence from \(c_2\) arises since the value of \(r\) at the reward statement \(r_2\) 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 \(y\)).
\[ r = 0 \]
\[ x = 0 \]
\[ y = 1 \]
\[ \text{if } x \text{ then} \]
\[ \quad r1: \text{reward}(1) \]
\[ \text{else} \]
\[ \quad r = y + 2 \]
\[ r2: \text{reward}(r) \]

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.

### 6.2 Reward Attribution Algorithm

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

The goal of this algorithm will be compute the relation \( R \) relating each choices to rewards it may influence.

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 \( R \subseteq L \times L \) 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 6.3. We use the following auxiliary functions and notation. The projection of the influence map \( I \) with respect to a set of variables \( X \) is defined as follows.

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

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

The \textsc{Reward} rule states that a \textsc{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 \textsc{reward} also might be influenced by choices and must also be considered to influence this statement. To this end, the function $\text{vars}$ computes all the variables contained in $e$, and the projection operator on $I$ selects the choices that influence those variables.

The \textsc{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 \textsc{Assignment} rule is very similar to the \textsc{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.

$$v = v + 2x + y$$

Assume the following initial definitions of $A$ and $I$.

\begin{equation}
A = \{c_1\} \text{ and } I = \{(v,c_2),(x,c_3),(y,c_3),(y,c_4)\}
\end{equation}

Then $I$ changes as follows. First, we subtract old definitions of $v$ since $v$ is being redefined thus removing $\{(v,c_2)\}$. 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 $\{c_2,c_3,c_4\}$. Finally, we include $A$, just $\{c_1\}$ in this case.
We cross these choices with \{v\} to get the new definition for \(I\) given below.

\[
I = \{(v, c_2), (v, c_3), (v, c_4), (x, c_3), (y, c_3), (y, c_4)\}
\]

Sequential composition of statements, described by \textsc{Seq}, threads \(I\) and \(R\) through both statements, nothing terribly special or exciting happens with this rule.

The \textsc{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 \textsc{If} rule merges the results \(R_t\) with \(R_e\) and \(I_t\) with \(I_e\) by taking their union.\(^1\)

Similar to \textsc{If}, the \textsc{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 \textit{forward-maybe} iterative data-flow analysis [1].

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

\[
r = 0
c1:\text{choose}(x, [0,1])
\text{if } x \text{ then}
\quad c2:\text{choose}(y, [0,1])
\quad r1:\text{reward}(1)
\text{if } y \text{ then}
\quad m = 2
\quad r2:\text{reward}(m)
\]

Consider the above example consisting of two choices (both without explicit context values). Initially \(A\), \(I\), and \(R\) are all empty. The \textsc{Assignment} rule assigning 0 to \(r\) has no effect since

\(^1\)If we intersected sets here, we would get the \textit{definite influences} instead of the \textit{potential influences} as we currently have.
nothing is influenced by any choices here. However, after the first choice, the \texttt{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 \texttt{if} statement the \texttt{IF} rule extends \(A\) by \(c1\) since that choice influences its guard expression \(x\). Next, the second \texttt{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 \texttt{if} statement following that reward extends \(A\) further adding its guard expression \(y\)'s influences \(\{c1, c2\}\). The assignment within that nested \texttt{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 \texttt{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.

\[
R = \{(c1, r1), (c1, r2), (c2, r2)\}
\]

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

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

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.
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.

### 6.3 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 a player wins a chess match, that person 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.
\begin{verbatim}
  i = 0
  while i < 10 do
    i = i + 1
    c:choose(a,[0,1])
    if a == 1 then
      r:reward(1)
  \end{verbatim}

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 iter-
ation 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 mis-compute 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 6.3 we can determine the set of choices \textit{definitely}
invali-
dated at each binding construct (a \texttt{choice} or assignment operation in the example imperative
language). For any statement that (re)defines $v$ we compute the invalidated set as follows.

$$\text{invalidated} = I[v] - \text{live}(I[v := \emptyset] \cup A)$$

$I[v]$ 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 ($I[v := \emptyset]$) 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 [1].

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, we have the following sets.

- $A = \emptyset$
- $I = \{(x,c1),(y,c2),(z,c1),(z,c2)\}$
- $I[z] = \{c1,c2\}$ ($z$ depends on both choices)
- $I[y := \emptyset] = \{c1,c2\}$ ($x$ depends on $c1$ and $y$ on $c2$, but none are live)
The invalidated choices would then be computed as.

\[ \{c_1, c_2\} - live(\{c_1, c_2\} \cup \emptyset) = \{c_1, c_2\} - \{c_1\} = \{c_2\} \]

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.

### 6.4 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 \( \epsilon \)-greedy exploration [70]. 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.

```plaintext
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 (a trace of events).

A Monte-Carlo algorithm is an unbiased learning algorithm that 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 the necessary 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 6.3 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 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 6.3 we further modify MCRA as follows. At each binding statement we first compute the invalidated set as described in Section 6.3. 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 mis-attributed to that choice.
6.4.1 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. Details are presented here.

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.

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

(The $\text{Res}(c, a)$ term is the residue or error term.)

Consider the variances of these random variables.

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

The variance of two (correlated) random variables is the sum of each variance plus twice the covariance between both variables. Hence, we can rewrite the right-hand side as follows.

$$\text{Var}(Q_{MC}(c, a)) = \text{Var}(Q_{MCRA}(c, a)) + \text{Var}(\text{Res}(c, a)) + 2 \text{Covar}(Q_{MCRA}(c, a), \text{Res}(c, a))$$
The variance of the new \( Q \) estimate for \((c,a)\) will be less than that of the standard estimate as long as the following inequality holds.

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

Since the term on the left is always positive, the above condition only fails when there is a large negative correlation (covariance) between the sum of influential reward terms and the rest. Although such a dominant negative correlation may occur in pathological cases, it is hard to construct any real examples where this occurs. In all reasonable cases, the variance in the residual dominates, and the modified estimate \((Q_{MCRA})\) has smaller variance than the standard estimator as expected.

Intuitively, this result makes sense. For each choice in a program, \(MC\) uses the full sequence of rewards that follow that choice. In contrast, \(MCRA\) considers a more accurate subsequence by eliminating those rewards that provably cannot have been influenced by the choice. Those rewards eliminated from the estimate can be thought of as noise in the reward signal.

### 6.4.2 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 it is an orphan. Intuitively, this is indicating that no choice the ABP library can make will influence our ability to get that reward. Therefore, it is spurious.
MCRA could silently ignore these rewards (since 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.

6.5 Empirical Evaluations

We implemented a translator for a small language very similar to the one presented in Figure 6.2 and the algorithm given in Figure 6.3 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). 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

---

2No exploration is performed during evaluation, and the ABP learner always chooses the best known option for each choice.
Figure 6.4 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.
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 this program uses 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 that we can easily determine optimal adaptive program behavior analytically and verify the ABP library reaches this.

The results of this algorithm are shown in Figure 6.5. 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).

\(^3\)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.
For this program we also compare the algorithm to two others. The \textit{SARSA(\(\lambda\))} algorithm is a venerable and robust learning algorithm and was used in older versions of ABP as the learning algorithm. Like the \textit{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 \textit{random} algorithm is at the bottom of the plot and is given to illustrate the estimated reward we would get by guessing randomly.
The **ROBOT PROBLEM**

We now proceed to illustrate a larger ABP program that exhibits some of these reward-attribution problems in a more complicated example consisting of several partially independent goals.

```plaintext
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)
```

Much like the signaling robot problem from Chapter 2, this program controls a robot with position \((x,y)\) moving around a grid. As before, 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 \((g_x, g_y)\). Moreover, each time the robot reaches its goal it gets a reward and then requests a new goal by calling the \texttt{InitGoalX} and \texttt{InitGoalY} functions. The move choice at \(c_2\) takes a context consisting of a normalized direction towards the robot’s goal. The function \texttt{context} pairs those vectors into one value.

Additionally, there is a partially independent secondary goal. Initially, the robot starts with an unknown number of goods that it must sell over the course of time. Each time step, the market price for goods changes and the program must choose to sell or retain the goods. This uncertainty is encoded with the choice \(c_1\).

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

Figure 6.6 shows the learning behavior of our various algorithms on this problem. \textit{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 \textit{SARSA}(\(\lambda\)). Instances of this algorithm found the optimal, but some got stuck at sub-optimal policies. The regular \textit{MC} algorithm did not perform very well, and although some learning runs did find an instance of the best known policy, far fewer did.
The TANGLED-IF-LOOP Problem

To test the benefit of the choice invalidation algorithm in MCRA we consider a version of the TANGLED-IF repeatedly executed in a loop (shown below).
\[ i = 0 \]
while \( i < 10 \) do
\[ i = i + 1 \]
ca:choose(a, [0,1,2,3])
cb:choose(b, [0,1,2,3])
\[ r = \text{normal()} + 0.5 \]
if \( b == 1 \) then
\[ \text{rb: reward}(r + 0.5) \]
else
\[ \text{rb: reward}(r) \]
if \( a == 1 \) then
\[ \text{ra1: reward}(r + 0.5) \]
else
\[ \text{ra2: reward}(r) \]

The wrapped problem (loop body) should be treated independently each loop iteration, but traditional algorithms such as \textit{MC} and \textit{SARSA}(\lambda) will not be able to observe this independence. Instead they incorrectly assume that rewards from later iterations depend on choices made in previous iterations. In addition, each loop iteration adds a few more options per choice than \textit{TANGLED-IF} and randomizes the reward slightly to make it a more difficult learning problem. However, the problem still has a unique optimal solution (policy) that is easily determined (that is, to reach the rewards with the additional 0.5 added to them).

Figure 6.7 shows the result and illustrates that with \textit{MCRA} all \( m = 32 \) learners quickly learn the optimal solution within about 10 runs, whereas the other methods struggle.

### 6.6 Summary

This chapter illustrates some problems that arise when viewing adaptive programs as MDPs. Specifically, that approach is very sensitive to the order of the choices made. However, in many
cases data-flow analysis can be used to determine exactly which choices a reward should be attributed to, and we can use this information to determine a better MDP for the program.

Prior work in ABP [57] showed some simple ABP programs with some of the same problems as those shown here 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, this work has given a formal definition to such an analysis, but applies it to a less sophisticated RL algorithm.

An important concept in ABP is the freedom it allows the programmer in structuring their programs. The cost of this freedom is inefficiency and sub-optimal learning. This chapter has presented 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. Additionally, this chapter illustrates how static analysis and compiler-support can benefit adaptive programming languages by speeding up learning and even detecting errors.
Chapter 7: Conclusion

Adaptive programming languages are languages with special constructs that allow the programs to adapt to an uncertain environment. Programs in these languages consist of normal program code as well as a few points of uncertainty represented as non-deterministic choices of some sort. Over many executions programs evolve and improve in response to a feedback or reward signal. This thesis has investigated several approaches to adaptive programming and some of the trade-offs and relative advantages of each method.

To illustrate the basic idea behind adaptive programming and how it relates to reinforcement learning Chapter 2 presented a simple imperative language with two adaptive constructs, choose and reward. We illustrated how programs using these two features mapped to formal learning processes from RL. Our first exploration of the meaning of adaptive programming languages was presented in Chapter 4 where we presented a domain specific language embedded within a functional programming language to support adaptive programming. This language was centered around the concept of generic adaptive values where both the underlying adaption pattern and feedback signal type were fully programmable. Around these core ideas we defined various patterns of adaptation for adaptive values to support the programmer, the most important being the contextual adaptive (expressed as adaptive functions). This approach allowed us to rapidly experiment with and discover different adaptation patterns in adaptive programs as well understanding the underlying concepts. However, a disadvantage to this approach is that it led to more complex adaptive programs. Users had to explicitly determine which adaptive values to give feedback to and what feedback to give them. In Chapter 5 we present a more specialized adaptive programming library within an object-oriented language. In this approach we fixed the
feedback type to a numeric type and linked all adaptive values to a chain of explicit contexts, each representing the current environment of the learning problem. This allowed us to represent adaptive programs as Markov Decision Processes and enabled us to utilize powerful learning algorithms to solve the problem by exploiting the sequential nature of program. However, there were cases that arose where this change proved problematic; seemingly innocent program transformations changed the order in which contexts are observed by the library causing rewards to influence the wrong adaptive values. To address this Chapter 6 defined a data-flow analysis to statically determine which rewards a given choice can and cannot influence. We illustrated how this greatly alleviates the problem for many cases.

This work has made various contributions in the field of partial programming. It defined the concept of adaptive values in a domain-specific language for representing adaptive programs. Moreover, it showed how many different smaller adaptives could be composed into larger ones using reusable combinators and generic data types. This work also contributed a definition of partial programming that formalized the inclusion of a reward as part of the program instead of as an external signal as was done previously by [2]. Finally, it showed how adaptive programs could be represented in such a way so that program structure could be leveraged to guide the learning algorithm more effectively.

One future direction for this work is to consider how the data-flow analysis could be applied to more sophisticated learning algorithms such as SARSA(\(\lambda\)). Currently, the way this thesis applies reinforcement learning makes it difficult to use this information; perhaps a different mapping of adaptive program to a learning problem could do better. Additionally, the data-flow analysis work only considered the use of static program structure; dynamic information (such as a runtime trace) might generate faster adaptation. Instead of computing the choices that may influence a reward, a different approach could compute the choices that definitely influence a reward. However, one challenge with this is in limiting the size of the runtime information
stored. For instance, a loop containing a choice that executes millions of times must, in theory, retain information about each of those choices made. Hence, this changes the space usage of the program to at least $O(n)$, which might be prohibitive if the original program was expecting to use constant space.

A second interesting direction would be to find creative ways to apply the learning algorithms used in Chapters 5 and 6 in a purely functional setting. In principle there are not any special technical difficulties implementing these algorithms in such a language. However, these higher-level functional languages might present a more opportunities for higher-level and concise definitions of larger adaptive programs.
Bibliography


