Programming by Demonstration: a Machine Learning Approach

Tessa Lau

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2001

Program Authorized to Offer Degree: Department of Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Tessa Lau

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

Daniel S. Weld

Pedro Domingos

Reading Committee:

Daniel S. Weld

Pedro Domingos

Henry Lieberman

Date:

University of Washington

Abstract

Programming by Demonstration: a Machine Learning Approach

by Tessa Lau

Co-Chairs of Supervisory Committee:

Professor Daniel S. Weld
Department of Computer Science & Engineering

Assistant Professor Pedro Domingos
Department of Computer Science & Engineering

Programming by demonstration (PBD) enables users to construct programs to automate repetitive tasks without writing a line of code. The key idea in PBD is to generalize from the user's demonstration of the program on a concrete example to a robust program that will work in new situations. Previous approaches to PBD have employed heuristic, domain-specific algorithms to generalize from a small number of examples. In this thesis, we formalize programming by demonstration as a machine learning problem: given the changes in the application state that result from the user's demonstrated actions, learn the sequence of instructions that map from one application state to the next. We propose a domain-independent machine learning approach to PBD that is capable of learning useful programs from a small number of examples. This approach addresses two difficult questions: (1) How do we construct the search space of possible program statements? (2) How do we search this large space efficiently?

Our solution is based on the concept of version space algebra. Mitchell [55] formalized concept learning as a search through a version space of hypotheses consistent with the examples. Concept learning may be thought of as learning functions that map from an instance to a binary classification. In this work, we extend version spaces to apply to com-

plex functions: functions that map from one complex object to another. We then present version space algebra, a means for combining several small spaces in order to construct complex version spaces. To illustrate the approach, we describe the SMARTedit programming by demonstration system for learning repetitive text-editing programs. SMARTedit is capable of learning useful programs from as little as a single training example. Finally, we generalize programming by demonstration to the broader problem of learning programs with loops and conditionals from traces of their execution behavior. We demonstrate this generalization with the SMARTpython system that is capable of learning programs with loops and conditionals from traces of the programs' execution.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This thesis is the result of countless discussions, support, advice, and guidance from a significant number of people. First of all I must thank my advisor Dan Weld for more than I can put into words. You believed in me and in the work, and showed me by example how to do great research. No less importantly, I thank my co-advisor Pedro Domingos for being a part of this research. Your clarity and vision helped turn our woolly notions into a crisp, beautiful framework.

I couldn't have made it this far without the wonderful Dan Egnor. Thanks for all of it: sticking with me through deadlines, making me eat when I'm hungry, talking to me about everything even when I don't want to, and keeping a sense of humor about our misbehaving cats.

My colleagues at the University of Washington and elsewhere have made this six-year thesis journey entertaining, enlightening, and exhilarating. Special thanks especially to: Corin Anderson and Steve Wolfman (the other two musketeers), Greg Badros, Jeremy Buhler, Adam Carlson, Michael Ernst, Gambit and Domino, Zachary Ives, Nick Kushmerick, Neal Lesh, Michael Noth, Jon Nowitz, Mike Perkowitz, Rachel Pottinger, and Doug Zongker.

Steve Wolfman worked closely with me on much of SMARTedit. Thanks for the endless discussions of details, both important and unimportant, and for not checking too many bugs in to my code. Steve gets the blame for the initial idea and implementation of unsegmented and startless programs, extending SMARTedit to provide mixed-initiative interaction, and countless enhancements to SMARTedit itself.

Jim Guerber implemented several of the version spaces in SMARTedit, including character and word offsets, character class disjunctions, and a port to Java. Dutch Meyer contributed a port to Emacs.

Chapter 1

INTRODUCTION

Computers are difficult to use. Although graphical user interfaces have helped to bring computing to nonspecialists, most of today's interfaces still require users to perform tedious and repetitive commands in precise detail in order to operate these powerful machines. Nowhere is this more evident than in repetitive tasks, where the same operation must be performed repeatedly on a number of varying instances.

Examples of such repetitive tasks abound through all facets of computer use, but arise especially in managing information: transforming lists of addresses from one format to another, collecting data from a set of web pages into a single document, printing collections of files, organizing files into folders, publishing data on the web, keeping address books and calendars up to date, managing bibliographies, and so on. Automating such tasks, which often span application boundaries, is usually only possible by writing a program.

Unfortunately, programming is difficult. The vast majority of computer users do not know how to write programs. Moreover, many people who do have the skills to write programs choose not to automate many of their repetitive tasks simply because it is faster to complete the task manually than it is to write and debug a program to accomplish the same objective. A would-be programmer is faced with two challenges: learning how to control each application via its application programming interface, and constructing a sequence of commands that correctly generalizes to all iterations of the repetitive task. Solving these challenges in concert often requires more time and effort than simply completing the task manually.

Programming by demonstration (PBD) enables any user to write programs simply by demonstrating the program on one or more concrete examples. If a user can use the appli-

cation, she can program it. Rather than writing and debugging statements in an obscure programming language, which may be completely decoupled from the visual appearance of an application, a user may construct a program simply by demonstrating the desired behavior of the program using the interface with which she is already familiar. From this sequence of user actions, a PBD system infers a program that is consistent with the observed actions and generalizes to new instances of the task.

The macro recorders available in many applications today are a powerful but degenerate form of programming by demonstration. Typical macro recorders capture the exact keystrokes entered by the user, and play them back on demand. However, these macros are brittle; recording a macro that performs correctly in a wide variety of different situations is a daunting endeavor, which may require several attempts. PBD shares a user interface similar to keystroke-based macros. Instead of recording a literal sequence of keypresses, however, a PBD system generalizes from the demonstrated actions to a robust program, possibly including loops and conditionals, that is more likely to work in different situations.

At the other end of the spectrum, a suite of office applications may provide a single scripting language (*e.g.*, Visual Basic) in which a programmer may write scripts that control the set of applications for which language bindings are defined. Beyond requiring abstract programming ability, scripting applications in this manner requires learning the programming interface for the application domain, including the mapping between user interface objects and the abstract programming objects accessible from the scripting language. Because of the extra effort necessary to learn the programming interface, even people with programming abilities often do not expend the effort to automate simple repetitive tasks.

PBD enables ordinary users to automate repetitive tasks *without programming*, tasks that they would otherwise have to complete tediously in precise detail. While PBD in a single application would eliminate many of the repetitive activities encountered by users of that application, our goal is to provide *cross-domain* PBD. Many repetitive tasks span multiple applications, including spreadsheets, email programs, web browsers, and database applications. The goal of this work is to provide a single unified PBD framework that does not rely on domain-specific heuristics to generalize from small numbers of examples.

The key component of a PBD system beyond simple macro recording is the *generalization*

```
Ardissono, L., and Sestero, D. (1996). Using dynamic user models in the
recognition of the plans of the user. In User Modeling and User Adapted
Interaction, volume 2, 157--190.

Carberry, S. (1990). Incorporating default inferences into plan
recognition. In Proc. 8th Nat. Conf. AI, volume 1, 471--8.

Ferguson, G., and Allen, J. (1998). Trips: An integrated intelligent
problem-solving assistant. In Proc. 15th Nat. Conf. AI, 567--572.
```

Figure 1.1: Example bibliography text to be converted to BibTeX

process. Because the user generates each training example, the system must be capable of generalizing from a very small number of examples. Previous approaches to programming by demonstration have employed heuristic, domain-dependent methods to reduce the user effort required for generalization. In some cases, the system asks the user to select the correct generalization. In others, the system follows a number of hand-crafted rules to select an appropriate generalization.

In contrast, this work is based on a formal machine learning framework for generalizing from demonstrations to programs. Our approach provides a clear definition of the space of hypotheses considered during the generalization process as well as clean way to specify preference biases between multiple consistent hypotheses. Before describing the contributions of this thesis, we first describe a motivating example of PBD in the text-editing domain.

1.1 A motivating example: bibliography editing

The text-editing domain is a source of many repetitive tasks. Figures 1.1-1.3 provide a concrete example of the sort of task faced by users in this domain. Consider the problem of converting bibliography entries from a standard published format to BibTeX format. Figure 1.1 shows several such bibliographic entries in the standard format, and Figure 1.2

shows the entries after they has been transformed into the desired format. Figure 1.3 shows a sequence of text-editing commands that a user might perform (directly in the user interface) to transform this particular bibliographic entry.

Imagine a user faced with tens or hundreds of such bibliographic entries, each with slight variations. Can she do better than to reformat all the entries by hand? Slight irregularities in the data (*e.g.*, author name variations, varying numbers of fields in each entry) render it difficult to record a macro (single sequence of keypresses) that can correctly transform all the entries. Alternatively, the user could write a script in an abstract programming language to automate the task. However, programming is a highly precise endeavor that is prone to errors; it is often the case that writing a script to perform a task takes longer than performing the task manually.

With programming by demonstration, a user teaches the system how to transform bibliographic entries, and thereby constructs a program capable of repeating that transformation, merely by demonstrating the correct behavior of the program on concrete examples. We shift the burden of generalization from the user to the system. With PBD, users are responsible only for providing examples, and not for generalizing from these examples to a robust program.

Our SMARTedit system is able to learn procedures to accomplish this task and more from a handful of training examples. For example, by watching the user's actions as she transforms the bibliography entry in Figure 1.1 from standard format to BibTeX format, our system learns procedures such as the one shown in Figure 1.4. Our experimental results are presented in Chapter 5. The remainder of this thesis discusses the version space algebra learning framework that allows us to learn these complex procedures from a small number of examples.

1.2 Cross-domain programming by demonstration

Once we have shown how to PBD-enable a single application, we turn to the problem of generalizing from this one domain to a PBD framework that can span multiple domains. A crucial requirement for cross-domain programming by demonstration is an abstract language that supports the types of control flow constructs necessary for representing programs of interest. Just as Microsoft Office uses the Visual Basic language to provide the glue necessary to script multiple Office applications at once, a general cross-domain PBD system must provide some common framework shared across all applications for learning programs that span the variety of different applications. While each statement in the program may be drawn only from a single domain, the general framework is not tied to a particular domain.

In this thesis, we show that our machine learning approach to programming by demonstration is general enough to learn programs in an abstract programming language (a subset of the Python programming language) that includes conditionals, loops, and arrays. Our SMARTpython system is capable of learning complete Python programs from traces of their execution behavior given only a handful of traces. We describe the generalized framework in such a way that domain-specific modules can be plugged into the framework in order to learn programs in specific domains.

In demonstrating that we have a generic framework that can learn programs from traces, we hope to show that the framework can be used for cross-domain PBD. Our SMARTedit system is one domain for which PBD is useful; the SMARTpython system shows how PBD systems for multiple domains can be combined together within a single framework.

1.3 Our contributions

This thesis makes three main contributions to the problem of cross-domain programming by demonstration. Notably,

1. A robust, domain-independent framework for programming by demonstration.

   Most previous approaches to PBD have employed heuristic, domain-specific algorithms for generalizing from user actions to an executable program. In contrast, we employ a general, machine learning approach that enables programming by demonstration in arbitrary domains without such heuristics. Much machine learning research (*e.g.*, into classification learning) has investigated the learning of functions that map from a vector of inputs to a single discrete output. As we show in this thesis, however, such learning algorithms are not up to the task of learning programs by demonstration.

   Thus, the first contribution of this thesis is a new machine learning framework for learning what we call *complex functions*—functions that map from one complex object to another complex object. In programming by demonstration, these complex objects take the form of states of the application; however, this framework is not limited to programming by demonstration. We explain this framework in terms of the version space formulation [55] originally developed for concept learning. A version space is a subset of the hypothesis space that contains only those hypotheses that are consistent with the examples. The hypotheses in the space are assumed to be partially ordered by generality; one hypothesis is more general than another if and only if it covers more examples than the other. But concept learning can be viewed as the learning of a function that maps from an input to a value in the set $\{0, 1\}$. Our contribution extends version spaces to learn *arbitrary* functions—not just those with a binary range. Further, in order to maintain these extended version spaces efficiently, we realize that efficient maintenance of the version space by its boundary sets only requires some partial order, not necessarily the generality ordering.

   Each action taken by the user during a demonstration causes a change in the state of the application. Thus, each action is essentially a *function* that maps from one application state to another. Different functions may be consistent with a single state change, the same way $f(2) = 4$ is consistent with the two functions $f(x) = x + 2$ and $f(x) = x^2$. In our approach, we partition the demonstration into a sequence of actions and learn each action individually. To learn a single action given the state before and after the action, we construct a hypothesis space containing the set of all possible actions (functions from state to state) that the user could have taken. As the user demonstrates actions, the state before and after each action is used to update the version space to contain only those hypotheses consistent with the observed states.

2. The version space algebra approach for constructing the the hypothesis space of allowable program statements.

   In this thesis we introduce a *version space algebra*—a framework for constructing complex version spaces hierarchically through the composition of simpler, more restricted version spaces. Specifically, we define three algebraic operators (union, join, and transform) that may be used to combine simple version spaces into complex ones. This modular approach leads directly to the concept of *version space libraries*—reusable component building blocks that can be ported easily to a new domain as part of a different version space hierarchy.

3. Efficient maintenance and execution of these hierarchical version spaces.

   The version space contains the subset of the hypothesis space that is consistent with the observed examples. With each example, we update the *version space* by removing those hypotheses that are inconsistent with the example. As more examples are provided, more hypotheses are removed from the version space, until it (hopefully) contains only the single correct hypothesis.

   Our third contribution is methods for efficient maintenance and execution of hierarchical version spaces. There is a fundamental tradeoff between increased expressiveness (to be able to represent the kinds of programs users want to create) and sample complexity (so those users obtain useful results in real time with only a handful of training examples). This thesis shows that the hierarchical version spaces constructed using version space algebra retain the efficient representation of the original version space approach, and that the version space algebraic operators preserve PAC learnability of the component version spaces. In addition, we present a probabilistic framework for constructing a preference bias by attaching prior probabilities to hypotheses in the version space. The probabilistic framework allows us to execute our version spaces and extract the most likely hypotheses even before the version space has converged to the single correct hypothesis. The framework allows application designers to incorporate their domain knowledge into the system. In sum, the version space algebra framework allows efficient, modular, and portable representation and execution of a complex hypothesis space.

We illustrate the version space algebra approach with two implemented systems, SMARTedit and SMARTpython. SMARTedit learns repetitive text-editing programs by example, automating repetitive tasks in some cases with as little as a single training example. While this system shows that we can apply version space algebra to PBD-enable a text-editing application, our ultimate goal is to provide a general, cross-domain framework for PBD-enabling a wide variety of applications. To that end, we apply version space algebra to an abstract, general-purpose programming language—a subset of Python. We show that the same techniques we use to learn text-editing actions can be applied to learn statements in this abstract programming language; and we further extend the approach to learn complete programs with complex control flow such as conditionals and loops. With SMARTpython, we show that our version space algebra approach is capable of learning programs from traces without using heuristics specific to a particular domain; we believe that our approach will thus be applicable to many domains beyond the ones presented here.

1.4 Overview of the rest of this thesis

The rest of this thesis is structured as follows. Chapter 2 describes prior work on programming by demonstration. Chapter 3 formalizes the problem of programming by demonstration as a machine learning problem, describes our version space algebra solution and the probabilistic framework, and summarizes related work in machine learning. Chapter 4 discusses the SMARTedit programming by demonstration system for text-editing, and its implementation using version space algebra. Chapter 5 details our evaluation of the SMARTedit system, including a user study and results with a corpus of repetitive text-editing tasks. Chapter 6 generalizes our approach beyond text-editing and describes the SMARTpython system for learning programs in an abstract programming language, including programs with loops, conditionals, and arrays. Chapter 7 outlines directions for future work, and the final chapter concludes with a summary of contributions.

```
@inproceedings(Ardissono-1996,
  author="Ardissono, L., and Sestero, D.",
  year=1996,
  title="Using dynamic user models in the
  recognition of the plans of the user",
  booktitle="User Modeling and User Adapted
  Interaction",
  pages="157--190"
)

@inproceedings(Carberry-1990,
  author="Carberry, S.",
  year=1990,
  title="Incorporating default inferences into plan
  recognition",
  booktitle="Proc. 8th Nat. Conf. AI",
  pages="471--8"
)

@inproceedings(Ferguson-1998,
  author="Ferguson, G., and Allen, J.",
  year=1998,
  title="Trips: An integrated intelligent
  problem-solving assistant",
  booktitle="Proc. 15th Nat. Conf. AI",
  pages="567--572"
)
```

Figure 1.2: Example text converted to BIBTEX format

Move the cursor to the beginning of the string `Ardissono`

Insert the text `@inproceedings(` ENTER

Copy the word `Ardissono`

Move to the end of the previous line after the parenthesis

Paste the word `Ardissono`

Insert the string `-`

Move the cursor to beginning of the string `1996`

Copy the word `1996`

Move to the end of the previous line after the dash

Paste the word `1996`

...

Move to the end of the string `157--190.`

Delete the period after `190`

Insert the string `''` ENTER `)`

Figure 1.3: Sequence of text-editing actions to transform one citation to the desired format

Move the cursor to the beginning of the next capitalized word

Insert the text `@inproceedings(`$\boxed{\texttt{ENTER}}$

Copy the following word to the clipboard

Move to the end of the previous line after the parenthesis

Paste the contents of the clipboard

Insert the string `-`

Move the cursor to beginning of the next occurrence of four digits

Copy the next four characters to the clipboard

Move to the end of the previous line after the dash

Paste the contents of the clipboard

...

Move to the end of three digits followed by a period

Delete one character to the left

Insert the string `''`$\boxed{\texttt{ENTER}}$)

Figure 1.4: General program to transform one citation to the desired format

Chapter 2

PRIOR WORK ON PROGRAMMING BY DEMONSTRATION

This work draws inspiration from a number of previous programming by demonstration systems, both for text-editing as well as for other domains. Edited collections by Cypher [16] and Lieberman [50] provide an overview of the field. After summarizing related work on programming by demonstration, we situate our work in the broader context of adaptive user interfaces. Related work in machine learning will be discussed in section 3.6.

2.1 Text-editing

Prior PBD systems can be roughly classified into two groups: those that reason about the sequence of actions performed by a user, and those that examine the sequence of application states as the user performs the demonstration. First we describe prior work using the state-based approach.

Nix [64] describes the Editing by Example (EBE) system that generalizes from the input/output behavior of the complete demonstration. In other words, EBE attempts to find a program that can reproduce the observed difference between the initial and final state of the text editor. In this respect, SMARTedit is a refinement of EBE that uses not only the initial and final state, but intermediate states as well. SMARTedit's approach has the drawback that it is sensitive to the order in which the user chooses to perform actions, but on the other hand it makes use of more information than EBE is given, and so SMARTedit is able to learn a superset of the programs learnable by EBE. One example of a text transformation supported by SMARTedit but not by EBE is a procedure to number a list of citations in increasing order.

The DEED system described by Fujishima [22] is similar to the EBE system except that it supports multiple focus points—regions of text that are changed on each *turn* (iteration in our terminology). On each turn, DEED asks the user to specify a list of focus subsequences (substrings of the text file to which changes are to be made), and then to specify the modification to each of the subsequences. The user may provide one or more turns. DEED performs generalization in two phases: focus interpretation (finding the hypotheses describing the beginning and the end points of each focus subsequence), and modification interpretation (finding the mapping between the original and the modified subsequence). If multiple interpretations match, one is chosen according to a heuristic preference bias. DEED extends EBE to handle multiple focus points, as well as more complex text transformations such as capitalization. DEED's expressiveness seems roughly similar to SMARTedit's. However, the fact that it ignores the actions used to modify the text, and uses only the input and output of the complete text transformation, means that it must search a larger space to find the correct interpretation. In order to keep the system responsive, the prototype DEED system limits the search space to the first 100 interpretations, indicating that efficient search is a concern. Future work will explore more connections between this work and ours, including the use of version space algebra to learn programs efficiently based on only the initial and final states, and user studies to determine which approach users prefer.

Most of the remaining text-editing PBD systems view a single demonstration as an exemplar of the correct program, and employ domain-specific heuristsics that alter portions of the trace to make them more general (such as generalizing from a constant to a variable). In contrast, SMARTedit maintains the set of all programs that are consistent with the demonstrated examples. For example, Witten and Mo [57] describe the TELS system that records high-level actions similar to the actions used in SMARTedit, and implements a set of expert rules for generalizing the arguments to each of the actions. TELS also uses heuristic rules to match actions against each other in order to detect loops in the user's demonstrated program. Unlike our approach, TELS only generates a single program (which may be incorrect), and can learn only from a single example. In addition, TELS's dependence on heuristic rules to describe the possible generalizations makes it difficult to imagine applying the same techniques to a different domain, such as spreadsheet applications.

Masui and Nakayama [53] describe the Dynamic Macro system for recording macros in the Emacs text editor. Dynamic Macro performs automatic segmentation of the user's actions—breaking up the stream of actions into repetitive subsequences, without requiring

the user to explicitly invoke the macro recorder. Dynamic Macro performs no generalization, and it relies on several heuristics for detecting repetitive patterns of actions. SMARTedit's ability to learn from unsegmented traces illustrates how this type of segmentation can be performed using our version space algebra.

Maulsby and Witten's Cima system [54] uses a classification rule learner to describe the arguments to particular actions, such as a rule describing how to select phone numbers in the local area code. (SMARTedit is able to learn a program to select all but one of the phone numbers in the Cima task given a single demonstration. The anomalous phone number lacks a preceding area code, and is also difficult for Cima to classify correctly.) Unlike other PBD systems, Cima allows the user to give "hints" to the agent that focus its attention on certain features, such as the particular area code preceding phone numbers of interest. However, the knowledge gained from these hints is combined with Cima's domain knowledge using a set of hard-coded preference heuristics. As a result, it is unclear exactly which hypotheses Cima is considering, or why it prefers one over another. In contrast, SMARTedit's version space algebra approach renders the hypothesis space explicit, and provides a clean way to specify a preference bias on the consistent hypotheses. An open direction for future work is to use the the types of hints employed by Cima to affect the probabilities on different hypotheses.

2.2 Programming by demonstration

Many PBD systems have been discussed in the literature, in many domains other than text-editing. In prior work [42], we formalized PBD in the email domain as an inductive learning problem using both a version space approach as well as an inductive logic programming approach. Compared to the previous work, this work combines a more expressive and flexible bias with a more natural and straightforward domain representation.

Lieberman's Tatlin system [49] infers user actions from observed state changes in an application. Tatlin learns how to cut and paste data from a calendar application to rows in a spreadsheet. Lieberman also discusses the problem of constructing appropriate data descriptions for the objects manipulated by the PBD system, such as "the window the user clicked on" or "the window named `Mail`"; perhaps the most important aspect of SMARTedit's learning component is its ability to infer the correct data description from examples. See Halbert [25] for additional early work on this topic.

SMARTedit is based on the premise that the user is deeply involved in the process of teaching the system how to perform a task, and willing to supervise the learning and execution of the system. In contrast, Ruvini and Dony's work on APE [74] uses version space learning to identify not only which actions a user wants to perform in a Smalltalk programming environment, but also *when* to execute those actions. Learning to identify when a particular sequence of text-editing actions is applicable is an important direction for future research.

Bauer *et al.* [6, 7] present TrIAs, a PBD system for learning wrappers for the purpose of training an information integration system. TrIAs is designed around a collaborative user interface in which a user teaches the system how to extract information from a web site by labelling a single example page. Unlike SMARTedit, TrIAs performs no generalization from multiple examples, instead using both heuristics and user hints to choose a hypothesis that is consistent with each part of the example.

The Eager [15] system detects and automates repetitive actions in Hypercard applications. Each user action is compared against the action history using a pattern-matching scheme. Once the system detects a repetitive sequence, it highlights its prediction of the next action, and offers to automate the remainder of the sequence. Peridot [61] allows programmers to construct user interfaces by drawing them interactively. It uses a set of condition-action rules to determine when to infer constraints over widget placement and iterations over sequences. Chimera [37] generalizes constraints between graphical objects from multiple snapshots and provides a macro-by-example facility for creating macros in a graphical editing domain. Tinker [48] supports programming Lisp by demonstration, but performs no inference, instead relying on the user to disambiguate the examples.

These PBD systems all rely on heuristic rules to describe when a generalization is appropriate. These heuristics make PBD systems (like the rule-based expert systems on which they are modelled) brittle, laborious to construct, and difficult to extend. For instance, Eager is hardcoded to recognize a fixed set of sequences over which a user may iterate, such

as the email messages in a mailbox or the days of the week. Peridot uses a set of domain-dependent condition-action rules to infer constraints and loops. Unfortunately, these fragile heuristics must be hand-crafted for each domain. In contrast, our framework uses novel machine learning techniques to perform robust trace generalization. As we demonstrate in the remainder of this thesis, our approach also yields low sample complexity—SMARTedit is capable of generalizing from a very small number of training examples.

Others have considered applying domain-independent algorithms to PBD. Witten *et al* [89] identify shortcomings in current machine learning approaches when applied to PBD, such as an inability to take advantage of domain knowledge or user hints. Paynter sketches out a general-purpose PBD framework using machine learning [68]. Nevill-Manning and Witten [63] describe a linear-time algorithm for detecting hierarchical structure in sequences by generalizing a grammar from repeated subsequences in a single example. Their algorithm is elegantly simple, but it is not obvious how to apply background knowledge to bias the generated grammars.

Paynter's thesis [69] describes the Familiar system, a domain-independent PBD framework, based on AppleScript, for automating repetitive tasks. Familiar employs decision tree learning algorithms to generalize a single "best" program that predicts the user's intended actions.

2.3 Adaptive user interfaces

Our work is similar to previous machine learning efforts at developing self-customizing software. Schlimmer and Hermens [75] describe a note-taking system that predicts what the user is going to write and facilitates completion; the system works by learning a finite state machine (FSM) modeling note syntax, and decision tree classifiers for each FSM state. Maes and Kozierok [51] use nearest-neighbor learning (adjusted by user feedback) to predict the user's next action from the action most recently executed, but in contrast to our work there is no attempt to learn loops or sequences of actions.

Rich and Sidner [73] describe the COLLAGEN collaborative agent system. COLLAGEN employs a hierarchical plan library structured according to a user's goals and intentions, and uses it to model user tasks. Lesh, Rich, and Sidner [43] apply plan recognition in COLLAGEN to infer a user's task from her actions. Using the task model, COLLAGEN enters into a collaborative discourse with the user to help her achieve her goals. In our system, the collaborative task is to teach SMARTedit how to perform a particular repetitive edit; we have begun work on a mixed-initiative interface for SMARTedit that could reduce user effort during this training session. See Wolfman *et al.* [90] for preliminary results.

Chapter 3

A FORMAL MODEL OF PROGRAMMING BY DEMONSTRATION

We cast programming by demonstration as the machine learning problem of inferring program statements based on examples of the state changes resulting from demonstrated actions. We assume that we can determine the application state before and after each user action; this is equivalent to recording the state at the beginning of the demonstration, and observing the sequence of actions the user performs. Each demonstration thus generates a sequence of states representing the changes that arise as the user demonstrates the target program.

Let a *state* be the environment visible to our system during the execution of a program. For example, in the text-editing domain the state contains the contents of the text buffer, the cursor position, contents of the clipboard, and the text selection if any. We formalize a *program* as a procedure that, given an initial state S_0, produces a *trace*—a sequence of states $S_0, S_1, ..., S_n$ resulting from the execution of the procedure on S_0, such that the execution of a single program statement in state S_i produces state S_{i+1}. The program's inputs and outputs are encoded in the trace; the contents of the state may include both outputs of prior computations as well as inputs to upcoming computations. A *training example* is a trace $\langle S_0, S_1, S_2, S_3, ..., S_n\rangle$. We say that a program is *consistent* with an example trace $T = \langle S_0, S_1, S_2, ..., S_n\rangle$ iff the program, when executed in state S_0 and given the inputs included in T, produces T as its trace.

Let the bias be the language denoting the set of allowable programs. We state the learning problem as follows:

> Given a language and one or more traces of a target program, output a program or programs in the language that is consistent with the examples and generalizes correctly to future examples.

In this thesis we focus on text-editing as a fruitful domain for programming by demonstration. Text-editing actions are functions that transform the application state into a new state. We model the application state of a text editor as a tuple (T, L, P, E), where T denotes the contents of the text buffer (a sequence of tokens in some alphabet, usually the ASCII character set), L denotes the cursor location (a tuple (R, C) representing a row and column position within the text buffer), P denotes the contents of the cliPboard (a sequence of tokens), and E denotes a contiguous region in T representing the sElection (highlighted text).

We define a comprehensive set of text-editing actions, including: moving the cursor to a new location, deleting a region of text, inserting a sequence of tokens, selecting a region of text, cutting and copying the selection to the clipboard, and pasting the contents of the clipboard at the current cursor location.

For example, moving the cursor from row 5, column 0 to row 6, column 0 causes the application state to change from one in which the cursor is at position (5, 0) to one in which the cursor is at position (6, 0). The action "move to the next line" is consistent with this state change, as is the action "move to the beginning of line 6". However, the action "move to line 7" is not consistent with this change, nor is the action "insert the string `hello`". Given the single state change in this example, the learning algorithm outputs the set of actions consistent with the example.

This thesis describes a novel machine learning approach for inferring program statements from examples of the application state before and after each action. We introduce a method for learning these functions called version space algebra. Using version space algebra, we build up a complex search space of functions by composing smaller, simpler version spaces. In the work described in this thesis, the component version spaces are either maintained using boundary sets, or small enough to be enumerated. Future applications may require heuristic search and/or sampling of some version spaces. We expect that version space algebra will still be useful in this context, by helping to subdivide a large learning problem into simpler ones, and allowing the use of heuristic solutions only in those subproblems where they are unavoidable, rather than employing a heuristic solution for the entire problem.

The version space algebra takes a divide-and-conquer approach to learning complex

procedures. For example, a function that moves the cursor to a new position in the text file could be learned as the combination of two simpler functions: one that predicts the new row position, and one that predicts the column position. In the next section we focus on the problem of learning a single action.

3.1 Learning actions

We learn programs by learning each statement individually in sequence, based on examples of the state before and after the statement is executed. Each program statement is a *function* that maps from one program state to another. A statement is consistent with a pair of states $\langle S_i, S_j \rangle$ iff when executed in state S_i the statement produces state S_j. For example, performing the action of moving the cursor in one state produces another state in which the cursor is at a different location.

Programming by demonstration requires a machine learning algorithm capable of inferring functions that map complex objects into complex objects. Most machine learning algorithms have focused on the problem of classification learning—learning functions that map from a complex object to a single discrete value. In contrast, the problem of programming by demonstration requires an algorithm capable of learning functions that map from one complex object to another—such as one application state to another.

Our framework is based on Mitchell's version space formulation [55], in which concept learning is defined as a search through a version space of consistent hypotheses. A concept learning hypothesis is a function that maps from an object into a binary classification.

Our work extends the version space formulation to learning functions that map from complex objects to complex objects, such as from one application state to another. For example, we want to learn a programming instruction that is consistent with the user's action of moving the cursor from one location to another in the text file, or an instruction for inserting a particular sequence of characters.

One might think that concept learning suffices for this learning problem. For example, one may classify tuples consisting of (input, output) states into those that exhibit the correct transformation, and those that do not. However, although this representation could be used in learning to classify (input, output) state pairs, in practice one is not given the output state to classify. The learner must be able to act on a novel input state, and produce a plausible output state, not merely classify a pair of proposed states. Moreover, the complexity and variability of these states prohibits a generate-and-test methodology.

In order to introduce our version space extensions we first define our terminology. A *hypothesis* is a function that takes as input an element of its domain I and produces as output an element of its range O. A *hypothesis space* is a set of functions with the same domain and range. The *bias* determines which subset of the universe of possible functions is part of the hypothesis space; a stronger bias corresponds to a smaller hypothesis space. We say that a training example (i, o), for $i \in domain(h)$ and $o \in range(h)$, is *consistent* with a hypothesis h if and only if $h(i) = o$. A *version space*, $\mathrm{VS}_{H,D}$, consists of only those hypotheses in hypothesis space H that are consistent with the sequence D of examples. When a new example is observed, the version space must be *updated* to ensure that it remains consistent with the new example. We will omit the subscript, and refer to the version space as VS, when the hypothesis space and examples are clear from the context.

In Mitchell's original version space approach, the range of hypotheses was required to be the Boolean set $\{0, 1\}$, and hypotheses in a version space were partially ordered by their generality. A hypothesis h_1 is more general than another h_2 iff the set of examples for which $h_1(i) = 1$ is a superset of the examples for which $h_2(i) = 1$. Mitchell showed that this partial order allows one to represent and update the version space solely in terms of its most-general and most-specific boundaries G and S (*i.e.*, the set G of most general hypotheses in the version space and the set S of most specific hypotheses). The consistent hypotheses are those that lie between the boundaries (*i.e.*, every hypothesis in the version space is more specific than some hypothesis in G and more general than some hypothesis in S).

Mitchell's approach is appropriate for concept learning problems, where the goal is to predict whether an example is a member of a concept. We extend the approach to any supervised learning problem (*i.e.*, to learning functions with any range) by allowing arbitrary partial orders. We base this proposal on the observation that the efficient representation of a version space by its boundaries only requires that some partial order be defined on it, not necessarily one that corresponds to generality. The partial order to be used in a given

version space must be provided by the application designer, or by the designer of a version space library. The corresponding generalizations of the G and S boundaries are the least upper bound and greatest lower bound of the version space. As in Mitchell's approach, the application designer provides an update function $U(\mathrm{VS}, d)$ that shrinks VS to hold only the hypotheses consistent with example d.

We say that a version space is *boundary-set representable* (BSR) if and only if it can be represented solely by its least upper bound and greatest lower bound. Hirsh [30] showed that the properties of convexity and definiteness are necessary and sufficient for a version space to be BSR.

3.2 Version space algebra

We now introduce an algebra over these extended version spaces. Our ultimate goal is to learn functions that transform complex objects into complex objects. However, rather than trying to define a single version space containing all the functions of interest, we build up the complex version space by composing together version spaces containing simpler functions. Just as two functions can be composed to create a new function, two version spaces can be composed to create a new version space, containing functions that are composed from the functions in the original version spaces. With this modular representation, we enable the creation of *version space libraries*—collections of reusable components that can be combined in different ways to construct new machine learning applications.

We define an *atomic version space* to be a version space as described in the previous section, *i.e.*, one that is defined by a hypothesis space and a sequence of examples. We define a *composite version space* to be a composition of atomic or composite version spaces using one of the following operators.

Definition 1 (Version space union) *Let H_1 and H_2 be two hypothesis spaces such that the domain (range) of functions in H_1 equals the domain (range) of those in H_2. Let D be a sequence of training examples. The* version space union *of $VS_{H_1,D}$ and $VS_{H_2,D}$, $VS_{H_1,D} \cup VS_{H_2,D}$, is equal to $VS_{H_1 \cup H_2,D}$.*

Unlike the BSR unions proposed by Hirsh [30], we allow unions of version spaces such that the unions are not necessarily boundary-set representable, by maintaining component version spaces separately; thus, we can efficiently represent more complex hypothesis spaces.

Theorem 1 (Efficiency of union) *The time (space) complexity of maintaining the union is on the order of the sum of the time (space) complexity of maintaining each component version space.*

Proof: The theorem follows directly from the fact that the union is maintained by maintaining each component version space separately. □

In order to introduce the next operator, let $C(h, D)$ be a consistency predicate that is true when hypothesis h is consistent with the data D, and false otherwise. In other words, $C(h, D) \equiv \bigwedge_{(i,o)\in D} h(i) = o$.

Definition 2 (Version space join) *Let* $D_1 = \{d_1^j\}_{j=1}^n$ *be a sequence of* n *training examples each of the form* (i, o) *where* $i \in$ *domain*(H_1) *and* $o \in$ *range*(H_1)*, and similarly for* $D_2 = \{d_2^j\}_{j=1}^n$*. Let* D *be the sequence of* n *pairs of examples* $\langle d_1^j, d_2^j\rangle$*. The join of two version spaces,* $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$*, is the set of ordered pairs of hypotheses* $\{\langle h_1, h_2\rangle | h_1 \in VS_{H_1,D_1}, h_2 \in VS_{H_2,D_2}, C(\langle h_1, h_2\rangle, D)\}$.

In many domain representations, the consistency of a hypothesis in the join depends only on whether each individual hypothesis is consistent with its respective training examples, and not on a dependency between the two hypotheses in a pair. In this situation we say there is an *independent join.*

Definition 3 (Independent join) *Let* $D_1 = \{d_1^j\}_{j=1}^n$ *be a sequence of* n *training examples each of the form* (i, o) *where* $i \in$ *domain*(H_1) *and* $o \in$ *range*(H_1)*, and similarly for* D_2*. Let* D *be the sequence of* n *pairs of examples* $\langle d_1^j, d_2^j\rangle$*. Iff* $\forall D_1, D_2, h_1 \in H_1, h_2 \in H_2\ [C(h_1, D_1) \wedge C(h_2, D_2) \Rightarrow C(\langle h_1, h_2\rangle, D)]$*, then the join* $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$ *is independent.*

If the join is independent, then the hypotheses in the version space join are exactly the hypotheses in the Cartesian product of the two component version spaces, and the join may be updated by updating each of the two component version spaces individually. However, not all joins are independent. For instance, given VS_1 containing hypotheses $\{A, B\}$, and

VS_2 containing $\{X, Y\}$, even though A and X are consistent with their respective data, it is not always the case that $\langle A, X\rangle$ is consistent with the joint data. Although we have not yet formalized the conditions under which joins may be treated as independent, Chapter 4 gives several examples of independent joins in the text editing domain, and Section 4.4.4 gives an example of a dependent join.

Joins provide a powerful way to build complex version spaces, but a question is raised about whether they can be maintained efficiently. Let $T(\mathrm{VS}, d)$ be the time required to update VS with example d. Let $S(\mathrm{VS})$ be the space required to represent the version space VS (perhaps with boundary sets).

Theorem 2 (Efficiency of join) *Let* $D_1 = \{d_1^j\}_{j=1}^n$ *be a sequence of* n *training examples each of the form* (i, o) *where* $i \in$ *domain*(H_1) *and* $o \in$ *range*(H_1)*, and let* d_1 *be another training example of the same type. Define* $D_2 = \{d_2^j\}_{j=1}^n$ *and* d_2 *similarly. Let* D *be the sequence of* n *pairs of examples* $\langle d_1^j, d_2^j\rangle$*, and let* $VS_{H_1,D_1} \bowtie VS_{H_2,D_2}$ *be an independent join. Then* $\forall D_1, d_1, D_2, d_2$

$$\begin{aligned} S(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}) &= S(VS_{H_1,D_1}) + S(VS_{H_2,D_2}) + C_1 \\ T(VS_{H_1,D_1} \bowtie VS_{H_2,D_2}, \langle d_1, d_2\rangle) &= T(VS_{H_1,D_1}, d_1) + T(VS_{H_2,D_2}, d_2) + C_2 \end{aligned}$$

for some constants C_1 *and* C_2.

Proof sketch: If the join is independent, then the set of hypotheses in the join is the cross product of the hypotheses in the two component spaces. In this case, it may be maintained by maintaining the two component version spaces separately. The time and space requirements for maintaining the join are then merely the sum of the time and space requirements for each component version space, plus a constant factor. □

Note that our union operation is both commutative and associative, which follows directly from the properties of the underlying set operation. The join operator is neither commutative nor associative. Although we do not define an intersection operator because we have found no application for it thus far, its definition is straightforward.

The functions in a child version space may need to be transformed into a different type before inclusion in the parent version space. For example, functions in the child version space may act on only a portion of the full input available to the parent, and they may return values in a simpler domain. For this purpose, we introduce the transform operator. Transforms may be used between a parent and a single child in the absence of other version space operators, or they may be associated with each child in a union or join.

Definition 4 (Version space transform) *Let τ_i be a mapping from elements in the domain of VS_1 to elements in the domain of VS_2, and τ_o be a one-to-one mapping from elements in the range of VS_1 to elements in the range of VS_2. Version space VS_1 is a* transform *of VS_2 iff $VS_1 = \{g | \exists_{f \in VS_2} \forall_i\, g(i) = \tau_o^{-1}(f(\tau_i(i)))\}$.*

Transforms are useful for expressing domain-specific version spaces in terms of general-purpose ones, or more generally, converting functions of one type to functions of another type. For example, in SMARTedit, a Location version space denotes a set of functions that output locations in the text buffer relative to the cursor position in the input state. The Move version space transforms a Location version space into a set of functions that move the cursor to a new location, and the Select version space transforms a Location into functions that select from the current cursor position to a new location. See Chapter 4 and Appendix A for more examples.

3.3 PAC analysis

Having defined the version space algebra, we now question whether or not these complex version spaces can be learned efficiently. PAC learning theory [83, 26, 34] studies the conditions under which a reasonably-correct concept can be learned with reasonable confidence, or in other words, be *probably approximately correct*. Although PAC analysis was originally developed for concept learning, it is easily extended to complex-function learning.

We assume inputs are drawn from an unknown probability distribution $\mathcal{D}$. We have no guarantee that the learned function will be close to the target function, because it might be based on misleading similarities in the observed sequence of training examples. Unless the learner is given training examples corresponding to every possible element in its domain, there may be multiple hypotheses consistent with the given training examples, and the learner is not guaranteed to output the correct one. For example, we might have as input a list of papers all by the same author, and learn a function that assumes that all authors share the same first and last names.

The true error of a hypothesis h, $error_{\mathcal{D}}(h)$, with respect to target function f and distribution $\mathcal{D}$ is the probability that h will produce the wrong output when applied to an input i drawn at random according to $\mathcal{D}$:

$$error_{\mathcal{D}}(h) \equiv \Pr_{i \in D}[f(i) \neq h(i)] \tag{3.1}$$

Let ϵ denote an upper bound on the error of the learned hypothesis, and δ be an upper bound on the probability that the learner will output a hypothesis with error greater than ϵ. Given desired values for ϵ and δ, PAC analysis places a bound on the number of examples required for learning.

Definition 5 (PAC learnability) *For a target function class F defined over a set of inputs I of length n and a learner L using hypothesis space H, F is* PAC-learnable *by L using H iff for all $f \in F$, distributions $\mathcal{D}$ over I, ϵ such that $0 < \epsilon < 1/2$, and δ such that $0 < \delta < 1/2$, learner L will output a version space VS such that $\forall h \in VS$, $error_{\mathcal{D}}(h) \leq \epsilon$ with probability at least $(1 - \delta)$, using a sample size and time that are polynomial in $1/\epsilon$, $1/\delta$, n, and $size(f)$.*

Blumer *et al.* [9] derive the following relationship between the required sample size m, the parameters ϵ and δ, and $|H|$:

$$m > \frac{1}{\epsilon}(\ln |H| + \ln(1/\delta)) \tag{3.2}$$

Thus a necessary and sufficient condition for m to be polynomial when using a hypothesis space H is that $|H| = O(e^{P(n)})$, where $P(n)$ is an arbitrary polynomial function of n.

We say that a version space operator $\diamond$ *preserves* PAC learnability if, given function class F_1 that is PAC-learnable using H_1, and function class F_2 that is PAC-learnable using H_2, $F_1 \diamond F_2$ is PAC-learnable using $H_1 \diamond H_2$.

Theorem 3 *The version space union, join, and transform operators preserve PAC learnability.*

Proof sketch: To show that each operator preserves PAC learnability, it suffices to show that the number of examples and the time required to maintain the composite space remain polynomial in $1/\epsilon$, $1/\delta$, n, and size(f).

Let F_1 be a function class that is PAC-learnable using H_1, and similarly for F_2 and H_2. Then $|H_1| = O(e^{P_1(n)})$ and $|H_2| = O(e^{P_2(n)})$. For the union, $|H_1 \cup H_2| \le |H_1| + |H_2| = O(e^{P_1(n)}) + O(e^{P_2(n)}) = O(e^{\max(P_1(n),P_2(n))})$. For the join, $|H_1 \bowtie H_2| \le |H_1| \times |H_2| = O(e^{P_1(n)}) \times O(e^{P_2(n)}) = O(e^{P_1(n)+P_2(n)}) = O(e^{P_{12}(n)})$. The transform is a one-to-one mapping so $|H|$ remains constant across a transform. Since the size of the composite hypothesis space remains at most exponential in $P(n)$, the number of training examples remains polynomial in $\ln|H|$.

If F_1 is PAC-learnable using H_1, then the time required to learn it is $O(P(n))$, and similarly for F_2 and H_2. By Theorem 1 and the previous paragraph, the time required to learn $F_1 \cup F_2$ using $H_1 \cup H_2$ is the sum of the times required to learn F_1 and F_2, which is polynomial in n. By Theorem 2 and the previous paragraph, the time required to learn $F_1 \bowtie F_2$ using $H_1 \bowtie H_2$ is polynomial in the sum of the times required to learn F_1 and F_2. The transform operator is a one-to-one mapping, so it requires $o(n)$ time to maintain. Since the number of examples and time remain polynomial across the union, join, and transform operators, the operators preserve PAC learnability. □

Note that the join of an unlimited number of version spaces does not preserve PAC-learnability, because the required number of examples is worst-case exponential in the number of version spaces being joined.

3.4 Version space execution

We have described how to define and update version spaces; next we discuss how to use them to make useful predictions in novel contexts. In a programming by demonstration context, we are interested not only in the literal hypotheses in a version space, but in the result of applying those hypotheses to a novel input. We say that a version space is *executed*

on an input by applying every hypothesis in the version space to that input, and collecting the multiset of resulting outputs. More formally, the result of an execution of a version space V on an input i is denoted by $\text{exec}_V(i)$ and defined as follows:

$$\text{exec}_V(i) = \{o : \exists f \in V, f(i) = o\} \tag{3.3}$$

Note that two distinct hypotheses may agree in their predictions when applied on a new input. In section 4.1, we discuss how version space execution is used to communicate the contents of the version space to the user for critiquing.

3.5 Probabilistic framework

We have constructed a probabilistic framework around the version space algebra that assigns a probability to each hypothesis in the version space. Such a probabilistic framework allows us to:

- *Choose better hypotheses during execution*: Even before the version space converges to a single hypothesis, it may still be executed on new instances to produce a set of consistent outputs. The probabilistic framework allows us to choose the most likely output.

- *Introduce domain knowledge*: An application designer probably has *a priori* knowledge about which hypotheses are more likely to occur than others. For example, when trying to reason about a user's decision to move the cursor through an HTML file, a hypothesis of "634 characters forward" is probably less likely than the hypothesis "after the next occurrence of the string `<TABLE>`". This knowledge can be gathered either empirically, such as by conducting user studies, or it may be set heuristically by the application designer. The probabilistic framework allows the designer to cleanly introduce this type of domain knowledge into the system. These probabilities can also be used to adapt to the user over time by giving higher probability to the types of hypotheses she employs.

- *Provide support for noisy data*: In a traditional version space, inconsistent hypotheses have zero probability and are removed from consideration. In a probabilistic framework, one can assign a small but non-zero probability to inconsistent hypotheses, such that noisy data does not irretrievably remove a hypothesis from the version space.

We define the probability of a hypothesis in a version space V, given V, as the probability that the hypothesis is true, given that the true mapping between domain(V) and range(V) is one of the hypotheses in V. To use the probabilistic framework, the application designer provides *a priori* probabilities as follows:

- A probability distribution $\Pr(h|H)$ for each hypothesis h in an atomic hypothesis space H, such that $\sum_{h \in H} \Pr(h|H) = 1$.

- A probability distribution $\Pr(V_i|W)$ for each version space V_i in a union W, such that $\sum_i \Pr(V_i|W) = 1$.

Each of these probability distributions defaults to the uniform distribution, if no prior knowledge is available about the distributions of the hypotheses in the version space.

The probability of a hypothesis h given an atomic version space V, $P_{h,V}$, is initialized to $\Pr(h|H)$. As the atomic version space is updated, the inconsistent hypotheses are removed from the version space. The probabilities of the remaining hypotheses are then rescaled to sum to 1. We say that the probability of a consistent hypothesis relative to the version space is $\Pr(h|V)$.

Thus the overall probability of a hypothesis h given a version space V, $P_{h,V}$, is defined inductively up from the bottom of the version space hierarchy depending on the type of V:

Atomic V is atomic. In this case, $P_{h,V} = Pr(h|V)$.

Transform V_1 is a transform of another version space V_2. Thus $V_1 = \{h | \exists_{f \in V_2} \forall_i \; h(i) = \tau_o^{-1}(f(\tau_i(i)))\}$. Since transforms are one-to-one, $P_{h,V_1} = P_{f,V_2}$.

Union V is a union of version spaces V_i with corresponding probabilities w_i. In this case, $P_{h,V} = w_i \times P_{h,V_i}$.

Join V is a join of a finite number of version spaces V_i. Thus, $V = \{\langle h_1, h_2, \ldots, h_n \rangle | h_i \in V_i, \mathrm{C}(\langle h_1, h_2, \ldots, h_n \rangle, D)\}$. In the general case, $P_{h,V}$ is the joint probability of the hypotheses $h_1, h_2, \ldots, h_n$ given the joined space. If, for all i, the probability of a hypothesis in V_i given V_i is independent of all the hypotheses in all the other version spaces in the join, then $P_{h,V} = \prod_i P_{h_i,V_i}$. Independence of the version spaces is a necessary but not sufficient condition for this to be the case.

Using this framework, each hypothesis in the version space has an associated probability. If the execution of a version space V on an input i produces a set of outputs o, we assign a probability P_V^o to each of these outputs:

$$P_V^o(i) = \sum_{\forall f | f(i) = o} P_{f,V} \tag{3.4}$$

We employ version space execution and the probabilistic framework to present the results of the learner to the user. Our SMARTedit system presents a list of output states to the user, ranked by probability, and allows her to select the correct one by cycling through the different output states. Chapter 4 describes SMARTedit in more detail.

3.6 Related work in machine learning

To situate this work in context, we review related work in a number of areas. One of the main benefits of the version space approach is to make it easier for application designers to formulate their problems as machine learning problems. By providing the ability to express not only classification functions, but arbitrary complex functions, the representation moves closer to concepts in the target application domain. Langley and Simon [41] discuss the difficulty of formulating a problem in a representation amenable to classification, identify several domains where non-classification learning may be applied, and describe a number of early attempts to do so.

Most closely related to our work is work on extensions to Mitchell's version spaces. Further afield, we summarize related work on learning programs, of which our work is an instance. Our probabilistic framework has similarities to other work on stochastic learning methods; we summarize those next. Finally, we relate this work to existing research on wrapper induction and information extraction.

3.6.1 Extensions and applications of version spaces

Our version space algebra is related to Hirsh's work on version spaces. For example, Hirsh [30] studies the algebra of boundary-set representable version spaces. Hirsh, Mishra, and Pitt [31] propose maintaining version spaces without boundary sets by representing instead the set of positive and negative examples explicitly, and computing operations over such version spaces by working directly with the training examples. Unlike Hirsh's work, we have extended version spaces beyond concept learning by allowing any partial order, defined the join operator, and allowed unions and intersections that are not representable by a single pair of boundary sets.

Norton and Hirsh [65] extend version spaces to deal with noisy data by viewing version space update as incremental version space merging. In this fashion they construct a collection of version spaces, each of which is consistent with some subset of the data; using a noise model, they assign probabilities to each of the generated version spaces. Applying this approach to our hierarchical version spaces and thereby extending them to handle noisy data is an area for future work.

VanLehn and Ball [84] have used version spaces to induce context-free grammars from examples. Since generality is undecidable for context-free grammars, and the size of the S and G sets could be infinite, VanLehn and Ball define an alternative partial order over reduced grammars, which they call FastCovers. This partial order leads to a finite boundary set representation that contains a superset of the set of reduced grammars consistent with the examples.

Subramanian and Feigenbaum [80] describe version space factorization, which is very similar to our version space join. Rather than using the factored version space to express more complex hypotheses, however, they use factorization within the concept-learning framework to choose a sequence of instances that speeds convergence of the version space.

Smith [79] proposes a knowledge integration framework for concept learning that subsumes Mitchell's candidate elimination algorithm and Hirsh's version space merging framework. The central idea is to unify examples, bias, and domain theories into a single model, by representing each bit of knowledge in terms of the set of of hypotheses consistent with that knowledge. Any form of knowledge can be combined with any other using Hirsh's incremental version space merging algorithm [29].

3.6.2 Comparison with plan recognition

We view programming by demonstration as a special case of the plan recognition problem. Plan recognition [33, 11, 85] aims to understand an agent's plan or goal based on a partial observation of the agent's behavior. In an intelligent user interface, this knowledge may then be used to predict the agent's further actions or suggest more optimal methods for accomplishing the goal.

The issue of whether programs are plans has been debated at length (for example, see [1]); however, for the purposes of this section we assume they are equivalent: a sequence of actions. Thus PBD can be viewed as the problem of observing a prefix of these actions, and generalizing to the complete plan.

Plan recognition systems typically require the use of a plan library: the set of plans from which the agent's target plan is assumed to be drawn. This plan library defines the plan recognizer's bias, or the space of hypotheses considered during the search. Kautz and Allen [33] employ a plan library in the form of a hierarchy of plans connected via abstraction and decomposition relationships.

A second approach to building a plan library has been to represent plans as sentences in a grammar and phrase the plan recognition problem as parsing in a grammar [85, 71]. The parsing-based approaches suffer from the drawback that plans must be totally ordered, which may cause an exponential increase in the size of the plan library (to represent all total orderings over a partially ordered plan).

In order to scale to large domains, a third approach has investigated automatic construction of the plan library, such as proposed by Lesh and Etzioni [46, 47, 44, 45]. They point out the similarities between goal recognition and concept learning: given a sequence of actions, find the goal or goals consistent with the observations. Our approach falls into this category. The basic idea is to provide a set of primitives and a set of rules for composing together the primitives to construct hypothesis space implicitly. Related to automatic construction is the idea of efficient search through this space.

The choice of the plan library representation influences the types of plans that may be recognized by the system. In the Kautz & Allen representation, arbitrary plans may be represented; however, manual construction of the plan library limits the scalability of this approach. On the other hand, automatic construction of the library can scale up to larger domains, but at the cost of reduced expressiveness. Our system is only capable of recognizing repetitive plans. However, many previous plan recognition systems assume that the agent is acting optimally, and only include optimal plans in the library. In contrast, we make no assumptions about the agent's behavior; our system is capable of recognizing the user's intent, whether or not the user performs optimally.

Another difference between the version space algebra approach and previous plan recognition systems is that we maintain the set of plans consistent with the observations, rather than finding a single consistent plan; and we allow probabilistic execution before the version space converges to a single plan. The Kautz and Allen approach produces the set of consistent plans in which the number of top-level actions is minimized, but provides no way to choose between them. In contrast, Charniak and Goldman [10] present a probabilistic method for deciding between plan hypotheses, but do not address the problem of determining which plans are consistent with the data. Lesh and Etzioni maintain the set of all consistent goals, and support execution before convergence in special cases (when all goals in the version space are necessary or sufficient for the target goal); however, they provide no general mechanism for probabilistic execution or introduction of prior probabilities on hypotheses.

Our approach can be divided into three phases:

- Translation: converting from a sequence of low-level keypress and mouse actions to a sequence of state changes representing high-level actions;

- Learning: inducing high level actions from multiple examples of the state changes associated with each action; and

- Recognition: finding a repetitive plan consistent with the induced actions.

While one could do plan recognition over the low-level actions (such as sequences of cursor key presses), we believe that our method of abstracting away from the exact sequence of low-level actions leads to a representation closer to the user's model of the application, and thus more robust and understandable plans.

Most previous plan recognition systems perform only the third phase: recognizing a plan given a sequence of actions. In our work, the third phase is trivial: once the actions are known, the plan is merely a repetition of the sequence of actions. Instead, we have focused our efforts on the first two phases.

Lesh and Etzioni's approach is the most similar to our work; we compare the two systems in more detail in the remainder of this section. The BOCE [47] system uses version spaces for goal recognition. Actions are represented in a STRIPS-like formalism [21], and a goal is a conjunction of literals that must be established by a plan (legal sequence of actions[1]). A goal is consistent with a sequence of observed actions if there exists a pseudo-optimal plan[2] containing those actions that, when executed in some state, asserts the goal. While BOCE represents goals explicitly in a predicate description language, our approach has no explicit goal representation.

Both BOCE and our approach use a version space representation of an implicit plan library, but the elements of each version space are very different. BOCE's version space contains *goals* ordered by generality, where each goal is a conjunction of literals.[3] In contrast, our approach constructs a version space of plans where each plan is represented as a function from state to state. Our version space algebra has a number of benefits over a regular version space: other partial orderings, not necessarily generality; representation of version spaces

[1] More formally, a partially ordered set with causal links [88].

[2] Every action in the plan must support some action in the plan or the goal. This requirement is weaker than having the plan be minimal and contain no irrelevant actions; in particular, it allows redundant sensory actions.

[3] Note that the use of a STRIPS-like language is essential, but many domains (such as text editing) may be difficult to encode in this formalism.

with “holes”; combination of boundary-represented version spaces with enumerated version spaces in the hierachy; the join operator and decomposition of the target function into simpler components.

3.6.3 Learning programs

Our work is closely related to automatic programming, e.g. [77, 78, 8, 81, 23, 5, 52]. Unlike language-specific systems, our version space algebra approach provides a general method for learning programs in a wide variety of languages Our approach also takes advantage of version space decomposition and boundary set representability to efficiently conduct an exhaustive search of the space of programs.

Genetic programming [36] focuses on learning programs using genetic algorithms. It differs fundamentally from ours in the type of search used; empirically comparing the two is an item for future work.

Reinforcement learners [82] can also be viewed as learning a class of programs. They have the major advantage of being robust to noisy data, but the class of programs they learn is quite limited. Rather than being instructed on each action, a reinforcement learner assumes only the presence of a “reward” function; entering desirable states increases its reward. Work on using partial program specifications to speed up reinforcement learning [2, 67, 18] is more closely related to our work. In these cases, the agent’s behavior is controlled by giving it a partial, hierarchical program description. Andre and Russell [2] have taken steps to broaden the set of program descriptions to include higher-level programming features such as parameterization, aborts and interrupts, and memory variables. Their approach for incorporating this prior knowledge into the learner is akin to our work in that it better crafts the learner’s bias to facilitate learning.

Our work is also related to research on learning for problem solving and explanation-based learning [56]. EBL algorithms are capable of generalizing from a single example with the help of a domain theory. For example, Shavlik [76] describes an explanation-based learning system that generalizes the concept of number. EBL is similar to version space algebra in that it has been used in learning for problem-solving, and is capable of learning

from a single example. While EBL learns by reformulating a single domain theory, however, the version space algebra maintains a range of possible theories.

Our goals resemble Andreae’s [3], who describes constraint-based techniques for learning robotic assembly procedures from traces. However, that work uses domain-specific heuristics to match key events and guide generalization.

The role of version space algebra in learning programs in procedural languages is similar to that of declarative biases in inductive logic programming [62]. While most ILP systems maintain and return only a single theory, the version space algebra maintains all theories efficiently. In prior work [42], our $\text{TG}\textsc{en}_{\text{FOIL}}$ system employed the FOIL ILP algorithm to learn programs by demonstration in the email domain. Although we have not directly compared the two approaches on the same domain, we have found that the programs produced with version space algebra are generally easier to understand, and map more directly to a natural domain representation, than did the programs learned with $\text{TG}\textsc{en}_{\text{FOIL}}$.

3.6.4 Probabilistic models

Hidden Markov models [72] provide a different framework for training a model to recognize sequences of actions. Solutions based on HMMs assume a generative model of output sequences and train the model on example sequences to learn the parameters of the model. HMMs have been widely used in user modelling, including predicting web page fetches [91], and detecting computer network intrusions [40]. In contrast to the flat sequence models learned by HMMs, our approach learns sequences with program structure such as loops and conditionals.

Programming by demonstration is similar to previous work on predicting Unix command lines. Several researchers have constructed probabilistic models for predicting a user’s next command. Davison and Hirsh [17, 28] use naïve Bayes to construct a matrix that computes the probability of the next command based on the previous command. They then update the probabilities in this matrix using an update function with an exponential decay such that older commands in the history are weighted less than more recent commands. Korvemaker and Greiner [35] expand on this work, using a mixture of experts to enhance prediction.

3.6.5 *Wrapper induction and information extraction*

Another body of related work is on techniques for wrapper induction and information extraction [39, 4, 14, 32]. The types of procedures learned while extracting information from semi-structured text files are very similar to the types of procedures learned by SMARTedit. In fact, several of the scenarios used to evaluate SMARTedit come directly from the RISE repository of information extraction tasks [58].

Wrapper induction procedures are similar to text-editing procedures. Kushmerick [38] defines a variety of wrapper classes of increasing functionality. Kushmerick's LR wrapper class denotes a set of procedures that is a subset of the procedures that SMARTedit can learn without loop constructs, and his HLRT wrapper class is a subset of the procedures SMARTedit can learn with loop constructs.

Muslea *et al.* [59] describe the STALKER system that learns wrappers by example using a rule-based set-covering approach. The system learns hierarchical wrappers from as little as two training examples. Muslea [60] has also investigated selective sampling (a form of active learning) to use multiple views to pick the training example that is likely to cause the version space to converge the most quickly. A promising direction of future work is to apply active learning to SMARTedit; we have already taken some steps towards this goal [90].

Bauer [6] describes a programming by demonstration system for wrapper induction. Based on a visual selection of the target information, the system constructs a wrapper in the HyQL language. Unlike our approach, the system can only accept a single training example. Rather than maintaining the set of consistent programs, the system uses heuristics to construct a single wrapper.

Chapter 4

THE SMARTedit PBD SYSTEM

In order to evaluate our version space algebraic framework for PBD, we implemented a system called SMARTedit in the domain of text editing. It has been used for a wide range of repetitive tasks, including manipulation of highly-formatted columnar data, transformation of semi-structured XML and bibliographic data, editing programming language code, and writing unstructured text.

First we describe the user interface presented by the SMARTedit system, then describe implementation details in terms of the version space algebra. A complete summary of all of the version spaces used in SMARTedit is given in Appendix A.

4.1 *SMARTedit user interface*

We illustrate SMARTedit by showing how it automates a simple text processing task. Suppose the user has some HTML with comments embedded in it and would like to remove the comment tags and all the text inside the comments. An HTML comment is a string delineated by the tokens `<!--` and `-->`. The comment may span multiple lines and contain arbitrary characters. In order to delete all such comments, a Microsoft Word user would have to enter a regular expression of the form `\<!--*--\>` into the search-and-replace dialog box. While this syntax may look straightforward to programmers, even an experienced programmer made several attempts before discovering correct syntax for that regular expression. In SMARTedit, however, no arcane syntax is required. The user simply demonstrates the desired functionality by deleting the first HTML comment, and the system is able to do the rest. Let's walk through this example to illustrate exactly what SMARTedit can do.

SMARTedit follows the familiar macro recording interface. To begin creating a SMARTedit program, the user pushes a "Start recording" button (Figure 4.1). The button then turns red, indicating that her actions are being recorded. She then begins demonstrating

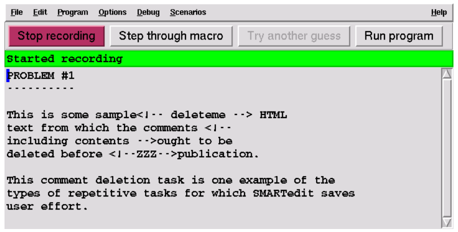

Figure 4.1: Starting to record a SMARTedit program

what she wants SMARTedit to do. In this task, she first moves the cursor to the beginning of the next HTML comment, right before the `<!--` characters, using any combination of cursor-motion keys or mouse clicks (Figure 4.2). She then deletes the entire HTML comment (Figure 4.3).

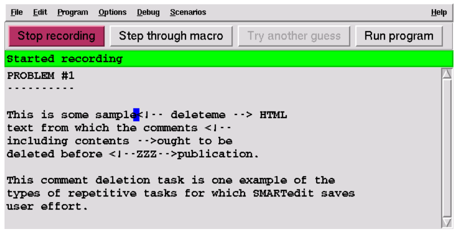

Figure 4.2: Moving to the first HTML comment

Because the demonstration is now complete, the user presses the stop-recording button to indicate that she is finished. SMARTedit has been recording the sequence of states that result from the user's editing commands, and learns functions that map from one state to the next. Based on the state sequence just observed, SMARTedit has updated its version space to contain only programs that are consistent with the demonstration the user has just performed. The user may then ask SMARTedit to predict her next action by executing the

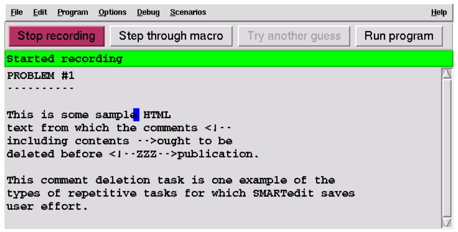

Figure 4.3: Deleting the first HTML comment

version space.

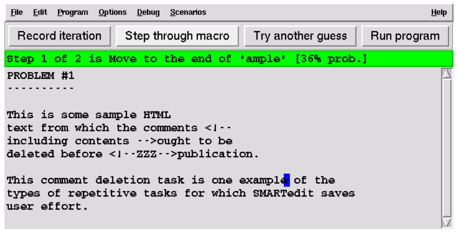

Figure 4.4: Pressing the Step button to see SMARTedit's first guess

SMARTedit learns procedures consisting of a sequence of actions—relatively high-level text-editing commands, such as moving the cursor to a new position, inserting a string, selecting or deleting a region of text, or manipulating the clipboard. The user invokes SMARTedit's learned macro an action at a time by pressing the "Step through macro" button (Figure 4.4). SMARTedit guesses what action she's likely to take next. In this case, SMARTedit makes an incorrect guess: that she wants to move to the end of the string `ample`, with 36% probability. The reason behind this guess is that the first HTML comment happened to occur after the word `sample`; SMARTedit tries to determine what aspect of the text is relevant to the user's action.

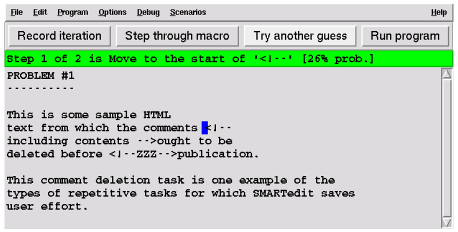

Figure 4.5: Pressing the Try button to see an alternate guess

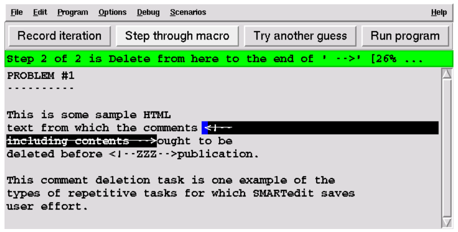

Figure 4.6: Pressing the Try button to see an alternate guess

The user indicates that this action is incorrect by pressing the "Try another guess" button (Figure 4.5), which causes SMARTedit to switch to its next most likely choice of action: correctly moving to the next HTML comment, with 26% probability. If the correct action is not within SMARTedit's first few guesses, the user is free to demonstrate the correct action instead of cycling through the guesses. The user indicates that this action is correct by stepping to the next action in the program, which is to correctly delete the extent of the HTML comment (Figure 4.6).

The user has now finished deleting the second HTML comment in the file. The next time she invokes the "Step" button, SMARTedit positions the cursor at the beginning of the next HTML comment, with 93% probability. SMARTedit has been learning from the interactions, and by this point has received two training examples: the user's demonstration

on the first HTML comment, and the feedback as the user corrected its predictions on the second HTML comment. As the user provides more examples, SMARTedit's confidence in its predictions increases, because its version space has been updated to contain fewer inaccurate programs. Once the user is confident that SMARTedit has learned the correct program, she can invoke the "Run program" button to cause SMARTedit to run its program on the remainder of the file without any further user input.

In addition to straight-line code, SMARTedit allows a user to record programs with loops. During recording, the user presses a button to indicate that she is about to begin recording the actions of a loop, and presses another button at the end of the loop; no action is necessary between iterations of the loop.

Each of SMARTedit's predictions is drawn from the hypothesis space constructed using our version space algebra. The rest of this chapter explains how SMARTedit has been constructed using the version space algebra framework.

4.2 Translation

We implemented the SMARTedit system by instrumenting the Tk [66] text widget, which provides the functionality of a simple editor. The implementation has two main parts. The **translator** converts from low-level actions such as keypresses, mouse clicks, and mouse movement to the high-level actions in the SMARTedit programming language. The **trace generalizer** implements the version spaces used to generalize from input/output states to a robust program.

A general problem that arises when constructing a programming by demonstration system is converting from interface events to high-level events in the programming language, a process that we call **translation**. For example, the Tk text widget upon which SMARTedit is based provides the ability to attach callbacks to low-level actions such as keypresses, mouse clicks, and mouse motion. While a PBD system could be built on top of such low-level actions (in fact, most macro recorders work at this level), such systems produce programs that are more brittle and difficult to construct. In contrast, actions at a higher level of abstraction may be closer to a user's mental model of the task, as well as allowing the system to generalize from multiple examples.

In SMARTedit, translation is accomplished through the use of callbacks on every possible low-level action (keypresses, mouse clicks, and mouse movement). Multiple low-level actions of the same type are grouped together to form a single high-level action. For example, pressing the cursor-left key moves the cursor one character to the left; a sequence of such low-level actions is grouped together into a single high-level "Move Cursor" action. This approach assumes that each low-level action maps to at most one high-level action. The translation process determines the pair of application states before and after each high-level action (for example, the state before movement begins and the state after the cursor has been moved to its final position).

Let s be a snapshot of the application state. A demonstration consists of a sequence of low-level actions $a_0, a_1, a_2, ...a_N$, performed in the initial state s_0. An action a_i performed in a state s_i generates a new state s_{i+1}; the action sequence applied to the initial state thus generates a state sequence $s_0, s_1, ..., s_{N+1}$ such that s_{i+1} is the result of executing action a_i in state s_i.

Let the *class* $C(a_i)$ of a low-level action be the corresponding high-level action entailed by this action. For example, $C(keypress(Left)) = Move$. Let a maximal translation sequence a_{ij} be a subsequence $a_i, a_{i+1}, ..., a_j$ of the action sequence such that for all $i \leq k \leq j$, $a_k = C_{ij}$ for some class C_{ij}. Moreover, a_{ij} is maximal in that $C(a_{i-1}) \neq C_{ij}$ and $C(a_{j+1}) \neq C_{ij}$. The action sequence therefore may be partitioned into a sequence of maximal translation sequences, $t_0, t_1, ..., t_n$, such that t_i is the index of the first action of the t_ith maximal sequence.

The translation process takes as input a sequence of actions and an initial state, and produces a sequence of states $S_0, S_1, S_2, ..., S_M$, $M <= N$, such that $S_0 = s_0$ and $S_i = s_{t_i}$. The state changes in this high-level representation correspond to high-level actions in the language.

The translation process in SMARTedit is highly dependent on the language definition, and makes strong use of the assumption that there is a unique mapping from low-level action to high-level action. The SMARTedit translation module therefore must know the semantics of each possible low-level action (*e.g.*, knowing that the pressing of the letter "a"

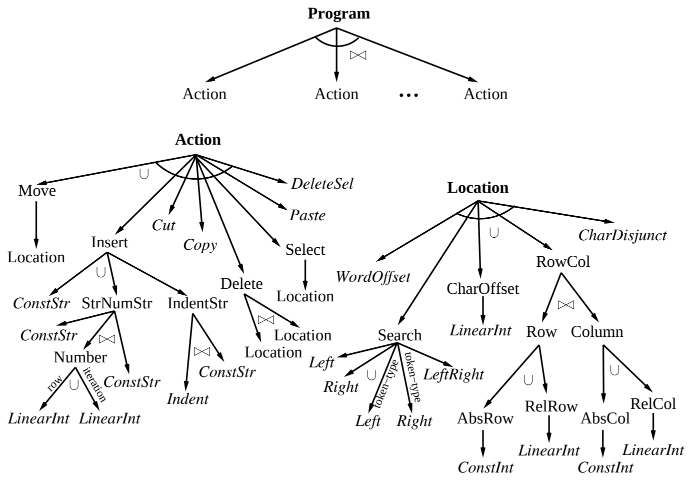

Figure 4.7: Version space for SMARTedit programs.

is part of an Insert action). In future work, it may be possible to automatically learn the required knowledge, perhaps through experimentation. This type of automatic knowledge capture is similar to work on learning planning operators through experimentation [87, 86] as well as work on recognizing end-user transactions from sequences in RPC logs [27].

4.3 SMARTedit version spaces

The translated state sequence represents a series of actions in the high-level language defined by the application designer. The learning problem is to find the target program that is consistent with one or more observed state sequences; to solve this problem SMARTedit applies the version space algebra framework defined in Chapter 3.

The complete version space used in SMARTedit is shown in Figure 4.7. (A full listing of the version spaces used in SMARTedit, along with a brief description of each, can be found in Appendix A.) The target space Program represents the version space of all text-editing programs learnable in our domain. A Program is a join of a fixed number of Action version spaces; the number of actions is determined lazily using the contents of the first demonstrated example. Each Action function represents a simple command a user might perform in a text editor, such as moving the insertion cursor, inserting and deleting text at the current cursor location, and manipulating the clipboard (selecting text and copying it to and from the clipboard).

The Insert version space represents a collection of functions that insert various forms of text into the buffer. SMARTedit recognizes several different types of insertion actions. The simplest one is insertion of a constant string. However, SMARTedit also recognizes strings that vary regularly. For instance, the StrNumStr space recognizes strings that consist of a constant string followed by a number followed by another constant string. The number can be a function of either the row number or the iteration number. For example, to number a list of citations in increasing order, one might want to insert the string [, followed by the iteration number, followed by the string].

Many text-editing actions deal with locations in the text file. For example, functions in the Move version space can be expressed as a transformation of functions that output locations in the text; the transformation outputs a state in which the cursor has been moved to the new location. Functions in the Delete version space have the effect of deleting text within a certain range, bounded by two locations in the text.

The Location version space encapsulates a useful collection of ways to specify a location in the buffer. It is a union of a number of different version spaces, each of which contains functions specifying a different type of location definition. The Location space transforms the input and output state into the format appropriate for each child version space. For example, the RowCol version space is a collection of functions that predict the row and column position of the cursor, such as "row 3 and column 26" or "the next row and the first column". The row and column position of the cursor is extracted (*i.e.*, transformed) from the complete editor state given to the Location version space for input to the RowCol version space.

4.4 String searching

A central concept in text-editing is that of positions relative to a particular string or pattern. To support this type of location, we have defined a number of version spaces based on the concept of string matching. First we define some terminology. Let string $X = x_1x_2x_3 \ldots x_i \bullet x_{i+1} \ldots x_{n-1}x_n$ denote a sequence of tokens drawn from some alphabet Σ. The symbol $\bullet$ denotes a position p in the string; positions are assumed to fall between two tokens. We say that X.left(p) is the string $x_1x_2x_3 \ldots x_i$ and X.right(p) is the string $x_{i+1} \ldots x_{n-1}x_n$.

4.4.1 Left and right searches

First we define a class of *right-search* hypotheses that output the next position such that a particular string is to its right. For each sequence S of tokens, define a hypothesis h_S^{right} that, given an input state (T, P, C, E) in which the cursor is at some position P in the text T, outputs the first position $Q > P$ such that S is a prefix of T.right(Q). The class of *left-search* hypotheses is defined similarly.

For example, suppose the user moves the cursor to the beginning of the next occurrence of the string "PBD". Three hypotheses that could be consistent with this action are h_{PBD}^{right}, h_{PB}^{right}, and h_P^{right}. Given this target position, the user may have been searching for the string "PBD", the string "PB", or the string "P". If upon moving the cursor to this new position, the user has skipped over the letter "P", then h_P^{right} is not consistent.

We can efficiently represent the right-search version space by employing a partial order on the hypotheses in the space, decreasing the space requirement from $O(|T|^2)$ to $O(|T|)$. The partial order $\prec^{right}$ is the string prefix relationship. Formally, $h_{s_1}^{right} \prec^{right} h_{s_2}^{right}$ iff s_1 is a proper prefix of s_2. For clarity, we omit the hypothesis notation and simply refer to each hypothesis as the string on which it depends.

The right-search version space can be efficiently updated as follows. The least upper bound (LUB) and greatest lower bound (GLB) boundaries of the right-search version space are initialized to be $\mathcal{S}$ and $\mathcal{C}$ respectively, where $\mathcal{S}$ is a token representing the set of all strings of length K (some constant greater than the maximum text buffer size) and $\mathcal{C}$ is a token representing the set of all strings of unit length. Let $\langle (T, P, C, E), (T, Q, C, E) \rangle$ be the input and output states in a training example for the version space. In other words, the cursor began at position P and ended at position Q, relative to text T. T.right(Q) is the longest possible string for which the user could have been searching. In moving from position P to position Q, however, the cursor may have skipped over a prefix of T.right(Q), indicating that such a prefix is not the target hypothesis. Let S_N be the longest prefix of T.right(P) that begins in the range $[P, Q)$.

To update the version space with this example, the LUB becomes the singleton set containing the longest common prefix of the string in the LUB and T.right(Q). If the LUB is $\mathcal{S}$, then it becomes the singleton set containing T.right(Q). The GLB becomes the singleton set containing the longer of the string already in the GLB, or S_N. If the GLB is $\mathcal{C}$, then the GLB becomes S_N. If at any point the string in the GLB is equal to or prefixed by the string in the LUB, then the version space collapses to the null set.

For example, suppose a user searches for the word `spaceship`, but in moving to the next occurrence of `spaceship` skips over the word `speak`. If this were the first example, the LUB would become the singleton set containing the string `spaceship` plus the rest of the text up to the end of the file, and the GLB would become the singleton set containing the string `spa`. The version space contains all prefixes of the string in the LUB, except the hypotheses `s` and `sp`.

The left-search version space is defined similarly to the right-search space. The partial order is the suffix relation, and each hypothesis h_s^{left} is a function that, when given an input (T, P, C, E), generates the position Q such that s is a suffix of T.left(Q).

Next we describe a procedure for efficiently executing the right-search version space on a given input state (a similar execution procedure exists for the left-search version space). This version space is equivalent to a set of strings, the longest of which is the string in the LUB, the others being some prefix of the string in the LUB. Conceptually, the execution process applies each hypothesis in the space to the input state and returns the set of outputs thus generated, along with the probability associated with each one. In practice, however, we compute the outputs without having to materialize the contents of the version space and execute each hypothesis separately, by exploiting the relationship between the hypotheses in the version space.

Each hypothesis represents a string; executing a single hypothesis requires searching for the next occurrence of that string relative to position P. The execution of the version space produces a set of (location, probability) tuples. For each hypothesis, find the next occurrence of the associated string in the text, and output that location along with the probability of the hypothesis. If multiple hypotheses predict the same location, the probability of that location is the sum of the probabilities of the hypotheses that predict that location.

We can execute the entire version space efficiently in time $O(|S| \times |T|)$ where $|S|$ is the length of the longest hypothesis in the version space and $|T|$ is the length of the text file[1], by comparing the string against the text starting at every position in the text file following the cursor position P. Let p_s denote the probability of the hypothesis containing the string s. The algorithm is as follows: find the first location where the first one or more characters of the text and hypothesis match. Denote the length of the common prefix as k_1 and the matching string itself as $s[1, k_1]$. Output this location, with associated probability

$$p = \sum_{i=1}^{k_1} p_{s[1,i]} \tag{4.1}$$

Continue searching from this occurrence for a match of at least length $k_2 > k_1$. Output the next matching location with probability

$$p = \sum_{i=k_1+1}^{k_2} p_{s[1,i]} \tag{4.2}$$

Repeat until the longest hypothesis has been matched or the end of text is reached, and return the set of output locations and their probabilities as the result of the execution.

To see why, consider two hypotheses h_1 and h_2 such that h_1 is a proper prefix of h_2. Let $f(h)$ be the position of the first occurrence of h after the cursor position. Then it must be the case that $f(h_1) \leq f(h_2)$. The first occurrence of h_1 cannot come after the first occurrence of h_2, since h_1 is a substring of h_2. Therefore, if $f(h_2) \neq f(h_1)$, then the search for $f(h_2)$ may begin at the location $f(h_1) + 1$, rather than starting anew from the cursor position.

[1]We suspect that we could achieve an $O(|S| + |T|)$ bound through a modification of the Knuth-Morris-Pratt algorithm.

4.4.2 Tokenized string searching

The previous section described version spaces for cursor positioning based on exact string matches. In addition, SMARTedit supports a small subset of regular-expression-style string searching: search for a sequence of token types, for a predefined set of token types. Examples of this type of string search hypothesis include searching for strings of five digits, strings of two uppercase characters, and so on.

We define the *token-type* transform as a function that transforms from an input state (T, P, C, E) to another state (T', P, C, E). If $T = x_1, x_2, ...x_n$ and type(x_i) is the *type* of token x_i, then T' =type(x_1), type(x_2), ..., type(x_n). SMARTedit defines a number of token types: the sets of lowercase letters `[a-z]`, uppercase letters `[A-Z]`, punctuation, digits `[0-9]`, and whitespace. Each token must have at most one type. For example, the string `Seattle, WA, 98195` is transformed into the tokenized string $\Omega\omega\omega\omega\omega\omega\omega\Pi$ $\Omega\Omega\Pi$ $\Delta\Delta\Delta\Delta\Delta$, where Ω represents the uppercase token type, ω represents the lowercase token type, Π represents the punctuation token type, and Δ represents the digit token type. In the current version of SMARTedit, these token types are built in to the system and may not be customized by the user. (Section 4.5.1 describes character class disjunction, which allows a user to define a custom token type by example, but limits the search to the next occurrence of a single token of that type rather than a sequence.)

SMARTedit uses this token-type transform to construct a pair of version spaces that perform searches over sequences of token types, analagously to the left and right searches over sequences of tokens. Each token-type version space performs an exact search over the transformed text. For instance, if the cursor position moves to `Seattle, WA, •98195`, then the example $\Omega\omega\omega\omega\omega\omega\omega\Pi$ $\Omega\Omega\Pi$ $\bullet\Delta\Delta\Delta\Delta\Delta$ is used to update the token-type version spaces. The hypothesis $\Delta\Delta\Delta\Delta\Delta$ is consistent with this example, *i.e.*, that the user is searching for a sequence of five digits. The tokenized version spaces are shown in Figure 4.7 as components of the Search space.

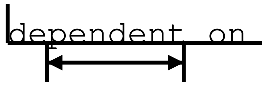

Figure 4.8: Version space for right string search example

4.4.3 Generalizing string searching

We may represent a string search version space as two offsets in a sequence of tokens. Once at least one training example for the right-search version space has been observed, for example, the space contains only the observed right-string plus all its prefixes. As more positive examples are observed, the upper boundary of the version space contains the singleton set containing the longest common prefix of all the positive examples. As more negative examples are observed, shorter hypotheses are removed from the space, thus increasing the lower boundary of the version space (the shortest prefix that does not cover any negative examples).

For example, Figure 4.8 shows a right-search version space trained with the positive examples `dependent on` and `dependently` and the negative example `decline`. The longest common prefix in the positive examples is the string `dependent` while the shortest prefix that doesn't cover any negative examples is the string `dep`. The version space contains all prefixes of the original string (`dependent on`) that are longer than 2 characters and shorter than 10 characters, but because of the partial order we can store this version space efficiently as the original string plus two offsets into that string.

The partial order used in the string search version spaces is not a generality partial order. Consider two string search hypotheses h_1 and h_2 such that $h_1 \prec h_2$. For the ordering to be a generality ordering, the set of examples covered by hypothesis h_1 must be a proper subset of the examples covered by hypothesis h_2. If each hypothesis returned the set of all positions where the text matches the search string, then this would be a generality ordering due to a subset relation. However, in this version space, each hypothesis returns only a single position (the first location where the text matches the hypothesis); there is not necessarily

A display was rendered for re+play. We re+played it.

Figure 4.9: Example of conjunctive left and right search hypotheses.

any relationship between the outputs of the two hypotheses h_1 and h_2.

Another way to view string searching is as a loop. While the text at the cursor position does not match the hypothesis, move the cursor forward by one character. In this light, string search hypotheses become concept learning hypotheses subject to a generality ordering. A hypothesis classifies positions as true when the surrounding text matches the search string, and false otherwise. The set of positions covered by one hypothesis may be a subset of the set of positions covered by a second hypothesis, in which case the two hypotheses are ordered by a generality relation. We use this idea to improve the efficiency of a class of hypotheses for conjunctive left/right string searches, as discussed in the next section.

4.4.4 Conjunctive string searching

Conjunctions are one example of a dependent join—a join in which the consistency of one hypothesis in the cross product depends on the consistency of the other. Hypotheses in a dependent join cannot be factored into their component parts and the consistency maintained independently, as they are for an independent join.

The independent join allows efficient representation of the space of hypotheses in the joint space. Not all joins have the independence property, however. In the text-editing domain, one example of a dependent join is string conjunction for left and right string search. Examples of hypotheses in this join include `re•play` (the position after the string `re` and before the string `play`) and `WA, •98195` (the position after the string `WA,` and before the string `98195`). Consider the sentence in Figure 4.9. Suppose the target hypothesis is `re•play` (*i.e.*, the position after the string `re` and before the string `play`). Plus signs in the figure indicate positive examples of the target concept; all other positions are negative examples. One might think that conjunctive string hypotheses can be represented with two independent version spaces, one for left (*e.g.*, `re`) and one for right (*e.g.*, `play`). However, the

conjunction of the two is not an independent join. Consider an independently maintained left-search version space, given a cursor movement from the beginning of the text to the first occurrence of `replay`. The shortest consistent hypothesis in this space is the string `r␣re`; the hypothesis `re` is made inconsistent by the fact that the cursor skipped over the string `rendered` in moving to `replay`. Similarly, the right-search version space does not contain the hypothesis `play`, because that string is skipped over in the word `display`. In fact, the correct hypothesis is the combination of the string `re` with the string `play`.

Figure 4.10 shows a visualization of the space of conjunctive left/right search hypotheses. The y-axis indicates possible left-search strings, while the x-axis indicates possible right-search strings. Each point in this coordinate space represents a single conjunctive hypothesis. For example, the lower-left point corresponds to the hypothesis `e•p`; the upper-right point corresponds to the hypothesis `r␣re•play`. Note that an independent join of left and right search hypotheses would only allow rectangle-shaped shaped version spaces. The need to represent more complex regions, such as the region shown in the figure, indicates that the join must be dependent.

The dark squares in the figure indicate hypotheses consistent with the training data; white squares are inconsistent. Negative training examples cause the lower left boundary of the consistent set (the *minimal* boundary) to grow up and to the right, while positive training examples cause the region to shrink in from the upper right by updating the *maximal* boundary. In this example, a negative training example `e•pl` (such as in the word `deplane`) has been incorporated into the version space. This negative example invalidates the two hypotheses `e•p` and `e•pl`. However, other hypotheses such as `re•play` and `re•pl` are still consistent.

In general, representation of a conjunctive version space (*i.e.*, a dependent join) requires $O(|H_1| \times |H_2|)$ space, using one bit to indicate whether each conjunction of hypotheses is consistent with the data. However, we exploit the partial orders of the component spaces in the conjunction for a more efficient representation.

We efficiently represent the string conjunction version space using space linear in the length of the left or right search string. This efficient representation takes advantage of the fact that maximal and minimal boundaries are always rectangular. The representation is as

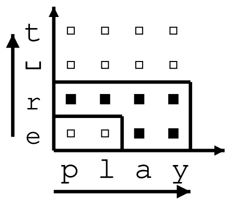

Figure 4.10: Conjunctive left/right search version space.

follows: for each vertical slice (*i.e.*, each possible value of the x-coordinate), represent the upper and lower bounds of this slice as the indices of the upper and lower boundaries respectively. This representation requires storage space for two integers for each x-coordinate, hence the linear upper bound on the space required. The set of indices for the upper boundary may be viewed as the upper boundary of the version space, and similarly for the lower boundary.

This version space takes advantage of a two-dimensional partial order constructed out of the partial orders on the component dimensions to efficiently represent the version space by its boundary sets. Generalizing the conjunctive left/right approach efficiently to further dependent joins is an area for future work.

4.5 Disjunction

Another class of useful hypotheses is disjunctive hypotheses, such as "move to the next occurrence of either `<UL>` or `<OL>`". Such hypotheses can be very difficult to learn: at one extreme, the single hypothesis representing the disjunction of all observed examples is always consistent. In this section we examine techniques for supporting disjunctive hypotheses within the context of string searching in SMARTedit.

4.5.1 Character class disjunction

We begin by discussing a simple type of disjunctive hypothesis: searching for the next occurrence of any single token drawn from a set of allowed tokens. Such sets include punctuation symbols, digits, or vowels. For instance, a vowel-set hypothesis may take the form "the location before the next occurrence of any of the letters `a`, `e`, `i`, `o`, or `u`". Unlike the token-type hypotheses described in section 4.4.2, here we allow user-defined sets of tokens.

Representing hypotheses for any fixed number of predefined sets, such as the set of vowels, is straightforward. In the general case, however, we wish to allow users to define their own character classes. Conceptually, the hypothesis space contains the power set of the set of all tokens in the alphabet. For example, for tokens within the ASCII character set, we define a hypothesis space containing the sets {}, {a}, {b}, {a, b}, ..., $\top$. $\top$ represents the set of all tokens in the alphabet.

Each target location results in one positive example and possibly a number of negative examples. The positive example is the token at the target location, and the negative examples are all the tokens that were skipped over to reach the target location. For instance, suppose the user moves to the token `a` but skips over the tokens `b` and `c`. Then the version space contains all character-class hypotheses that contain the token `a` but do not contain `b` and `c`.

We can represent this large version space more compactly by employing a partial order and representing boundary sets. We define a partial order $\prec$ such that $h_1 \prec h_2$ if either $h_2 = h_1 \cup \{c\}$ for some $c \notin h_1$, or $h_1 \prec h_3 \prec h_2$ for some h_3. Figure 4.11 illustrates such a version space for alphabetic tokens. This partial order is not a generality ordering—each function (corresponding to a set) outputs the next location where one of the tokens in the set appears. However, the version space may still be efficiently represented by the boundaries—the least upper bound (LUB) and greatest lower bound (GLB)—of the consistent set. Each positive example raises the GLB to remove hypotheses that do not contain this token. Each negative example lowers the LUB to exclude hypotheses containing this token.

An even more efficient representation is gained by representing the version space with a vector, containing one tri-state bit for each token in the alphabet, representing whether

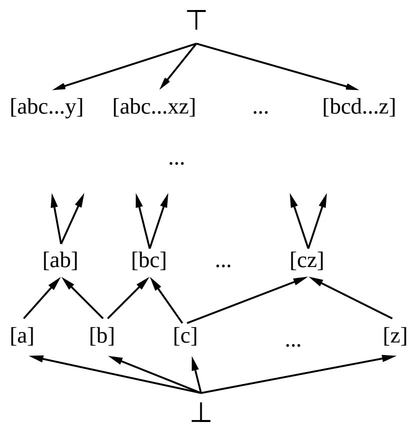

Figure 4.11: Version space with partial order for character class hypotheses

it has been observed as a positive example (+), negative example (-), or unseen (?). Given the bit vector, the contents of the version space may be computed dynamically. It contains all sets of single tokens that are either positive or unseen, all sets of two positive or unseen tokens, and so on, up to the set of all unseen or positive tokens. All hypotheses in the version space contain all positive tokens. No hypotheses in the version space contain any negative tokens. Once all tokens have been seen (as either positive or negative examples), the version space will contain at most one hypothesis.

Theoretically, the version space with bit vector representation may be executed by computing the set of hypotheses in the version space, executing each hypothesis on the input state, and collecting the resulting output states as usual. However, we have developed an algorithm for computing the output states without computing the extension of the version space, by employing only the information contained in the bit vector.

The algorithm works by calculating the probability that the target location occurs at

each successive position in the text, starting from the input location. The algorithm will successively distribute a total probability mass of 1 to successive locations in the text. Consider the first token following the input location. If this token is negative, then the version space cannot contain any hypotheses containing this token; the algorithm resumes on the next token. If this token is positive, then all hypotheses in the version space must contain this token; the algorithm assigns all of the remaining probability mass to this location. If the token is unseen, then exactly half of the hypotheses in the version space contain this token (by symmetry); the algorithm assigns P of the remaining probability mass to this location, temporarily marks this token as negative, and resumes at the next token.

Setting P to 0.5 is equivalent to setting a uniform probability distribution over hypotheses in the version space. To see this, observe that by symmetry, exactly half of the hypotheses in the space contain any previously unseen token; thus half of the hypotheses will predict stopping at the first occurrence of that token. Setting P to a smaller value encodes a bias against longer hypotheses (*e.g.*, hypotheses containing larger sets of tokens).

For example, consider an alphabet containing only the tokens `a`, `b`, and `c`. Suppose the text consists of the string `abcbac`, and that the target hypothesis is {`b`, `c`} (*i.e.*, the position before the next occurrence of either `b` or `c`). For the first example, the user moves from `•abcbac` to `a•bcbac`. No set containing `a` is consistent with this example. Moreover, `b` must occur in the target hypothesis. Thus the version space contains only the hypotheses {`b`} and {`b`, `c`}. To execute the version space in the input state `a•bcbac`, the algorithm considers each subsequent position in turn, starting with a probability mass of 1. The first token following the input location is the token `c`, which is unseen. Thus the algorithm assigns P probability to this location (note that half of the consistent hypotheses predict stopping at token `c`). The next token is `b`, which is positive; the algorithm assigns 1-P probability to this location, and terminates, since all hypotheses in the version space must contain the token `b`.

```
<HTML><BODY>
<UL>
      <LI> Apple
</UL>
<OL>
      <LI> Banana
</OL>
<UL>
      <LI> Cherry
</UL>
</BODY></HTML>
```

Figure 4.12: HTML document for string disjunction example

4.5.2 String disjunction

The techniques discussed in the context of single-token disjunction may be extended to handling disjunctions of strings. Although we have previously considered a token to be a character in the text file, we now consider each substring in the text as a single token. If the text is of length n, then the number of substrings is $O(n^2)$. Each string token may be represented by two indices—the starting and ending position of the string—each ranging from 1 to n.

With this model, we define string disjunction hypotheses similarly to character disjunction hypotheses by considering the hypothesis space to be the power set of tokens in the text. The same techniques used to represent the version space for the character disjunction case, such as the bit vector representation, may be applied for string disjunction as well.

The primary difference between character and string disjunctions is that each target location generates a source of negative examples (the tokens that have been skipped over) but no confirmed positive example. The candidate positives (in the right-search case) are all tokens that begin at the target location; but which of these tokens is the correct hypothesis is unknown.

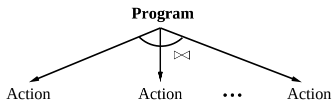

Figure 4.13: Version space of programs as a sequence of actions

For example, suppose the target hypothesis is to search for the next occurrence of either `<UL>` or `<OL>` within an HTML document shown in Figure 4.12, starting with the cursor at the top of the file. The first example is the position before the first `<UL>`. The candidate tokens for a right-search space include `<U`, `<UL`, `<UL>`, and so on, up to the token consisting of the text between `<UL>` and the end of the file. The token `<` is not consistent because it was skipped over in this example. Thus, unlike the character class disjunction case described above, we cannot mark a single token as positive, but must maintain the set of candidate positives.

Execution of a string disjunction version space is computationally complex due to the lack of confirmed positive examples. As in the character disjunction case, we compute the probability that the target location occurs at each successive location in the text. However, computing each probability requires a sum over the probabilities of each token beginning at this location, resulting in an $O(n^3)$ algorithm. More efficient execution of string disjunction version spaces remains an area for future work.

4.6 Learning unsegmented programs

The SMARTedit system described thus far is capable of learning a fixed-length program from a segmented sequence of actions, learning each action individually from examples of the state before and after the action. These actions may be viewed as part of a single Program version space (see Figure 4.13). A Program consists of a join of a fixed number of Action version spaces. The transformation function for the top-level Program space takes a sequence of states $s_0, s_1, ..., s_n$ and constructs n examples of the form $\langle s_{i-1}, s_i \rangle$, such that

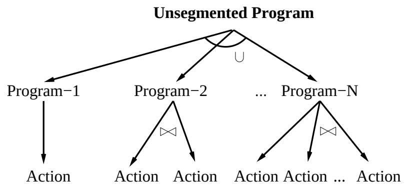

Figure 4.14: Version space of unsegmented programs up to length N

the i^{th} tuple is used to update the i^{th} Action version space.

The Program version space requires knowledge about the segmentation between iterations in order to decide how to assign each state example to the proper version space. SMARTedit collects segmentation information from the user by virtue of the start/stop recording button clicks; the user must explicitly stop recording between each iteration of the repetitive task. If a user wants to record multiple iterations at once (perhaps for a more complex task that is known to require several demonstrations to learn correctly), she must stop and restart the recorder between each iteration. In this section we extend SMARTedit to learn from *unsegmented* traces, that is, traces which the user does not manually segment into iterations.

The insight behind unsegmented programs is that when we do not know the correct segmentation (i.e. how many actions make up an iteration), we maintain a version space containing all possible segmentations. The UnsegmentedProgram version space is shown in Figure 4.14. This version space contains a union of Program version spaces, each of which is composed of a different number of actions, from 1 to N. We lazily construct this version space such that N is the number of actions in the trace, making the assumption that the user will demonstrate at least one complete iteration, if not more.

The UnsegmentedProgram version space is updated with a trace as follows. Given a sequence of N actions, and an iteration length M such that $M \leq N$, the UnsegmentedProgram maintains a union of N Program version spaces, such that $\mathsf{Program}_1$ consists of a single

Action, $\mathsf{Program}_2$ consists of two Actions, and $\mathsf{Program}_N$ consists of N Actions. $\mathsf{Program}_1$'s single Action version space is updated with all N observed actions as if each action were an example of a unit-length program. $\mathsf{Program}_2$ is updated with $N/2$ pairs of actions, and so on. $\mathsf{Program}_N$ is updated with the entire trace as if the entire trace were an example of a single length-N iteration. Many of these Program version spaces collapse fairly quickly because the action types do not match up except in programs of length M and multiples of M.

Execution of the UnsegmentedProgram version space produces the union of the execution of each of the nested Program version spaces. In SMARTedit, what this means is that the user is able to cycle through different hypotheses of the form "Step 2 of 3 is to Select up to ..." and "Step 1 of 4 is to Move to...". Each hypothesis may correspond to a different $\mathsf{Program}_i$ version space and therefore be part of a program with a different iteration length. To reflect our belief that shorter programs are more likely than longer programs, we weight the $\mathsf{Program}_i$ members in the union such that the prior probability of $\mathsf{Program}_i$ is $1/2^i$.

This approach can be extended to learn programs from traces where the starting point in the trace is not identified; the repetitive program (with an unknown iteration length) occurs at an unknown position in the trace. Rather than burdening the user with having to explicitly begin the recording, why not always record, and learn programs by passively watching the user's actions?

We call this construct a startless program and learn it as follows. Just as we defined the UnsegmentedProgram as a union of Program version spaces for all candidate iteration lengths, a StartlessProgram version space is a union of UnsegmentedProgram version spaces. This version space learns from an ongoing trace. As each action is observed, a new UnsegmentedProgram is created such that its program begins at this action. The same action is also used to update all the other UnsegmentedProgram version spaces as if it were the next action in the sequence for each such version space.

Each of these techniques increases the number of hypotheses in the version space, and thus more examples are required to learn the correct program. In practice, unsegmented program can be learned reasonably quickly with few training examples; the variety of different action types cause most of the Program version spaces to collapse quickly. However,

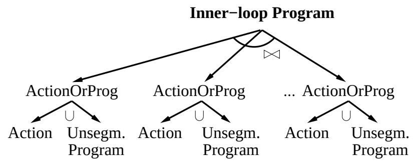

Figure 4.15: Version space of programs containing nested loops

the StartlessProgram version space is less useful in practice, primarily due to the fact that it never stops growing; each new action may always be the start of a new program.

4.7 Learning nested loops

SMARTedit as described thus far is capable of learning a single straight-line program that consists of a sequence of text-editing actions. For many text-editing tasks, such as reformatting bibliographic citations, variances in each record require a program with conditionals or loops. For instance, a single bibliographic entry may require conditional formatting depending on whether it is a book or a journal article. Different bibliographic entries have different numbers of fields and varying numbers of authors, requiring a program capable of looping over each item individually.

In this section we show how to extend SMARTedit's version spaces to learn programs containing loops. The idea is that each component of the top-level Program join can be either an Action or itself a Program (see Figure 4.15). Each observed user action is either part of a single primitive action, or part of a nested program.

To keep the search space manageable, SMARTedit requires users to manually indicate when they enter and exit nested programs by clicking the start-nested-loop and end-nested-loop buttons during the demonstration. Given this segmentation, a state sequence may be correctly partitioned into sequences of top-level actions as well as actions in an nested loop. (We currently limit the nesting depth to one, because of limitations in the user interface, but the approach scales to an arbitrary depth.) The number of actions in each program is

determined lazily according to the length of the first training example.

For nested loops, we felt that the overhead for the user of indicating the boundary between iterations, in addition to indicating the nested loop segmentation, was an unnecessary burden. Moreover, nested loops tend to contain few actions. Hence, the version space used for nested loops is an UnsegmentedProgram, which learns from unsegmented sequences of actions.

While the segmentation given by the user is sufficient to construct the nested-loop version space and learn a program, the question is raised of how to execute the resulting learned program on behalf of the user. Suppose the user presses the next-step button to execute the next action, after just having executed the last step in a nested program. Which step should be executed at this point, the next top-level action, or the first action in the nested program (*i.e.*, continuing to iterate through the nested loop)? For instance, suppose the target program contains a loop over the actions ABC followed by action D. After action C is executed, either A or D could plausibly follow this action.

Our loop model has several parts: the initialization condition, that sets up an iteration variable; the loop body, a sequence of primitive actions; the increment, which updates the iteration variable on each pass through the loop; and the termination condition, which is tested to determine whether to exit the loop or continue executing it.

In the context of SMARTedit, the position of the cursor may be viewed as the loop variable, initialized to the cursor position at the start of the loop. After each loop iteration, the cursor position is incremented to point to the next region of text over which to perform the actions in the loop body. For instance, while editing bibliography entries, the cursor begins at the start of the first bibliography entry. The actions in the loop body manipulate the text of this entry. On the next loop iteration, the cursor begins at the start of the second entry, and so on. The remaining question is how to define the loop termination condition.

We have considered several alternatives. The first is to have the user indicate the value of the termination condition each time the loop is executed. This is the solution implemented in SMARTedit. When stepping through the learned program, the user must choose whether to execute the next action in the nested loop, or switch to the next action in the top-level program. This choice is integrated with the try-another-guess button so that it is consistent with the existing interface. Without any knowledge of the likelihood of the loop terminating, SMARTedit places an arbitrary prior probability on possible next actions such that there is only a 20% probability of exiting the loop and continuing with the main program. It would be straightforward to extend this approach to vary this probability with the length of the iteration (perhaps with a maximum probability at the average number of iterations in past executions of the loop).

An alternative approach is to use *hypothesis inapplicability* to heuristically guess the value of the termination condition. The idea here is that some hypotheses may not be applicable in an input state (*e.g.*, moving to the next line when the cursor is already at the end of the file). We have considered various heuristics, including inapplicability of the first action in the loop body, to predict loop termination.

Using inapplicability as the loop termination condition breaks down for nested loops, however, since it is often the case that inapplicability is only true within a certain range of text. For example, consider a program with a loop over each author in a bibliography entry, such that the first action in the loop body moves to the next author's last name. That movement action will always be applicable; in the worst case, it will move to the first author of the *next* bibliography entry. But if the inapplicability were limited to the range of text containing only this citation, then the heuristic would be more useful. Essentially, this test is an example of a *conjunctive* condition—the loop terminates where there are no more author names *within the current bibliography entry.*

In summary, this chapter described the SMARTedit system for learning text-editing programs by demonstration, including the version space representation used to learn from examples. Next we describe a set of experiments designed to evaluate how well the system performs as well as its usability.

Chapter 5

EVALUATION

We evaluate the SMARTedit system using two methods. First we study SMARTedit's performance on a collection of repetitive text-editing scenarios. Then we report on the results of a small user study.

5.1 Empirical evaluation

We have applied the SMARTedit system to a representative collection of repetitive text-editing scenarios Each scenario is a collection of training examples; a training example is a sequence of (T, L, P, S) states (text buffer, cursor location, clipboard contents, and selection region). Table 5.1 shows the scenarios we used to evaluate SMARTedit, along with the total number of instances in each, and the number of instances the system needed to see before making the correct prediction on all remaining instances (*i.e.*, applying the correct transformation to them). These scenarios are described more fully in Appendix B. An asterisk (*) indicates that the task requires inner loops, and the result shown assumes that SMARTedit is given the termination condition for the loop. The citation-to-bibtex task is the same task as described in Section 1.1. All training was performed by an experienced SMARTedit user. In some cases, the system succeeds after just a single example. While many of the scenarios consist of a small number of examples (because we used the data from real-world tasks), we have examined the learned version spaces and verified that they would produce the correct behavior in general. Although we do not present timing results, the learning algorithm runs virtually in real time on a 600 MHz Pentium III.

Many of the scenarios were learnable with only two training examples. Some of the others required more training examples because of irregularities in the training data. For example, the outline scenario involves manipulation of a file in Emacs outline format. The task is to number each top-level outline element in consecutive order, and make a list of all of the top-level elements at the top of the file. After the first two examples, SMARTedit learns a program that works correctly on the third, fourth, and fifth examples. However, the sixth example is irregular: while all previous top-level elements are followed immediately by a second-level bullet, the sixth top-level outline element is followed by a few lines of extra text prior to the next bullet. As a result, SMARTedit's version space contains two hypotheses that are consistent with all previous examples, but differ on this example. Since the incorrect hypothesis has higher probability, SMARTedit performs the wrong action on this training example. According to our metric, SMARTedit thus requires six examples to learn the task correctly. However, if the examples had occurred in a different order, SMARTedit would have required only three examples to learn the task correctly. We are investigating the use of active learning to identify anomalous training examples earlier in the training process [90].

The bibitem-newblock scenario marked with a * in Table 5.1 requires the use of inner loops to solve correctly. The task in this scenario is to convert from a BibTeX intermediate representation to a human-readable format by removing the formatting commands. The task requires a program that loops over a variable number of `\newblock` commands in each bibliography entry, as well as performing some fixed work at the beginning and end of each entry. Table 5.1 shows the number of training examples required to learn to perform the bibitem-newblock task correctly assuming that the termination condition is given (*i.e.*, that the user indicates to SMARTedit when to terminate the inner loop and resume execution of the main program); the current version of SMARTedit is not able to predict when to terminate the inner loop on its own.

5.2 User evaluation

We conducted a small user study to gather user experiences with the SMARTedit system. We asked six undergraduate computer science majors to perform a set of seven tasks with and without SMARTedit's PBD capability. We gave each participant a five-minute introduction to SMARTedit's user interface, and guided them through one or two simple tasks using SMARTedit to familiarize them with the interface. We then asked them to perform

a sequence of seven tasks, first using SMARTedit's programming by demonstration feature, and again using the same editor but without invoking the programming by demonstration capability (i.e., completing the repetitive task manually). Each participant spent less than an hour total interacting with SMARTedit; none had had any prior experience with the program. After they had finished the tasks, we asked them to complete a form with questions about SMARTedit's helpfulness, usability, and whether or not they would consider using the program again.

The tasks we chose varied in difficulty and ranged from 4 to 27 iterations per task. They were chosen to reflect reasonably-sized tasks that we imagine users might naturally face. For each task, and for both SMARTedit and manual execution, we measured the time, the number of key presses, and the number of mouse clicks a user required to complete the task. The tasks were, in order: html-comments, country-codes, column-reordering, xml-comment-attribute, smartedit-results, number-citations, and bibitem-newblock.

The first six tasks were straightforward fixed-length programs, roughly increasing in difficulty; the seventh task (bibitem-newblock) required users to construct a program with an inner loop. We allowed users to give up on completing a task with SMARTedit if they became frustrated after five minutes of effort. All users completed the first six tasks with the exception of two users on the fifth task. Only one user successfully completed the seventh task using inner loops; another cleverly solved it with SMARTedit using two passes over the data.

We measured SMARTedit's performance with two metrics: the time savings gained by using SMARTedit over doing the task manually, and the percent of user actions (both keypresses and mouse clicks) saved using SMARTedit compared to manual.

Figure 5.1 shows the difference in seconds between the time taken to complete each task manually, and the time to complete it with SMARTedit. Bars above the zero line indicate that a user completed the task more quickly with SMARTedit. X's indicate missing data; in two cases, we failed to collect timing data during the experiment, and two users were unable to complete the fifth task. Users learned to use SMARTedit more effectively over time; by the fourth task, most users were able to benefit significantly from SMARTedit. Although not all users spent less time with SMARTedit than on the manual task, the results are

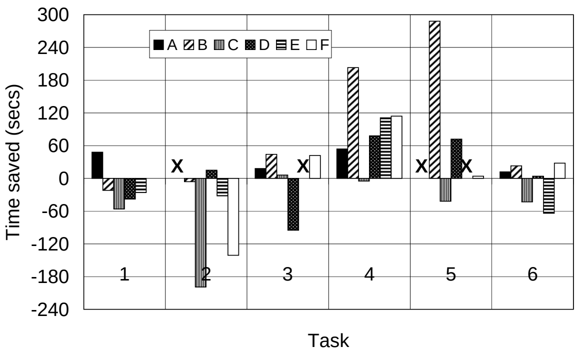

Figure 5.1: Time savings using SMARTedit compared to manual performance

encouraging given that none of the participants had any prior experience with the system. In addition, the measured time includes the time users spent questioning how the system worked, especially during the first few tasks.

Figure 5.2 shows the percentage of user actions saved when using SMARTedit as a fraction of the number of manual actions (*i.e.*, the number of manual actions minus the number of SMARTedit actions, divided by the number of manual actions). We define a user action to be either a keypress or a mouse click. Bars above the zero point show a savings with SMARTedit compared to manual execution. The higher the bar, the fewer actions a user performed with SMARTedit. X's indicate users that could not complete the task using SMARTedit. As the figure shows, starting with the third task, all users had learned enough about SMARTedit to be able to teach it the task using fewer keystrokes and mouse clicks than in the manual case.

The time saved by using SMARTedit on a task depends on the number of iterations in the task; one expects the savings to increase with the number of iterations. We averaged

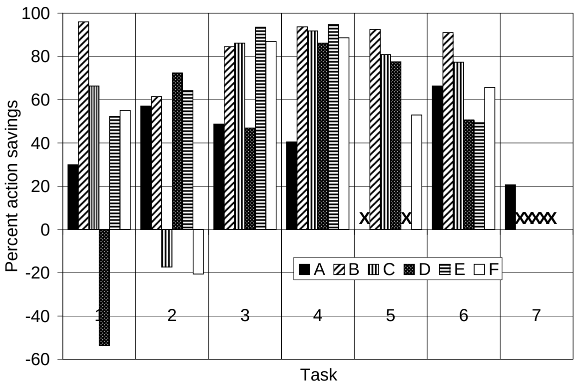

Figure 5.2: User action savings using SMARTedit compared to manual performance

the time savings across all users for each task; Figure 5.3 shows the average time savings on a task versus the number of iterations in the task. We fit a line to these points, obtaining r^2 of 0.97. The figure confirms our intuition, and indicates that for novice users, the break-even point (the number of iterations for which the total time spent operating SMARTedit equals the time spent performing the task manually) is about 15 iterations. We note that participants became more proficient in using the system over the course of the study, so these results may be skewed by a practice effect.

Since all of our user study participants were untrained in the use of SMARTedit, much of their time was spent learning how to operate the system. To provide an alternative view of SMARTedit's usefulness, two experienced SMARTedit users performed the smartedit-results task both with SMARTedit and without, recording the time spent per iteration. Figure 5.4 shows the cumulative time spent performing each iteration of the task for each user in both situations. Both users operated SMARTedit naturally: they demonstrated a single iteration, stepped through the second iteration while correcting SMARTedit's wrong guesses, and ran

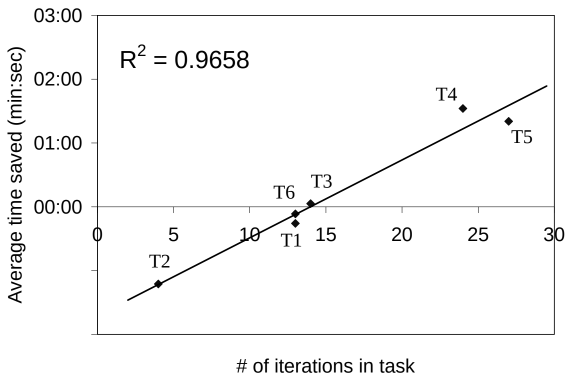

Figure 5.3: Average time saved for each task

the program uninterruptedly starting from the third iteration. The uninterrupted run steps slowly through the first few actions of the learned program, and speeds up over time. As shown in the figure, it took longer to teach SMARTedit than to perform the task manually on the first two iterations, but the total time was much shorter with SMARTedit than without. The break-even point is 5 iterations. While this result does not generalize to tasks of arbitrary complexity, it suggests that the break-even point decreases as a user gains experience with the system.

We also obtained user feedback in the form of responses to a questionnaire, summarized in Table 5.2. (User A did not return the survey.) The feedback was quite positive. We asked users to rate (on a scale of 1 to 5, 5 being the best) SMARTedit's helpfulness in accomplishing the tasks, SMARTedit's usability, and whether they would use the system again. As the table shows, participants found SMARTedit helpful and would use it again. One user wished he had had SMARTedit at work, saying "this would have been a nice thing to have." Others called it "cool" and "clever." Another noted the small number of user

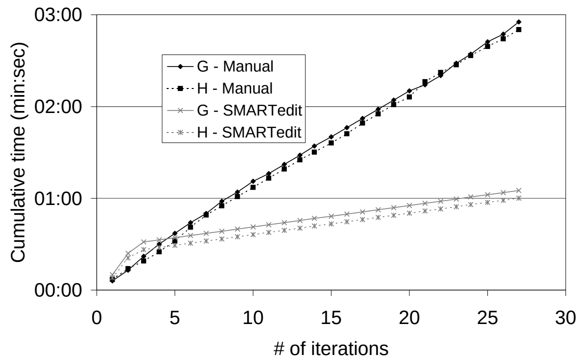

Figure 5.4: Cumulative time required for experienced users on one task

actions necessary to finish the task with SMARTedit and said, "Four keys and nine mouse clicks. I like that!"

The study indicated that SMARTedit clearly needs improved usability before it gains acceptance. Its primary failing is a lack of an "undo" facility; if a user wrongly confirms one of SMARTedit's guesses by stepping through it to the next action, SMARTedit's version space could collapse, and users have no choice but to retrain the program from scratch. Improving SMARTedit's usability is a clear direction for future work, as is performing more extensive user studies.

Although SMARTedit is technically capable of learning programs with inner loops, users found it difficult to use the feature correctly in the seventh task. The most prominent failing is the fact that SMARTedit does not yet learn loop termination conditions, so users have to manually guide SMARTedit out of the inner loop so that it can resume the actions in the main program. Users did not expect to have to guide SMARTedit after the second example, and all but one user failed to maintain the close attention required to guide SMARTedit out of the inner loop at the appropriate time. In the next chapter, we describe a framework for learning loops and conditions in the context of an abstract programming language.

Table 5.1: Scenarios used to test the SMARTedit system

Scenario	**# Train Exs.**	**Total # Exs.**
bibitem-newblock*	1	13
c++comments	1	5
column-reordering	1	14
country-codes	1	4
modify-to-rgb-calls	1	20
number-fruits	1	14
prettify-paper-info	1	10
subtype-interaction	1	3
xml-comment-attribute	1	24
addressbook	2	6
citation-creation	2	13
grades	2	7
html-comments	2	13
latex-macro-swap	2	8
number-citations	2	13
number-iterations	2	7
smartedit-results	2	27
zipselect	2	6
game-score	3	7
html-latex	3	7
indent-voyagers	3	32
mark-format	3	6
bold-xyz	4	50
citation-to-bibtex	5	10
bindings	6	11
boldface-word	6	11
ul-to-dl	6	7
OKRA	10	14
outline	10	14

Table 5.2: User feedback on the SMARTedit system

Question	**User B**	**User C**	**User D**	**User E**	**User F**
Helpful?	4	4	4	4	5
Usable?	4	4	3	2	3
Use again?	4	5	4	5	5

Chapter 6

LEARNING PROGRAMS FROM TRACES

Our goal of a cross-application framework for programming by demonstration relies on being able to learn a single abstract programming language, supporting complex control flow statements, that ties together statements in a variety of application domains. As an analogy, consider the role of Visual Basic for scripting the applications in the Microsoft Office suite. In order to script an application using Visual Basic, the application must provide a programming interface (an API) that maps from the internal functionality of the application to objects and methods callable from Visual Basic. Independently of its use for scripting Office applications, however, Visual Basic is an abstract programming language that supports conditionals and loops. Together, the APIs and the abstract language combine to make it possible to write complex programs with loops and conditionals that span multiple Office applications.

Our goal is to provide the ability to program multiple applications by demonstration, in the same way that Visual Basic allows programmers to write scripts that span multiple applications. The SMARTedit system has shown how we can learn actions in a particular application domain. In the scripting analogy, each SMARTedit action corresponds to a statement in the text-editing API.

In this chapter, we provide the second part of the puzzle: an abstract programming language that supports loops and conditionals, that can tie together statements in multiple application domains, to provide cross-application PBD. We describe a system that is capable of learning programs using a subset of the Python programming language. The SMART-python system supports conditionals, loops, and arrays. The remainder of this chapter first presents the general program-learning framework, and then describes and evaluates the SMARTpython system created using this framework.

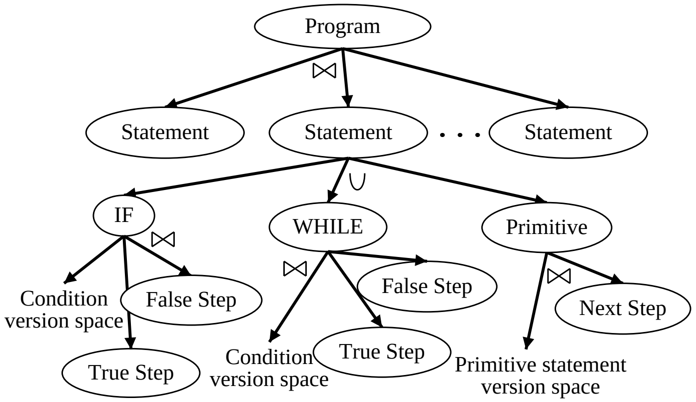

Figure 6.1: Domain-independent version space for learning complete programs

6.1 A framework for program learning

As before, we formalize a *program* as a procedure that, given an initial state S_0, produces a *trace*—a sequence of states $S_0, S_1, ..., S_n$ resulting from the execution of the procedure on S_0, such that the execution of a single program statement in state S_i produces state S_{i+1}. The program's inputs and outputs are encoded in the trace. A *training example* is a trace $\langle S_0, S_1, S_2, S_3, ..., S_n\rangle$. We say that a program is *consistent* with an example trace $T = \langle S_0, S_1, S_2, ..., S_n\rangle$ iff the program, when executed in state S_0 and given the inputs included in T, produces T as its trace. Each state S_i may contain a program step identifier that indicates the program step executed on this state in the trace.

In addition to primitive (domain-specific) actions, we extend the set of learnable program statements to contain control flow statements: the IF statement, which causes control flow to branch to one program statement or another depending on the truth value of a Boolean expression; and the WHILE statement, which repeatedly executes a sequence of statements as long as a Boolean expression holds true.

Figure 6.1 presents a version space that shows how we learn complete programs from traces. Each statement in the program is either a primitive statement (such as an assignment statement in the Python language, or a text-editing action) or a control flow statement. Previous chapters have shown how to learn sequences of primitive statements given examples of their input/output behavior; this chapter extends that approach to learning control flow statements as well.

Each control flow statement (IF or WHILE) is a join of a Condition version space with two more version spaces, each containing the program step to be executed if the condition evaluates to true or false, respectively. The Condition version space contains the set of all Boolean expressions supported for a particular domain.

The framework in Figure 6.1 is domain-independent. The Condition and Primitive statement version spaces in the figure (drawn without bubbles) may be replaced by the appropriate version spaces for a particular domain in order to learn programs with loops and conditionals in that domain. For instance, if we defined a version space with a set of conditionals appropriate to the text-editing domain, then that space and SMARTedit's Action version space could be substituted into this version space to learn text-editing programs with loops and conditionals.

6.2 State configurations

Given a fixed bias, varying the contents of the state directly affects the difficulty of the learning problem. We identify several possible state configurations:

Incomplete The state contains a subset of the variables available to the program; some relevant variables are hidden.

Variables observable The state includes the complete set of variables available to the program.

Step observable The state includes the complete set of variables available to the program, as well as a unique identification of the program step executed between each pair of consecutive states.

Fully observable The state includes the complete set of variables available to the program, unique identification of the program step, and a set of change predicates indicating whether each variable has changed between each pair of consecutive states.

When the state is fully observable, the problem of learning programs is easiest. The program step identifier uniquely determines which action causes this state change, and the change predicates focus attention on the relevant variables in the state. This state configuration is approximately the configuration available to the SMARTedit system. For simple programs with straight-line control flow, the program-learning problem reduces to learning a sequence of primitive actions. For more complex programs with inner loops, if the user were to indicate the complete loop segmentation (beginning, end, and between each iteration) it would be equivalent to knowing the program step for each action.

When parts of the complete state are unavailable, the learning problem becomes progressively more difficult. Without change predicates (a step-observable state), the system must compare the two states in order to determine what aspect of the state has changed from one state to the next. If the state difference is not obvious (perhaps because one variable has been assigned the same value it previously held), then the system must maintain all such candidate hypotheses. In this chapter we focus on learning programs with this state configuration. While the presence of a program step identifier may seem to be a strong assumption, we note that the program step identifier merely provides a unified view of the types of program structure information that may be available. For instance, the program step maybe derived implicitly from the segmentation information given in a SMARTedit trace.

Without the program step identifier (a variables-observable state), the program structure (*i.e.*, loops and conditionals) must also be inferred from the trace. The system must guess which action generates each state change, as well as guessing whether any two pairs of states are instances of the same statement in the program.

When part of the state is hidden, the learning problem is most difficult. The correct program may reference variables not observable in the state. As a concrete example, imagine learning a text-editing program that looped over the names of the user's relatives. If such a

list were available to the learner as part of the state, such a program would be much easier to learn; without this knowledge, the learner cannot predict which name should be inserted next.

This chapter presents an approach to learning programs from traces with the step-observable state configuration.

6.3 Learning conditionals

Previous chapters of this thesis have addressed the problem of learning individual statements from examples of their input/output behavior. In this section we show how to extend this approach to learning conditionals—statements that test a Boolean expression and branch to different parts of the program depending on its truth value.

We express control flow statements as functions that map from an input state to a new state in which the program step identifier has been updated to indicate which statement to execute next. Each control flow statement is a function that, given the state, tests a condition and produces the index of the next step to be executed based on the truth value of the condition. These functions are expressed as a join of a domain-dependent condition version space (to be supplied for a particular programming language), and the TrueStep and FalseStep spaces that each contain one program step identifier (which branch to take on the true and false values, respectively, of the condition). Note that this join is not independent; the true and false branches depend on the condition. For example, suppose condition p determines whether to branch to program step p_t if true and p_f otherwise. Condition $\neg p$ must then cause a branch to p_f if true and p_t otherwise. Both combinations are equally valid and should both be in the version space. In practice, however, we assume that the first branch observed in the trace corresponds to the true value of the condition, and learn the appropriate formulation of the condition. In order for this approach to succeed, the condition version space must be complete; that is, for all conditions p, the inverse condition $\neg p$ must also be in the version space.

The main problem in learning the condition is learning which expression correctly differentiates between the two possible branches. Without loss of generality, assume that the

Program step	Variable values			Label
	i	j	k	
1	18	12	0	S_0
2	18	12	0	S_1
3	18	12	6	S_2
4	12	12	6	S_3
1	12	6	6	S_4
2	12	6	6	S_5
3	12	6	0	S_6
4	6	6	0	S_7
1	6	0	0	S_8
5	6	0	0	S_9

Figure 6.2: Trace of greatest common divisor program in execution

first branch observed in the trace corresponds to the truthful evaluation of the condition; label the corresponding input state as a positive example. When the same conditional statement is executed elsewhere in the trace, it may result in a branch to a different step; the statement's input state in such a situation is labelled as a negative example of the condition.

Given positive and negative examples, *i.e.*, states in which the condition either holds or does not hold, the learning problem is to infer the condition that correctly classifies the states. Note that unlike the problem of learning actions as functions, the condition learning problem is simply a concept learning problem; traditional version spaces and generality partial orderings apply. We give an example of such a condition-learning version space in the next section.


```
1: while j > 0 do:
2:       k := i % j;
3:       i := j;
4:       j := k;
5: od
```

Figure 6.3: Code for greatest common divisor program

6.4 Learning Python programs

6.4.1 A motivating example

To illustrate the problem of learning programs from traces, consider a program that computes the greatest common divisor of two integers, shown in Figures 6.2 and 6.3. Given program traces such as the one shown in Figure 6.2, the goal is to infer a program that is capable of producing those traces. Such a program is shown in Figure 6.3.

The program in Figure 6.3 contains two types of statements: a WHILE loop that tests an inequality condition and several forms of assignment statements. Each assignment statement, when executed in one state, produces another state in which a new value has been assigned to the appropriate variable. The WHILE statement tests a condition and outputs a state in which the variable values have not changed, but the program step has been updated to indicate which branch to follow.

If change predicates are available, or if the changed variable can be easily detected by comparing the pair of states, the learning problem is straightforward. For example, consider states S_1 and S_2 in Figure 6.2. In state S_1, variable k has value 0, while in state S_2, k has value 6. Variables i and j preserve their previous values of 18 and 12, respectively. The intervening program statement must be an assignment to variable k. Assignment statements that are consistent with this state change include the correct statement $k := i \bmod j$, as well as the statements $k := 6$, $k := i - j$, $k := j - 6$, and so on.

If change predicates are not available, and the states are identical except for the program

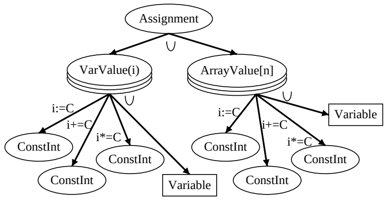

Figure 6.4: Version space for learning primitive statements in SMARTpython

step indentifer, then several types of statements are consistent with this state change. The statement may be a control flow statement which tests a condition and branches to a different part of the program; it may be an assignment statement that assigns to one of the variables the same value it previously held.

Each primitive statement causes control flow to resume at a deterministic next step. In the example, after executing step 4, the program always executes step 1 (testing the condition of the loop for the next iteration). Unlike primitive statements, conditional statements cause control flow to branch to one part of the program or another depending on the truth value of a Boolean expression. By examining which step the program branches to after the conditional, we distinguish states in which the condition is true from those in which it is false. Equivalently, these states are positive and negative examples of the target condition.

We have applied our version space algebraic framework to learn programs written in a subset of the Python programming language. The next section describes SMARTpython and its version space implementation.

6.4.2 Version space implementation

The domain-independent framework for learning programs shown in Figure 6.1 is parameterized by two domain-specific version spaces: one for learning primitive statements, and another defining the set of Boolean expressions recognized as conditions. This section presents the two domain-specific version spaces used in SMARTpython.

Figure 6.4 shows the version space to learn Python statements. Each node in the hier-

Table 6.1: Grammar describing the assignment statements supported in the SMARTpython system

$\langle$*Assignment*$\rangle$ ::= $\langle$Var$\rangle$ $\langle$Op$\rangle$ $\langle$Value$\rangle$
$\langle$*Assignment*$\rangle$ ::= $\langle$ArrayVar$\rangle$ $\langle$Op$\rangle$ $\langle$Value$\rangle$
$\langle$Var$\rangle$ ::= i | j | k
$\langle$ArrayVar$\rangle$ ::= A[$\langle$Var$\rangle$] | A[0] | A[1] | ... | A[N]
$\langle$Value$\rangle$ ::= Constant | $\langle$Var$\rangle$ | $\langle$ArrayVar$\rangle$
$\langle$Op$\rangle$ ::= := | += | -= | *= | /= | %=

archy represents a version space. A stacked node represents multiple parameterized version spaces. Rectangular leaf nodes indicate enumerated version spaces, while oval leaf nodes indicate boundary-represented spaces. Nontrivial transforms are shown as edge labels.

The target space is called Assignment; the current version of the language only supports assignment statements that set a variable to a certain value. The Assignment space is a union of a number of subspaces, one each for each scalar variable and array member supported by the language. For example, SMARTpython supports three scalars (i, j, k) and a single array variable (A) of length five; the Assignment space is a union of eight version spaces.

Each of these eight spaces is equivalent, parameterized by the variable on the left hand side of the assignment. Consider the VarValue(i) space as an example. This space represents all possible assignments to variable i. It is composed of a union of three ConstInt spaces and one Variable space. Each ConstInt space represents a set of functions $f(x) = C$. Transforms convert each of these ConstInt spaces into the sets of assignment statements $i := C$, $i := i{+}C$, and $i := i * C$, respectively. Finally, the Variable version space represents an enumerated set of assignment statements, such as $i := j$ and $i := A[k]$. The grammar shown in Table 6.1 defines the assignment statements supported by SMARTpython.

The *VarValue(i)* and *ArrayValue[n]* spaces obey what we term the *fast consistency* property: the consistency of all hypotheses in the space can be checked against most training examples in $O(1)$ time. In these cases, large portions of the search space can be collapsed without having to examine each individual hypothesis or update each version space in the

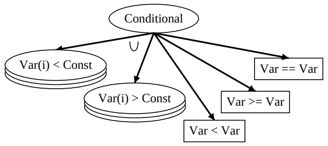

Figure 6.5: Version space for learning Boolean expressions in SMARTpython

subtree. For example, if it is known that variable i's value has changed, then no assignment to a different variable can be consistent. Therefore, all the hypotheses in other member spaces of the *Assignment* union, such as *VarValue(j)* and *ArrayValue[n]*, are immediately known to be inconsistent.

Note that if we had access to a change predicate that identified the variable that is updated in each statement, then we could factor out the name of the variable from the value to which it was assigned, and replace the eight-way union with a single join of variable name with variable value. If the variable name were uniquely identified by a change predicate, the variable name version space would immediately converge to the correct hypothesis. Since we do not make this assumption, we must represent the full cross product in the join of variable name and variable value; the eight-way union is the manifestation of this cross product.

An example serves to illustrate the dependence of the join. Suppose $i = 0$ and $j = 3$ hold in the initial state, and the variable values remain unchanged after target statement has been executed. The two hypotheses $i := 0$ and $j := 3$ are consistent with the target statement, but $i := 3$ is not; the right hand side of the assignment statement is not independent of the left hand side. On the other hand, if a change predicate indicates that variable i has been updated from one state to the next, then this knowledge may be used to restrict the set of consistent hypotheses. The assignment statement may be factored into a version space for the left hand side and another for the right hand side, and a separate version space maintained for each side.

Figure 6.5 shows the Boolean expressions supported as conditions in the SMARTpython

Table 6.2: Grammar describing the conditions supported in the SMARTpython system

⟨*Condition*⟩ ::= ⟨Var⟩ < ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ >= ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ = ⟨Var⟩
⟨*Condition*⟩ ::= ⟨Var⟩ > Constant
⟨*Condition*⟩ ::= ⟨Var⟩ < Constant
⟨Var⟩ ::= i | j | k
⟨ArrayVar⟩ ::= A[⟨Var⟩] | A[0] | A[1] | ... | A[N]

system. The target Conditional version space is a union of several types of conditions. Each Var(i)<Const space is a parameterized space that contains the set of conditions of the form $i < C$ for constant values C. The union contains one of each of these spaces for each scalar variable and each array member supported by the language. The Var(i)>Const spaces are defined similarly. The Var<Var and Var>=Var version spaces are enumerated spaces that contain the set of conditions comparing one variable with another. The complete set of conditions expressible in SMARTpython is given the grammar in Table 6.2.

Recall that we can label examples of the condition (*i.e.*, program states) as positive or negative based on the branch followed after executing the conditional statement. Consider the Var(i)<Const version space, for instance. A hypothesis in this space is consistent with a positive example state iff the corresponding condition evaluates to the true in the state. A hypothesis is consistent with a negative example iff the corresponding condition evaluates to false in the state. We efficiently represent this version space by its boundary sets by realizing that less-than hypotheses are partially ordered by a generality relation. One hypothesis $i < C_1$ is more general (holds in more states) than another $i < C_2$ iff $C_1 > C_2$. Thus the consistent hypotheses in the version space may be represented by the boundaries of the version space, $i < C_S$ and $i < C_G$, such that $i < C_S$ is the most specific consistent hypothesis and $i < C_G$ the most general. The other condition version spaces that test a variable against a constant may also be represented by boundary sets.

This work on learning conditions from state examples is very similar to work on detecting invariants [20]; such work could be used as a module in our learner.

6.5 Evaluation

We evaluate the SMARTpython system by testing how many traces are required for it to learn a variety of sample programs. We have drawn sixteen programs from the following sources:

- UW CSE 100 (Autumn 2000), the introductory Fluency in Information Technology class designed for non-CS-majors; quiz and exam programs were translated from Visual Basic to our subset of Python.
- Introductory programming and computer science textbooks [19, 24, 13].
- Classic algorithms, such as bubble sort and primality test.

The complete source code for our test programs is listed in Appendix D. For each program, we generate a set of 20 training examples and 100 test examples. Each example begins with a random initial state contructed by assigning a random value between -100 and 100 to each scalar variable and to each element of a length-5 array. To create the training examples, we construct a program trace by simulating execution of the program starting with the given initial state. The test examples consist only of the random initial state.

We evaluate the accuracy of the learner by training it incrementally on the 20 training examples, and testing its accuracy after each new training example. The accuracy of the learner is the fraction of the 100 test examples for which the learned program generates the correct trace. If the version space has not converged to a single program, at each step we execute the highest probability program statement in the version space. Since each training and test example is randomly generated, we repeat this entire procedure 25 times and average the results.

On average, our system requires 5.1 examples to reach 100% accuracy; the median number of examples is 5. Figure 6.6 shows the accuracy versus number of training examples for

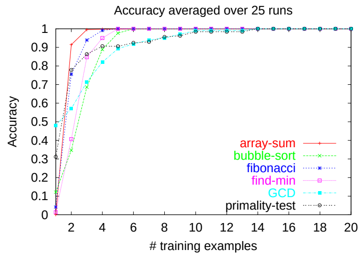

Figure 6.6: SMARTpython's accuracy on a representative selection of programs

the six most complex programs. (The remaining programs behave similarly.) The greatest-common-divisor program shown in Figure 6.3 achieves 100% accuracy with 14 traces. The bubble-sort program achieves 100% accuracy with 6 traces. These results indicate the feasibility of our approach and show that nontrivial programs can be learned from a relatively small number of traces.

Chapter 7

FUTURE WORK

This work has demonstrated the feasibility of learning programs from traces of their execution behavior; however, many opportunies for extending the work remain. In this chapter we present a few of the potential directions such extensions could take.

Future work falls roughly into three categories: extensions to the existing SMARTedit and SMARTpython systems, version space algebra extensions, and applying our approach to new domains. We consider each of these in turn.

7.1 Extensions to the SMARTedit system

While the bias defined by SMARTedit's version space provides a useful set of text-editing actions, it is desirable to increase the expressiveness of SMARTedit's instruction language. Regular expression matching is one feature often requested by our trial users. The tokenized string search already supported by SMARTedit covers one special case of regular expression matching.

An alternative way to extend the expressiveness of the programming language supported by SMARTedit is to incorporate a syntax parser and define hypotheses that refer to parts of the parse tree. For instance, a C-language parser could be invoked when editing a source code file; useful locations in such a file include "the start of the next function definition" and "the end of the expression under the cursor".

Chapter 6 examined the problem of learning loops and conditionals in an abstract programming language. We have applied some of that work to the SMARTedit system to learn programs with loops in a limited form, by asking the user to explicitly indicate the segmentation between the loop and the main program. Investigating the types of conditionals that would be useful in a text-editing context and adding them to SMARTedit as loop termination conditions is a natural next step to take.

The probabilistic framework allows us to associate a probability with each hypothesis in the verison space. We have chosen these probabilities heuristically such that the system produces reasonable predictions. A future direction is to learn these probabilities empirically, perhaps by conducting a study of common text-editing behavior. Another direction is to adapt these probabilities to a user's preferences and work practices over time. For example, people manipulating highly-formatted tabular data may perform row/column positioning more than they use string search positioning, whereas people editing semi-structured HTML data may be the opposite. Further, the probabilities could be biased according to the file type. For instance, when editing HTML data, more probability could be given to string search hypotheses that match the tags defined in the HTML standard.

Many of our desired extensions increase the size and complexity of the search space, which increases the sample complexity correspondingly. In order to keep this approach feasible for PBD, where users are rarely motivated to provide more than a handful of training examples, the learning process must employ more information to speed the learning process. One promising direction is the paradigm of *active learning* [12], in which the learner selects the example which most reduces the size of the version space. Such example selection would require a user interface for clearly communicating the learner's intentions to the user.

More generally, the learning process as a whole benefits from a mixed-initiative interaction. Programming by demonstration is essentially a teacher-student interaction in which the teacher (user) attempts to transfer knowledge to the student (learner) by communicating examples. We can extend this model in several ways to increase the rate of learning as well as minimize the teacher's use of scarce resources (time and effort). In other work [90] we have shown how a mixed-initiative interface, in which both teacher and student take responsibility for the learning interaction, can speed learning. Additionally we demonstrate that the improved learner can adapt to a particular user's preferences over time.

While the work on mixed-initiative interfaces is promising, it is only a first step in the creation of user interfaces that do more with less input from the user. Significant directions to extend this work include: incorporating information about the user's context (*e.g.*, integrating with other applications such as calendars and web browsers) and investigating other modalities for communicating with the user, such as speech or gestures.

7.2 *Extensions to the SMARTpython system*

Chapter 6 presented experimental results measured in terms of the number of complete program execution traces required to learn a program. Within a single trace, the number of examples of each particular program statement can range from as little as a single example to many (if the statement is part of an oft-executed loop). Programs with a single conditional not enclosed in a loop are particularly troublesome to learn because each program execution results in only a single example of the conditional. A more accurate performance metric for the system could be a measure of the number of examples required to learn each program statement.

While capable of expressing a variety of interesting programs, the language supported by SMARTpython is still a small fraction of that used in real programs. Future work will investigate scaling up the system to learn programs with more variables (both scalar and array), adding more data types (*e.g.*, dictionaries, containers, and objects), more complex control flow (*e.g.*, subroutines and recursion), and more expressions such as conjunctive conditions.

As we increase the complexity of the search space, we will need to consider heuristic search methods; the version space algebra allows us to combine heuristic search with complete search within the same framework. Finally, we will conduct a formal analysis of our approach and investigate tradeoffs between expressiveness and learnability as we scale to larger programs.

7.3 *Version space algebra extensions*

Increasing the expressiveness of either of the above systems requires new techniques to handle searching larger and more complex spaces. As the search space increases, efficiency becomes a concern. The version space may reach a size where, even with boundary sets, a complete search is not feasible. One direction of future work is to investigate means of combining boundary-represented version spaces with heuristically-searched spaces in order to efficiently search the global space. In addition, we may need to investigate the use of sampling in order to efficiently execute such large version spaces. Also, as the number of

component version spaces grows, it may be difficult to determine the best structure for a domain; one direction of future work is to search automatically through the space of version spaces in order to find the best structure for a domain.

One possible technique for learning from small numbers of examples is version space revision, or dynamically weakening the bias to include less-likely hypotheses if the stronger initial biases fail. For instance, the learner could begin its search with a core version space representing a set of common hypotheses. Once this version space collapses, the learner could then shift its focus to a larger, expanded version space containing a wider selection of less-probable hypotheses.

One assumption we have made in this work is the availability of pre-parsed examples. An area for future work is to investigate the problems (such as credit assignment) that arise when the parsing of an example into sub-examples is not given *a priori*.

Another assumption we have made is noise-free examples. We believe that the probabilistic framework can be enhanced to handle noisy examples. One possibility is to reduce the probability of inconsistent hypotheses, rather than eliminate them entirely from the version space. We will need to investigate mechanisms for efficiently maintaining such a probabilistic version space. Norton and Hirsh [65] present an alternative method, where rather than maintaining a single version space containing potentially-inconsistent hypotheses, they maintain a set of version spaces, each of which is consistent with some interpretation of the data. Future work will investigate applying their method to our probabilistic representation.

In programming by demonstration domains, users will generally find it inconvenient to explicitly specify the program step identifier; learning programs without this extra bit of state will be one challenging direction of future research. Without the program step identifier, we will need to search for a program structure consistent with the traces; the version space will then be a union of all possible program structures.

The version space algebra operators we have defined here (union, join, and transform) are sufficient to represent the learning problems we have faced during the design and implementation of our two systems. However, when the version space algebra approach is applied to new domains it is quite possible that more operators will be necessary to express the functions and concepts of the new domain. Future work will then investigate useful new operators and study their properties.

We have begun the creation of a version space library—a collection of domain-independent version spaces that can be reused in new applications. Ideally, this version space library would to grow to include many useful modules, and make it even easier for developers to design new version spaces to apply machine learning to their domains of interest.

7.4 Cross-application PBD

The ultimate domain for programming by demonstration is not any one particular application, but a cross-application solution that allows users to demonstrate programs to automate repetitive tasks spanning all of their applications. Few users do all their work within a single application; many tasks cross the boundaries between applications to manipulate data from spreadsheets, calendars, web browsers, email clients, and presentations. Providing cross-application PBD is the ultimate goal for the work presented here. Future work will apply our version space algebra to more domains and tie them together using the SMARTpython framework in order to achieve our goal of cross-domain programming by demonstration.

Wrapper construction by demonstration is a promising direction for future work. Previous work in automating wrapper construction has followed two approaches. Kushmerick [38] focuses on automated wrapper construction, where the user is not involved in the training process beyond providing labelled examples. At the other extreme, Bauer [6] takes a human-centered approach to wrapper construction, guiding a user through a training dialogue to build each wrapper. Our approach combines the best of both of these systems. Unlike Bauer's work, our formal machine learning approach is capable of learning from multiple examples; the search space of possible wrappers is clearly defined. In addition, the availability of an HTML parser would enable the definition of significantly more interesting hypotheses than the literal string matching supported by Kushmerick's system.

Chapter 8

CONCLUSIONS

Programming by demonstration allows non-programmers to construct programs to automate repetitive tasks simply by demonstrating the program's behavior on a concrete example. The key problem in a PBD system is how to *generalize* from the demonstrated program trace(s) to a robust program capable of performing the task in new contexts. Since a human is responsible for generating each demonstration, a PBD system must be able to generalize from a very small number of examples; typical users are willing to provide at most a few examples of the target program's behavior. Previous approaches to PBD have employed heuristic, domain-specific rules to generalize from small numbers of examples. In contrast, our work casts programming by demonstration as a domain-independent machine learning problem. This thesis makes four main contributions: modelling PBD as an inductive learning problem; the version space algebra framework; the SMARTedit system for programming by demonstration in the text-editing domain; and a general framework for learning programs from traces.

8.1 Contributions

Contribution 1: Programming by demonstration as inductive learning.

We model actions or statements in a program as functions that map from one state of the application to another. By observing the state of the application as a user demonstrates the target program, we collect the sequences of states representing the actions the user intends. Learning programs is thus a matter of learning these state transformation functions from examples of their inputs and outputs.

Unlike classification learning, in which the target function maps from an input to a discrete-valued prediction, learning programs requires an algorithm capable of inferring complex functions—functions that map from one complex object to another.

Chapter 3 presented our formal model of PBD as an inductive learning problem. We have extended version spaces beyond concept learning (functions that map from an input to a Boolean output) to complex-function learning. The original version space framework employed a generality partial ordering over hypotheses in the space to achieve efficient representation of the space by its boundary sets. However, efficient representation is not dependent on the generality ordering; we have demonstrated that efficient representation requires only some partial ordering, and given an example of such a non-generality ordering for search hypotheses in the text-editing domain.

Contribution 2: Version space algebra framework for building and searching large complex-function spaces.

Learning programs requires a large search space that contains the set of all program statements supported by the language. Section 3.2 described the version space algebra that allows an application designer to construct a hierarchical complex-function search space by composing together simpler spaces. The modularity of our approach means that we can construct version space libraries—collections of domain-independent version spaces that may be reused in new applications. We have also shown that the version space algebra is efficient: each operator preserves PAC learnability. Finally, we have proposed a probabilistic framework that associates a probability with each hypothesis in the space, which is one way to incorporate domain knowledge into the learning process.

Contribution 3: The SMARTedit system for programming by demonstration in the text-editing domain.

Using version space algebra we have constructed a PBD system that automates repetitive text-editing tasks. Our SMARTedit system (Chapter 4) learns text-editing programs given as few as a single training example. The SMARTedit user interface is similar to a typical macro recording interface: after pressing a button, a user demonsrates the target task on concrete data, and then presses another button to end the recording. SMARTedit, however, differs from standard macro recording in that it generalizes from the demonstration to a robust program that is capable of completing the task in new contexts.

Chapter 5 presents our evaluation of the SMARTedit system, which takes two forms: a set of scenarios on which we test SMARTedit's learning ability, and a user study designed

to evaluate its impact on users.

To test SMARTedit's ability to learn useful programs from a small number of examples, we collected a number of repetitive text-editing tasks drawn from real-world problems, including bibliographic editing tasks. The results show that a program that generalizes correctly for each of these scenarios can be learned quickly in as few as one or two training examples.

Our user study indicates that novice users (computer science undergraduates) are able to quickly learn how to use SMARTedit, and complete their tasks faster and with less effort. In addition, a survey completed by study participants indicates that they found SMARTedit useful and would use it again.

Contribution 4: A general framework for learning programs from traces.

Our framework casts programming by demonstration as a machine learning problem. More broadly, we have applied this framework to learn programs in an abstract programming language containing loops, conditionals, and arrays. Chapter 6 abstracts away from the problem of learning single program statements, and shows how to learn complete programs from traces. To learn these more expressive programs, we assume that the state includes a program step identifier that indicates the mapping between state changes and program steps.

The generalized framework is based on the idea that a program step (a conditional, loop statement, or primitive statement) is a function that maps from one program state to another. Included within the output state is the program step identifier which indicates the index of the next program step to be executed. This framework is domain-independent; given two version spaces describing domain-dependent concepts (conditions and primitive statements), the framework supports learning complete programs in the language.

We employ this generalized framework to learn programs written in a subset of the Python programming language. We have applied SMARTpython to a collection of fourteen simple programs drawn from introductory programming classes and computer science textbooks. For example, the SMARTpython system learns a bubble-sort program given as few as six traces of the program's execution.

8.2 Conclusion

Machine learning may be viewed with generality as learning programs from examples of what they should do. We have proposed a novel framework for learning procedures from traces of their execution behavior, helping to extend the reach of supervised machine learning applications beyond classification learning.

We have employed this framework to address the problem of programming by demonstration. PBD allows even novice users to construct programs to automate repetitive tasks, simply by demonstrating the behavior of the program on a concrete example.

In order to generalize from small numbers of examples, previous approaches to PBD have used heuristic, domain-dependent rules to guide generalization. In contrast, this thesis presents a robust machine learning approach. We have applied it to the text-editing domain as well as an abstract programming language. PBD has the potential to allow users to do more computing with less effort. The solution presented here is a step towards this goal.

BIBLIOGRAPHY

[1] P. Agre and D. Chapman. What are plans for? *Robotics and Autonomous Systems*, 6(1–2):17–34, 1990.

[2] David Andre and Stuart J. Russell. Programmable reinforcement learning agents. In *Advances in Neural Information Processing Systems*, volume 12, 2000.

[3] P. Andreae. Constraint limited generalization: Acquiring procedures from examples. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 6–10, 1984.

[4] N. Ashish and C. Knoblock. Semi-automatic wrapper generation for Internet information sources. In *Proceedings of the Second IFCIS International Conference on Cooperative Information Systems*, pages 160–169, Los Alamitos, CA, June 1997. IEEE-CS Press.

[5] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12:73–119, 1979.

[6] M. Bauer, D. Dengler, and G. Paul. Instructible information agents for web mining. In *Proceedings of the 2000 Conference on Intelligent User Interfaces*, January 2000.

[7] Mathias Bauer and Dietmar Dengler. Trias: Trainable information assistants for cooperative problem solving. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 260–267, May 1999.

[8] A. W. Biermann. On the Inference of Turing Machines from Sample Computations. *Artificial Intelligence*, 3:181–198, 1972.

[9] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's razor. *Information Processing*, 24(6):377–80, 1987.

[10] E. Charniak and R. Goldman. A probablistic model of plan recognition. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 1, pages 160–5, July 1991.

[11] E. Charniak and R.P. Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[12] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Machine Learning*, 15(2):201–221, 1994.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1991.

[14] J. Cowie and W. Lehnert. Information extraction. *C. ACM*, 39(1):80–91, 1996.

[15] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–217. MIT Press, Cambridge, MA, 1993.

[16] Allen Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, 1993.

[17] B. D. Davison and H. Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time Series Problems*, pages 5–12, Madison, WI, 1998. AAAI Press. Technical Report WS-98-07.

[18] T.G. Dietterich. State abstraction in MAXQ hierarachical reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 12, 2000.

[19] Edsger W. Dijkstra. *The Discipline of Programming*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1976.

[20] M. Ernst, A. Czeisler, W. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 449–458, Limerick, Ireland, June 2000.

[21] Oren Etzioni, Steve Hanks, Daniel Weld, Denise Draper, Neal Lesh, and Mike Williamson. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 115–125. San Francisco, Calif.: Morgan Kaufmann, 1992.

[22] Y. Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of Intelligent User Interfaces '98*, pages 101–108, 1998.

[23] C. Green and D. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10:241–279, 1978.

[24] David Gries. *The Science of Programming*. Springer-Verlag, New York, NY, 1981.

[25] Daniel C. Halbert. SmallStar: Programming by Demonstration in the Desktop Metaphor. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 102–123. MIT Press, Cambridge, MA, 1993.

[26] D. Haussler. Quantifying inductive bias. *Artificial Intelligence*, 36(2):177–221, 1988.

[27] JL Hellerstein, TS Jayram, and I Rish. Recognizing end-user transactions in performance management. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*. Menlo Park, CA: AAAI Press, 2000.

[28] H. Hirsh and B. D. Davison. An adaptive unix command-line assistant. In *Proceedings of the First International Conference on Autonomous Agents*, pages 542–543, Marina del Ray, CA, 1997. ACM.

[29] Haym Hirsh. Incremental version-space merging. In *Proceedings of the Seventh International Conference on Machine Learning (ICML90)*, pages 330–338. Morgan Kaufmann, 1990.

[30] Haym Hirsh. Theoretical underpinnings of version spaces. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 665–670. San Francisco, CA: Morgan Kaufmann, July 1991.

[31] Haym Hirsh, Nina Mishra, and Leonard Pitt. Version spaces without boundary sets. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 491–496. Menlo Park, CA: AAAI Press, July 1997.

[32] J. Hobbs. The generic information extraction system. In *Proc. 4th Message Understanding Conf.*, 1992.

[33] H.A. Kautz and J.F. Allen. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, 1986.

[34] M. Kearns and U. Vazirani. *An introduction to computational learning theory.* MIT, 1994.

[35] Benjamin Korvemaker and Russell Greiner. Predicting unix command lines: Adjusting to user patterns. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 230–235, Austin, Texas, July 2000. Menlo Park, CA: AAAI Press.

[36] J. R. Koza. *Genetic programming II : Automatic Discovery of Reusable Programs.* Massachusetts Institute of Technology, Cambridge, MA, 1994.

[37] David Kurlander. Chimera: Example-Based Graphical Editing. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 270–290. MIT Press, Cambridge, MA, 1993.

[38] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1-2):15–68, 2000.

[39] N. Kushmerick, D. Weld, and R. Doorenbos. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 729–737. San Francisco, CA: Morgan Kaufmann, 1997.

[40] T. Lane. Hidden markov models for human/computer interface modeling. In *Proceedings of the IJCAI-99 Workshop on Learning about Users*, pages 35–44, 1999.

[41] P. Langley and H. A. Simon. Applications of machine learning and rule induction. *Communications of the ACM*, 38:54–64, November 1995.

[42] Tessa Lau and Daniel S. Weld. Programming by Demonstration: an Inductive Learning Formulation. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces (IUI 99)*, pages 145–152, Redondo Beach, CA, USA, January 1999.

[43] N. Lesh, C. Rich, and C. Sidner. Using plan recognition in human-computer collaboration. In *Proceedings of the Seventh Int. Conf. on User Modelling*, Banff, Canada, July 1999.

[44] Neal Lesh. Adaptive goal recognition. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence.* San Francisco, CA: Morgan Kaufmann, 1997.

[45] Neal Lesh. *Scalable and Adaptive Goal Recognition.* PhD thesis, University of Washington, 1998.

[46] Neal Lesh and Oren Etzioni. A sound and fast goal recognizer. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1704–1710. San Francisco, CA: Morgan Kaufmann, 1995.

[47] Neal Lesh and Oren Etzioni. Scaling up goal recognition. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pages 178–189, 1996.

[48] Henry Lieberman. Tinker: A Programming by Demonstration System for Beginning Programmers. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 49–64. MIT Press, Cambridge, MA, 1993.

[49] Henry Lieberman. Integrating User Interface Agents with Conventional Applications. In *Proceedings of the 2000 Conference on Intelligent User Interfaces*, pages 39–46, San Francisco, CA, 1998.

[50] Henry Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2001.

[51] Pattie Maes and Robyn Kozierok. Learning interface agents. In *Proceedings of AAAI-93*, pages 459–465, 1993.

[52] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence*, 6, 1975.

[53] Toshiyuki Masui and Ken Nakayama. Repeat and Predict—Two Keys to Efficient Text Editing. In *Conference on Human Factors in Computing Systems (CHI '94)*, pages 118–123, 1994.

[54] David Maulsby and Ian H. Witten. Cima: an interactive concept learning system for end-user applications. *Applied Artificial Intelligence*, 11:653–671, 1997.

[55] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[56] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[57] Dan Hua Mo. Learning Text Editing Procedures from Examples. Master's thesis, University of Calgary, December 1989.

[58] Ion Muslea. RISE: Repository of Online Information Sources Used in Information Extraction Tasks. http://www.isi.edu/~muslea/RISE/, retrieved on 10/4/2000.

[59] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4(1/2):93–114, March 2000.

[60] Ion Muslea, Steven Minton, and Craig A. Knoblock. Selective sampling with redundant views. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 621–626. Menlo Park, CA: AAAI Press, 2000.

[61] Brad A. Myers. Peridot: Creating User Interfaces by Demonstration. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 125–153. MIT Press, Cambridge, MA, 1993.

[62] C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. de Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, Amsterdam, the Netherlands, 1996.

[63] C.G. Nevill-Manning and I.H. Witten. Identifying Hierarchical Structure in Sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[64] Robert P. Nix. Editing by Example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.

[65] S. W. Norton and H. Hirsh. Classifier learning from noisy data as probabilistic evidence combination. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 141–146, Menlo Park, CA, 1992. AAAI Press.

[66] John Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[67] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In Michael I. Jordan, Michael J. Kearns, and Sara A. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. The MIT Press, 1998.

[68] Gordon W. Paynter. Generalising Programming by Demonstration. In *Proceedings Sixth Australian Conference on Computer-Human Interaction*, pages 344–345, Nov 1996.

[69] Gordon W. Paynter. *Automating iterative tasks with programming by demonstration.* PhD thesis, University of Waikato, February 2000.

[70] D. Pynadath and M.P. Wellman. Generalized queries on probabilistic context-free grammars. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):65–77, 1998.

[71] D.V. Pynadath and M.P. Wellman. Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, pages 507–514, 2000.

[72] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–285, February 1989.

[73] C. Rich and C. Sidner. Segmented interaction history in a collaborative agent. In *Third Int. Conf. Intelligent User Interfaces*, pages 23–30, Jan 1997.

[74] Jean-David Ruvini and Christophe Dony. APE: Learning User's Habits to Automate Repetitive Tasks. In *Proceedings of the 2000 Conference on Intelligent User Interfaces*, pages 229–232, New Orleans, LA, January 2000.

[75] J. Schlimmer and L. Hermens. Software agents: Completing patterns and constructing user interfaces. *J. Artificial Intelligence Research*, pages 61–89, 1993.

[76] J. Shavlik and G. DeJong. An explanation-based approach to generalizing number. In *Proceedings of IJCAI-87*, pages 236–238, August 1987.

[77] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring lisp programs from examples. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 260–267, Tblisi, Georgia, USSR, 1975. San Francisco, CA: Morgan Kaufmann.

[78] L. Siklóssy and D. A. Sykes. Automatic program synthesis from example problems. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 268–273, Tblisi, Georgia, USSR, 1975. San Francisco, CA: Morgan Kaufmann.

[79] Benjamin D. Smith. *Induction as Knowledge Integration.* PhD thesis, University of Southern California, 1995.

[80] D. Subramanian and J. Feigenbaum. Factorization in experiment generation. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 518–522, Philadelphia, PA, 1986. Menlo Park, CA: AAAI Press.

[81] Phillip D. Summers. A Methodology for LISP Program Construction from Examples. *J. ACM*, 24(1):161–175, January 1977.

[82] R. S. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.

[83] L. Valiant. A theory of the learnable. *C. ACM*, 27(11):1134–42, 1984.

[84] K. VanLehn and W. Ball. A version space approach to learning context-free grammars. *Machine Learning*, 2:39–74, 1987.

[85] M. Vilain. Getting serious about parsing plans: a grammatical analysis of plan recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 190–197, 1990.

[86] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[87] Xuemei Wang. Planning while learning operators. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, May 1996.

[88] D. Weld. An introduction to least-commitment planning. *Artificial Intellligence Magazine*, 15(4):27–61, Winter 1994. Available at `ftp://ftp.cs.washington.edu/pub/-ai/`.

[89] I.H. Witten, C.G. Nevill-Manning, and D.L. Maulsby. Interacting with learning agents: implications for ml from hci. In *Workshop on Machine Learning meets Human-Computer Interaction, ML'96*, pages 51–58, July 1996.

[90] Steven A. Wolfman, Tessa Lau, Pedro Domingos, and Daniel S. Weld. Collaborative Interfaces for Learning Tasks: SMARTedit Talks Back. In *Proceedings of the 2001 Conference on Intelligent User Interfaces*, 2001.

[91] I. Zukerman, D.W. Albrecht, and A.E. Nicholson. Predicting users' requests on the www. In *Proceedings of the Seventh International Conference on User Modelling*, pages 275–284, June 1999.

Appendix A

SMARTEDIT VERSION SPACES

In this section, we summarize the version spaces used in the SMARTedit system, organized roughly in order of appearance starting from the target space Program.

Program The target version space. A Program consists of a join of a fixed number of ActionOrProg version spaces. Functions in this space map from an input state to an output state. The number of component version spaces is lazily determined from the first training example.

ActionOrProg A union of an Action and an UnsegmentedProgram. Functions in this space map from an input state to an output state.

UnsegmentedProgram A union of N Programs. Each Program consists of a join of a fixed number of ActionOrProg spaces. The first Program consists of a single ActionOrProg, the second has two, and so on. Functions in this space map from an input state to an output state. The number of component version spaces is lazily determined from the first training example.

Action A union of all the types of text-editing actions known to SMARTedit: Move, Insert, Delete, Select, Cut, Copy, Paste, DeleteSelection. Functions in this space map from an input state to an output state.

Move A transformation of a Location version space. Functions in this space map from an input state to an output state. Functions in the Location space map from an input state to a cursor position. The τ_o^{-1} transformation converts from the cursor position to a state in which the cursor is positioned at that location.

Delete A transformation of a join of two Location spaces specifying the left and right endpoints of a region. Functions in this space map from an input state to an output state. Functions in the join of two Locations map from an input state to a pair of locations. The τ_o^{-1} transformation converts a pair of locations to a state in which the text between those locations has been deleted.

Insert A transformation of a union of three version spaces: ConstStr, StrNumStr, and IndentStr. Functions in this space map from an input state to an output state. The τ_o transformation for the ConstStr component converts the output state into the string that is inserted between the input and output states. The τ_o^{-1} transformation converts a string into a state in which the string has been inserted into the text buffer at the current cursor position.

ConstStr Atomic version space of functions that accept no input and return a constant string.

StrNumStr A join of three version spaces: ConstStr, Number, and ConstStr. Functions in this space map from an input state to a string. The τ_o transformation extracts the string inserted between the input and output state, and splits it into three substrings: a non-numeric string prefix, a number, and the remainder of the string. Each string is used to update the associated version space. The τ_o^{-1} transformation concatenates the strings returned by each of the three component version spaces.

Number A union of two LinearInt version spaces. Functions in this space map from the input state to a string representing a number. Functions in the component spaces map from an integer to an integer. The τ_i and τ_o transformations extract the row number and the iteration number from the input and output states, respectively; row and iteration number are used to update the two component version spaces. The τ_o^{-1} transformation converts an integer into its string representation.

LinearInt Atomic version space of functions from integer to integer of the form $f(x) = x + C$.

IndentStr A join of two version spaces: Identity and ConstStr. Functions in this space map from the input state to a string. The τ_o transformation extracts the string inserted between the input and output state, and splits it into two substrings: a sequence of whitespace tokens, and the remainder of the string. The τ_i transformation for the Identity space extracts the amount of whitespace indentation on the current line of the input state. The two whitespace strings are used to update the Identity space, and the remainder of the string updates the ConstStr space. The τ_o^{-1} transformation concatenates two strings and returns a single string.

Identity Atomic version space containing at most a single function: $f(x) = x$.

Select A transformation of a Location version space. Functions in this space map from an input state to an output state. Functions in the Location space map from an input state to a cursor position. The τ_o^{-1} transformation converts from the cursor position to a state in which the selection spans the region between the cursor position in the input state and the new cursor position.

Cut Atomic version space representing a cut action.

Copy Atomic version space representing a copy action.

Paste Atomic version space representing a paste action.

DeleteSelection Atomic version space representing the action of deleting the selected text without copying it to the clipboard.

Location A union of several verion spaces: Search, WordOffset, CharOffset, RowCol, and CharClass. Functions in this space map from an input state to a location.

Search A union of several version spaces: two Left spaces, two Right spaces, and a LeftRight conjunctive space. A token typing τ_i transformation is applied to one each of the Left and Right spaces. This transformation converts the literal sequence of tokens in the text of the input state into a sequence of token *types*. Token types include lowercase letter, uppercase letter, whitespace, digit, and punctuation.

Left Atomic version space whose functions map from an input state to a location. The new location is the position at the end of the next occurrence (relative to the position in the input state) of a sequence of tokens. Each function in this space reflects a different sequence of tokens.

Right Atomic version space whose functions map from an input state to a location. The new location is the position and the beginning of the next occurrence (relative to the position in the input state) of a sequence of tokens. Each function in this space reflects a different sequence of tokens.

LeftRight Atomic version space whose functions map from an input state to a location. The new location is the position in the middle of a left matching string and a right matching string. Each function in this space reflects a different combination of left and right sequences of tokens.

WordOffset A transformation of a LinearInt space. Functions in this space map from an input state to a location. The τ_i transformation converts from the input cursor position to an integral word offset relative to the beginning of the text. The τ_o transformation converts the output cursor position similarly. The τ_o^{-1} transformation converts from an integer representing a word offset to a position in the text.

CharOffset A transformation of a LinearInt space. Functions in this space map from an input state to a location. The τ_i transformation converts from the input cursor position to an integral character offset relative to the beginning of the text. The τ_o transformation converts the output cursor position similarly. The τ_o^{-1} transformation converts from an integer representing a character offset to a position in the text.

RowCol A join of two version spaces: Row and Column. Functions in this space map from the input state to a cursor position. The τ_i and τ_o transformations for Row extract the row value of the cursor position in the input and output state, respectively. For the Column component, the transformations extract the column value. The τ_o^{-1} transformation converts a row and column value into a cursor position.

Row A union of two version spaces: AbsRow and RelRow. Functions in this space map from a row value to a row value.

AbsRow A transformation of a ConstInt version space. Functions in this space map from nothing to a row value. The τ_o transformation converts from a row value to an integer. The τ_o^{-1} transformation converts from an integer to a row value.

RelRow A transformation of a LinearInt version space. Functions in this space map from a row value to a row value. The τ_i and τ_o transformations convert from the row value of the input/output states to integers, and the τ_o^{-1} transformation converts from an integer to a row value.

AbsCol A transformation of a ConstInt version space. Functions in this space map from nothing to a column value. The τ_o transformation converts from a column value to an integer. The τ_o^{-1} transformation converts from an integer to a column value.

RelCol A transformation of a LinearInt version space. Functions in this space map from a column value to a column value. The τ_i and τ_o transformations convert from the column value of the input/output states to integers, and the τ_o^{-1} transformation converts from an integer to a column value.

CharClass An atomic version space representing the set of consistent sets of tokens in the alphabet. Functions in this space map from an input state to the next occurrence of any token in the set.

Appendix B

SMARTEDIT SCENARIOS

In this section we provide a brief description of each of the scenarios used to test the SMARTedit system. Although some of the scenarios are artificial, many are derived from actual repetitive tasks faced by SMARTedit users.

OKRA Given a web page containing a list of names, email addresses, scores, and the date the name was entered, select each of those items in turn for each name on the page. This information extraction task comes from the RISE repository.

addressbook Convert single-line addresses into a multi-line format, suitable for printing on a mailing label.

bibitem-newblock Convert a list of bibliographic entries from BibTeX intermediary format to human-readable form.

bindings For each call to the `bind()` function in a Python source code file, add a second function call immediately following the `bind` whose first argument is the first argument to the `bind`. Ensure that the inserted code preserves the indentation level on the previous line.

bold-xyz Boldface the name of a company everywhere it appears on the web page, using the HTML bold tags. The company name sometimes has a space between the two words in the name.

boldface-word Boldface the word SMARTedit in a LaTeX document.

c++comments Replace C++-style comments (`//`) with C-style comments (`/* */`), assuming one comment per line of text.

citation-creation Automatically construct a citation for each bibliography entry, of the form "[Hirsh, 1997]". It should be extracted from the first argument to the `\bibitem` command in each entry.

citation-to-bibtex Convert a list of citations in a normalized, human-readable format into BibTeX format.

column-reordering Rearrange the columns in a structured text file containing columns separated by spaces. The contents of the first column become the last column.

country-codes Extract (country, country code) tuples from an HTML page by converting the information in an HTML table to a list of comma-separated values. This is the example task for Kushmerick's wrapper induction system [38].

game-score Convert a list of sports games into a structured format that indicates the winners, losers, and scores. This example task comes from the TELS system [57].

grades For a list of grades and students, one per line, delete the name of the student so that only the grade remains.

html-comments Delete the HTML comments, including their contents, from a web page.

html-latex Convert HTML to LaTeX format by escaping less-than and greater-than characters into LaTeX's math mode.

indent-voyagers For plain text extracted from a web page, fix the indentation by removing the extraneous spaces at the beginning of every line, and make the text wrap by removing the newline at the end of each line of text.

latex-macro-swap Convert from LaTeX's `\tt` operator to a user-defined macro, by changing all occurrences of `{\tt text}` to `\prog{text}`.

mark-format In a Python script, change all occurrences of the statement outstate.selection_extent[?] to mark_split(outstate.selection_extent[?]), for any expression in place of the question mark.

modify-to-rgb-calls In a Python script, change all calls to the functions _to_rgb and _to_tk_rgb to pass the variable self as their first argument.

number-citations Number each citation in a bibliography sequentially using the format [i], placing the number on a line by itself at the beginning of each citation.

number-fruits Number consecutively all the lines in the file using the format "*i.* ".

number-iterations Number consecutively all the lines in the file that have a pound sign (#) at the beginning of the line.

outline For a file in Emacs outline format, number all the high-level section headers consectively starting from 1, and copy each numbered high-level section header to the top of the file.

pickle-array Convert an array of numbers from one format (a Python pickled format) to another.

prettify-paper-info Given a text file with doubly-spaced lines indented to appear in the middle of the page, remove the double spacing and the extraneous indentation such that each line is flush with the left margin.

smartedit-results Convert from the results generated by SMARTedit's scenario test harness (whitespace-separated columns) into a LaTeX table. Use the & character to separate columns and end each line with two backslash characters.

subtype-interaction In a Python script, make each defined class inherit from a given base class by adding the string "(Interaction)" before the first colon in a class definition.

ul-to-dl In an HTML file containing a list of links to online publications, change the <li> entries to <dt>/<dd> entries where the <dt> component is the set of links to the paper (including the link to the compressed version) and the <dd> component is the paper info (minus the links and with quotes around the title).

xml-comment-attribute Reformat XML text so that commented-out elements are replaced with the uncommented element with the addition of an extra attribute.

zipselect Given a file containing address data, one address per line, copy each zip code to the end of the file.

Appendix C

SMARTPYTHON VERSION SPACES

In this section, we summarize the version spaces used in the SMARTpython system, organized roughly in order of appearance starting from the target space StatementOrConditional.

StatementOrConditional The target version space. Each state change in the program results from either a Statement or a Conditional.

Statement A join of NextStep and an Assignment statement.

NextStep An atomic version space consisting of functions of the form $f(x) = C$; the constant C indicates the index of the next step to be executed after this one.

Assignment A union of two types of assignment statements: VarValue and ArrayValue. Each VarValue child is parameterized by a different variable name supported by the language. One ArrayValue child is instantiated for each possible index of the array. Each child version space represents a different left hand side of an assignment statement.

VarValue(i) A union of a number of ways to specify the right hand side of an assignment statement whose left hand side is variable i. Three components are transformations of ConstInt spaces, representing the three statements $i := C$, $i := i + C$, and $i := i * C$ respectively. The fourth component of the union is a Variable version space.

Variable An enumerated version space containing all possible variable names, including all scalar variables, array variables indexed by constants, and array variables indexed by a scalar variable.

ArrayVariable(j) A union of a number of ways to specify the right hand side of an assignment statement whose left hand side is the j^{th} member of the array. Three components of the union are transformations of ConstInt spaces, representing the three statements $A[j] := C$, $A[j] := A[j] + C$, and $A[j] := A[j] * C$ respectively. The fourth component of the union is a Variable version space.

Conditional A union of different forms of conditional expressions. The union includes one instantiation of Var(i) ¡ Const for each variable i, one instantiation of Var(i) ¿ Const for each variable i, and the Var ¡ Var version space. A variable in this context means a scalar, an array indexed by a constant, or an array indexed by a scalar.

Var(i) $<$ Const An atomic version space, parameterized by variable i, that contains hypotheses of the form $i < C$ for variable i and all real numbers C.

Var(i) $>$ Const An atomic version space, parameterized by variable i, that contains hypotheses of the form $i >= C$ for variable i and all real numbers C.

Var $<$ Var An atomic version space that contains hypotheses of the form $i < j$ for all variables i and j. A variable in this context means a scalar, an array indexed by a constant, or an array indexed by a scalar.

Var $>=$ Var An atomic version space that contains hypotheses of the form $i >= j$ for all variables i and j. A variable in this context means a scalar, an array indexed by a constant, or an array indexed by a scalar.

Var $=$ Var An atomic version space that contains hypotheses of the form $i = j$ for all variables i and j. A variable in this context means a scalar, an array indexed by a constant, or an array indexed by a scalar.

Appendix D

PROGRAMS USED TO EVALUATE SMARTpython

Bubble sort

```
i = 0
while i < 5:
        j = i
        j = j + 1
        while j < 5:
                if A[j] < A[i]:
                        k = A[i]
                        A[i] = A[j]
                        A[j] = k
                j = j + 1
        i = i + 1
```

Greatest common divisor

```
while j > 0:
        k = i % j
        i = j
        j = k
```

Fibonacci

```
i = 1
j = 1
A[0] = 0
while i < k:
        i = i + 1
        j = j + A[0]
        A[0] = j
```

Find-max

```
i = 0
k = 0
while i < 5:
        if A[i] >= A[k]:
```



```
                k = i
        i = i + 1
```

Find-min

```
k = A[0]
i = 1
while i < 5:
        if k > A[i]:
                k = A[i]
        i = i + 1
```

Array-sum

```
i = 5
j = 0
while i > 0:
        i = i - 1
        j = j + A[i]
```

Primality test

```
i = 2
A[0] = 1
while i < j:
        k = j % i
        if k <= 0:
                A[0] = 0
        i = i + 1
```

If-statement

```
i=3
if i < j:
        k = 5
else:
        k = 1
```

Array-rhs

```
i=0
while i < 5:
A[i] = A[i] + i
i = i + 1
```

Array test

```
i = 0
A[0] = 2
A[1] = 3
i = i + 1
```

Array test 2

```
i = 0
while i < 5:
A[i] = 2
i = i + 1
```

CSE 100 x

```
i = 10
i = 3+i
i = i*4
i = i-1
```

CSE 100 y

```
i = 10
if i>5:
if i>10:
i = 20
else:
i = 5
else:
i = 0
```

Simple

```
i=2
j=3
i=i+j
```

Squid

```
k = 0
for i in range(1, 4):
 for j in range(1, i+1):
    k = k + 1
```

While loop


```
i=0
j=0
while i < 5:
i = i + 1
j = j + i
```

VITA

Tessa Lau received a B.A. in Computer Science and a B.S. in Applied & Engineering Physics from Cornell University in 1995. She received a M.S. in Computer Science & Engineering from the University of Washington in 1997, and in 2001 she completed a Ph.D. from the University of Washington in Computer Science & Engineering.