Using Ensembles of Decision Trees to Automate Repetitive Tasks in Web Applications

Zachary Bray
Computer Laboratory
University of Cambridge
JJ Thomson Avenue, CB3 0FD, Cambridge, UK
zachary.bray@cantab.net

Per Ola Kristensson
Cavendish Laboratory
University of Cambridge
JJ Thomson Avenue, CB3 0HE, Cambridge, UK
pok21@cam.ac.uk

ABSTRACT
Web applications such as web-based email, spreadsheets and form filling applications have become ubiquitous. However, many of the tasks that users try to accomplish with such web applications are highly repetitive. In this paper we present the design of a system we have developed that learns and thereafter automates users' repetitive tasks in web applications. Our system infers users' intentions using an ensemble of decision trees. This enables it to handle branching, generalization and recurrent changes of relative and absolute positions. Our evaluation shows that our system converges to the correct solution after 3–8 iterations when the pattern is noise-free, and after 3–14 iterations for a noise level between 5–35%.

Author Keywords
End-user programming, programming by example

ACM Classification Keywords
I.2.2. Automatic Programming: Program synthesis. I.5.5. Implementation: Interactive systems.

General Terms
Algorithms, Design, Experimentation, Human Factors

INTRODUCTION
Many of the tasks we do in our day-to-day lives are repetitive in their nature. Greenberg [5] has conducted several empirical studies in which users perform a range of repetitive tasks, such as dialing phone numbers and searching for information in manuals. For users it is near impossible to avoid needless repetition in most software since most software is not specifically tailored to users' needs. Programming by example (PBE) [16] is a research field that tries to help users with avoiding repetition by designing intelligent agents that learn users' tasks [10].

In this paper we present a PBE system that uses ensembles of decision trees to infer and automate a variety of repetitive tasks in web applications. Our system works by observing users' actions and inferring the underlying repetitive patterns. Suggestions for automation are then proposed to the user. Our system can handle branching, generalization and recurrent changes of absolute and relative position changes. After a few iterations our system learns a variety of recurring sequences of user interface actions. We have implemented our system as an extension to the Firefox web browser. Our evaluation shows that ensembles of decision trees converge to the correct solution after 3–8 iterations when the pattern is noise-free and after 3–14 iterations when the noise level is between 5–35%.

EICS'10, June 19–23, 2010, Berlin, Germany.

SYSTEM
Our system was developed as a Firefox extension. Firefox is built on top of Mozilla, which uses XML User Interface Language (XUL) and JavaScript to control the user interface. XUL is used to specify how user interface components are laid out in the user interface, while JavaScript is used to control the behavior. Firefox follows the Document Object Model (DOM) Level 2 Specification [7]. DOM is used to specify both the Firefox user interface (via XUL), as well as the web pages (HTML, XHTML and XML documents). Thus the DOM provides a unified interface for our system to interact with.

An important aspect of a highly interactive system is latency. In our case, the inference system must produce inferences quickly enough to keep up with the user's interaction with the web browser. In practice, this means that the average latency for an inference needs to be less than 200 ms. We found that JavaScript was too slow to satisfy this requirement. We therefore wrote the inference system in Java and used LiveConnect [4] to bridge Java to Firefox's JavaScript interpreter. Our complete system consists of three sub-systems: the event system, the graphical user interface, and the inference system. We now describe each of these sub-systems in turn.

EVENT SYSTEM
The event system has two primary functions. The first is to capture DOM-events in the browser, enrich them, and then convert them into events that can be handled by the inference system. The second is to dispatch new events to the browser.

Firefox uses two DOM-documents: one for the browser and one for the current web page. The event system attaches listeners to the root node of both these DOM-documents. When a user interacts with an element of either DOM-document, the element fires an event that propagates upwards in the element tree until it eventually reaches the root node where it is caught by our listener. Then we compute an address vector by concatenating the indices of the child elements traversed on the path from the document root to the element where the event originated. Next we construct an event containing the DOM event type, the address vector, and the attribute values of the element. Then we send this event to the inference system.

GRAPHICAL USER INTERFACE

The GUI has three primary functions. The first is to enable users to configure aspects of the system. The second is to non-intrusively present the user with suggestions for automation (see top of web page in Figure 1). The third is to provide feedback to users about the next inferred user action. We provide this feedback by using green highlighting to notify the user where the system thinks the next event will take place (Figure 1). This technique has previously been used in EAGER [2].

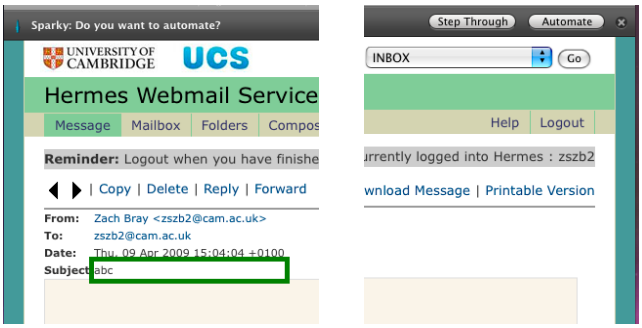

Figure 1. Our PBE system (Sparky) is non-intrusively suggesting automating the user's task (top). The next predicted user interface element is highlighted with a green rectangle (near bottom).

INFERENCE

The inference system uses ensembles of decision trees to perform inference on the events delivered by the event system.

Task and Input

The inference task is a supervised statistical classification problem: given a sequence of user-triggered events, how can we infer the user's future events?

Sequences of past events form training examples $\mathrm{E} = \{E_i\}$. Each training example $E = (\mathbf{i}, o)$ consists of an input event vector $\mathbf{i} = [e_1, e_2, \ldots, e_n]^T$ and an output event o. The input event vector consists of the sequence of n events that led to the output event o. The task is to learn a function f that maps a sequence of input events to an output event:

$$f : \mathbf{i} \rightarrow o \qquad (1)$$

In our system each event is a map of attributes onto values.

Decision Trees

A decision tree learning algorithm analyzes a set of examples $\{E_i\}$ to generalize patterns in data [13]. The examples are pairs containing an input event vector $\mathbf{i}$ and an output event o. Assuming the output event is dependent on the input events, there will be attribute values in the input events that imply certain output events. We use this information to construct a tree T that can infer output events from input events.

Entropy measures the uncertainty of a random variable. Here the random variable ranges over output events $\{o_i\}$ within a set of examples $\mathrm{E} = \{E_i\}$. The entropy $H(\mathrm{E})$ is:

$$H(\mathrm{E}) = \sum_{c \in \{o_i\}} -P_c \log_2 P_c \,, \qquad (2)$$

where P_c is the proportion of output events that are equal to output event c in the training set E.

Information gain measures the reduction in entropy of a random variable when there is knowledge about the current state. Here, the current state is a sequence of input events. The information gain $IG(\mathrm{E}, A)$ is:

$$IG(\mathrm{E}, A) = H(\mathrm{E}) - \sum_{v \in V(A)} \frac{|\mathrm{E}_v|}{|\mathrm{E}|} H(\mathrm{E}_v), \qquad (3)$$

where $V(A)$ is the set of distinct values that attribute A can take on, $\mathrm{E}_v \subseteq \mathrm{E}$ is the subset of training examples that have the value v for the attribute A, and $|\cdot|$ is the cardinality of a set.

The task of finding the optimal decision tree is NP-complete. We currently use a greedy algorithm, detailed in Quinlan's book [14]. Due to space constraints we only sketch the general idea here. The decision tree algorithm creates the tree T recursively. Each node in the tree contains a set of examples to classify. The algorithm tries to split the examples on each of the possible attributes. For each attribute split, it computes the information gain IG (Equation 3). The algorithm makes the final split of the examples using the attribute that maximizes the information gain. It then creates child nodes for each attribute value split. For each of these child nodes, it calls itself recursively and passes in the child node, the associated examples, and the remaining attributes. When the information gain from splitting on any of the remaining attributes is zero the recurrence terminates.

Improvements to C4.5

We made two changes to Quinlan's decision tree algorithm C4.5 [14], which we detail here.

First, we found that we could reduce the upper bound time complexity of the algorithm from $O(mn(\log n)^2)$ to $O(mn(\log n))$, where m is the number of attributes and n is the number of examples in the training set. For each split, the C4.5 algorithm needs to decide which is the best attribute to split on. For each attribute, the algorithm sorts the examples in the order of their attribute value, then splits, and thereafter works out the information gain. We eliminated the sorting step in the above process by instead having each attribute use a hash table of collections. The algorithm goes through each of the examples and uses a hash of their attribute value to assign them to a collection. This reduces the time complexity of finding the best split for a level in the tree from $O(mn \log n)$ to $O(mn)$.

Second, every time a user triggers an event in the browser, the system converts it into an output event that completes a new example. It uses this to build an updated decision tree which is only used to infer the next output event from the current input event vector. Hence, by lazily constructing the branches of the decision trees, we can reduce the upper bound time complexity further.

We found that these two relatively straightforward modifications dramatically increased classification speed.

Ensembles

It is important that the inference system can model a variety of tasks. To achieve this we use ensembles of decision trees. To our knowledge, this is the first use of ensembles of decision trees in PBE.

The utility of an ensemble is determined by two factors. The first is the coverage of the problem space provided by its individual classifiers. The second is the design of the decision function that determines which decision tree makes the final classification.

Coverage

After extensive experimentation we discovered that using a variety of decision trees, each with different sizes of input event vectors in the examples, leads to dramatically increased accuracy and faster convergence.

The system also limits the number of previous examples used to build decision trees. This improves the adaptability when users change their activity. A decision tree with a relatively small set of examples allows new input events to have more influence on how the decision tree will split.

Decision Function

The design of the decision function that chooses the decision tree is crucial for the success of an ensemble. Many different variations of combining decision trees have been explored in the literature (see Banfield et al. [1] for a recent overview).

We decided to choose decision trees according to the system's confidence in the correctness of their classifications. We designed a decaying confidence function that biases confidence towards recently correct trees. Again, this improves the adaptability of the inference system when users change their activity.

Each time a tree receives an input event it adds a Boolean value to a buffer that represents whether the previous prediction made by the tree was correct or not. This buffer has a fixed size, which means that new events push out the oldest events from the buffer when the buffer is at its capacity. The confidence of a decision tree $c(T)$ is computed using the following formula:

$$c(T) = \sum_i b_i \alpha^i \,, \tag{4}$$

where $(b_1, b_2, \ldots, b_n)$ is the buffer of Boolean values, $b_i = 1$ if the prediction made by T i predictions ago was correct, otherwise $b_i = 0$, and α is an empirically determined constant. In our implementation we set $\alpha = 0.6$.

Temporally Dispersed Tasks

Temporally dispersed tasks are tasks that occur infrequently, for example, doing a regional house search once every morning. Such repeated events are difficult to infer for a decision tree because it would need to keep a long history of events. Decision trees are not scalable to large sets of examples when integrated into a web browser. Our system works around this by complementing the ensemble of general decision trees with a collection of decision trees for each URL. Each input event has a URL attribute attached to it. The system uses the domain names extracted from these URLs as keys to find the domain-name-specific decision trees. It then uses these decision trees in conjunction with the general decision trees to determine the tree with the highest confidence.

Relative and Absolute Addressing

A key aspect of the decision tree is how it defines the equality of two output events. Equation 2 uses a set of output events to compute the entropy of a set of examples. To create a set of output events the system needs to determine which output events are equal. In our system two output events are equal if and only if they have exactly the same address vector, type, and extra information associated with the type.

Decision trees take each received input event and generate an output event that they use in conjunction with previous input events to form a training example. An output event has the following attributes: an address vector, a type, and any extra attributes associated with the type, such as the button of a click event. To generate an output event from an input event, the system copies the type and extra type attributes into a new output event. The address vector can be formed via either absolute or relative addressing.

Absolute Addressing

Using absolute addressing the system copies the address vector from the input event without modification to the output event. In this mode, the system is able to infer sequences of events as long as there is no recurring address pattern that needs to be generalized. It is unable to generalize address patterns because each of the addresses stored in the output events are unique, which means that similar changes in position are not classified under the same output event.

Relative Addressing

Using relative addressing the system does not copy the address vector from the input event to the output event. Instead it computes the vector difference between the two previous input events' address vectors and uses that as the address vector for the output event. This enables the system to model relative changes in addressing.

There are two situations in which the system needs to switch between absolute and relative addressing when performing inference.

Repeated Change of Position until Certain Element

The first situation is when a user performs a task where the system needs to infer a repeated change in element positions until it reaches a certain element. For example, a user of a music downloads website intends to teach our system to add all the singles by an artist that are currently on offer to their shopping basket. When they search for "Queen", the website presents them with a page containing a list of five singles by Queen, and a fixed button at the bottom of the page labelled "More". When the "More" button is clicked it dynamically replaces the list with the next five singles. To teach the system to add all singles that contain the word "offer" in the blurb to the shopping basket the user clicks on the blurb of each item in the list in order, and when the word "offer" is in the blurb they also click on the item's "add to basket" button. At the end of the list the user clicks on the "More" button and restarts the process from the start of the new list.

To model this task the system uses an ensemble of both absolutely and relatively addressed decision trees. Whether or not to use an absolutely or relatively addressed decision tree depends on the address of the last input event. The decision is made by computing an address-specific confidence based on recent events that were preceded by an event with a particular address. Each time a decision tree receives a new input event it uses the previous input event's address vector as a key to retrieve a Boolean buffer. It adds a Boolean value to this buffer that represents whether or not the previous prediction the decision tree made was correct.

When the decision tree needs to infer from an input event it then uses the input event's address vector as a key to get the corresponding Boolean buffer. It then uses an address-specific confidence function to find its confidence $c_a(T)$:

$$c_a(T) = \sum_i b_i \alpha^i \,, \tag{5}$$

where $\{b_i\}$ is the Boolean buffer and α has the same definition as in Equation 4.

The system then fuses the general confidence function $c(T)$ (Equation 4) with the address-specific confidence function $c_a(T)$ (Equation 5) via a linear combination, thus producing a combined confidence $c_c(T)$:

$$c_c(T) = \beta c_a(T) + (1-\beta)c(T) \,, \tag{6}$$

where $\beta \in [0.5, 1]$ is an empirically determined constant that controls the relative contribution of the address-specific confidence over the general confidence. In our implementation we set $\beta = 0.8$.

Intertwined Absolute and Relative Position Changes

The second situation where switching between absolute and relative addressing is needed occurs when the user is performing a task that involves intertwined relative and absolute position changes. Recall the previous music downloads website example. Imagine that clicking the "More" button appended items to the list rather than replacing them. The relative and absolute position changes then intertwine with each other and do not depend on the preceding event's address vector. Instead they depend on the address of the event two places back in the event sequence.

To solve this problem our system breaks the patterns into several sub-patterns consisting of absolute periods and relative periods. The system uses decision trees to model each of these sub-patterns and then joins them together. Each of the decision trees has an on-period and an off-period. They are fed new events and contribute to the inference during their on-periods, but are ignored during their off-periods.

For example, the pattern RRRAARRRAARRRAA (where R stands for relative position changes and A stands for absolute position changes) is broken down into RRR-- and AA--- (where dashes stand for off-periods). A relatively addressed decision tree with an on-cycle of three and an off-cycle of two can model the first pattern. Similarly, an absolutely addressed decision tree with an on-cycle of two and an off-cycle of three can model the second pattern. However, the second pattern occurs at an offset from the start of the initial pattern. The system models this by using decision trees for each combination of sub-pattern and offset. For example, the decision trees modeling the AA--- sub-pattern would include AA---, -AA--, --AA-, ---AA, and A---A. During the learning phase the two sub-patterns begin to learn their respective parts of the RRRAARRRAARRRAA sequence. The final task for the system is to choose the correct decision trees from all available sub-pattern decision trees at the right times. This

is done via a confidence function (Equation 4) adapted to handle the on- and off-periods.

EVALUATION

We evaluated our system on two test cases extracted from actual user activities. The test cases were chosen because of the high difficulty level for a PBE system to accurately infer the patterns behind them. These test cases were converted into test patterns. To test the system against varying levels of noise, we used a teacher simulation where the noise level could be calibrated. Each simulation performed a task a number of times. For each task the simulator would perform each step of the task one by one. In between each of the steps the simulator would inject a mistake if a uniformly distributed random variable fell within the noise threshold.

The results for each example consist of values averaged over 100 tests of 25 iterations each, where each run of 25 iterations began with no examples present in the ensemble.

Test Case 1: Web-based Spreadsheet

The first test case was a web-based spreadsheet application.[1] This test case assessed branching, generalization and recurrent address switching behavior.

Task

The user is filling the rows of a column on a spreadsheet with three alternative lines of text. The spreadsheet implementation is unusual in the way it requires the user to use a static text box at the top of the page to enter text into a cell. This means our system must successfully switch between absolute and relative modes of addressing.

Results

Figure 2 (left) shows mean accuracy as a function of the number of iterations, with no noise. As shown the system has linearly converged to the correct solution after the third example.

Figure 2 (right) shows the mean number of iterations required to converge to the correct solution as a function of the noise level. As can be seen, there is an exponential trend, and the number of iterations needed to reach the correct solution range from 3–14 for a noise level in the range 5–35%.

Test Case 2: Web-based Email

The second test case was a web-based email application.[2] This test case assessed if the system can infer branching behavior.

Task

The user is going through each of the messages stored in their account. The user is looking for a keyword in the subject of the message. If the user finds the keyword the user interacts with some of the on-screen components to move the message into another folder. Then the user continues by navigating to the next message and repeats.

Results

Figure 3 (left) shows mean accuracy as a function of the number of iterations, with no noise. The system is completely accurate after the eighth example. Obviously, the system cannot reach 100% accuracy unless it has seen all the branch outcomes. The curve in Figure 3 (left) is similar to a geometric probability distribution of the branch outcomes which suggests the system converges to the correct solution as soon as it has seen each of the outcomes.

Figure 3 (right) shows the mean number of iterations required to converge to the correct solution as a function of the noise level. Similar to figure 2 (right) it shows an exponential trend, and the number of iterations required range from 3–9 with a noise level in the range 5–35%.

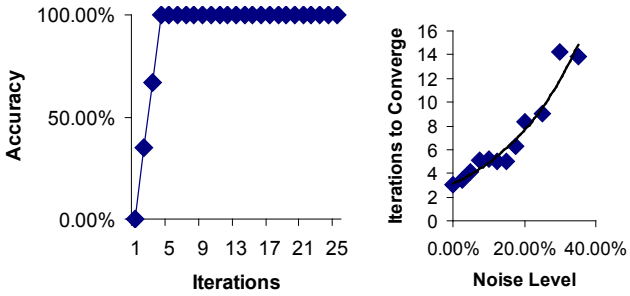

Figure 2. Results from the web-based spreadsheet test case. (Left) Mean accuracy as a function of the number of iterations. (Right) Mean number of iterations required to converge to the correct solution as a function of the noise level.

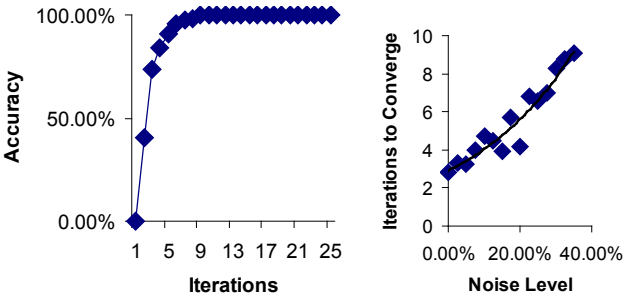

Figure 3. Results from the web-based email test case. (Left) Mean accuracy as a function of number of iterations. (Right) Mean number of iterations required to converge to the correct solution as a function of the noise level.

For both test cases, the time taken for the system to produce an inference ranged between 10–200 ms on a 2.1 GHz Mac.

RELATED WORK

One of the earliest PBE systems, possibly the first, was Teitelman's PILOT system [17, 6]. Some of the first modern systems were Metamouse [12] and EAGER [2]. The former automates drawings by asking the user a series of questions, while the latter automates loops for users.

[1] http://www.simple-groupware.de

[2] http://webmail.hermes.cam.ac.uk

Overviews of the general field of PBE can be found in the books *Watch What I Do* [3] and *Your Wish is My Command* [9].

A recent example of a system that automates web application tasks is CoScripter [8]. Similar to our system it is also implemented as a Firefox extension. However, unlike our system CoScripter does not attempt to infer a variety of user activities. Rather it is a macro recorder with an accompanying scripting environment that enables users to easily edit and share their scripts. In the context of PBE, Ruvini and Dony [15] used a custom concept learning algorithm to infer repetitive user actions in a Smalltalk programming environment. Mahmud and Lau [11] created CoTester which is designed to simplify web site testing. It automatically groups low-level web site actions, such as a mouse click, into high-level subroutines, such as "Add to cart". Unlike these previous approaches our inference system uses decision trees [13]. We improve upon this technique by using ensembles of decision trees. The use of ensembles has been proposed before in the machine learning literature (see Banfield et al. [1] for a recent survey). While previous approaches inspired aspects of our design, our particular solution with confidence functions is novel. By using ensembles of decision trees and taking into account absolute and relative position changes we have demonstrated how it is possible to infer a wide variety of user tasks.

CONCLUSIONS

We have presented a system that automates users' repetitive tasks in web applications. Our system is implemented as a Firefox extension and is capable of making predictions with low latency (less than 200 ms). Our system uses an ensemble of decision trees of varying lengths to be able to infer a wide variety of tasks. In comparison to previous PBE systems our system can infer complex tasks, such as tasks with a high branching factor, and tasks with intermixed relative and absolute position changes. Such tasks are common in many web applications and it is therefore important that the machine learning algorithm is capable of accurately modeling them. Our system is also tolerant to noise, which is essential for a PBE system to be effective in practice. We evaluated our system with two test cases that required our system to handle branching, generalization and recurrent address switching behavior. With no noise our system converged to the correct solution after 3–8 iterations depending on the branching factor of the task. With a noise level between 5–35% our system converged to the correct solution after 3–14 iterations.

ACKNOWLEDGEMENTS

We thank Carl Scheffler, Philip Sterne and Keith Vertanen for their assistance. The following applies to P.O.K. only: The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007 2013 under grant agreement number 220793.

REFERENCES

1. Banfield, R.E., Hall, L.O., Bowyer, K.W. and Kegelmeyer, W.P. A comparison of decision tree ensemble creation techniques, *IEEE T. Pattern Anal. 29,* 1 (January 2007), 173–180.
2. Cypher, A. EAGER: programming repetitive tasks by example, in *Proceedings of CHI '91* (New Orleans LA, April-May 1991), ACM Press, 33–39.
3. Cypher, A. (ed.). *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge MA, 1993.
4. Flanagan, D. *JavaScript: The Definite Guide, Fifth Edition*, O'Reilly Media, Sebastopol CA, 2006.
5. Greenberg, S. *The Computer as Toolsmith: the Use, Reuse, and Organization of Computer-Based Tools*, Cambridge University Press, Cambridge UK, 1993.
6. Kay, A. Foreword. In Cypher, A. (ed.), *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge MA, 1993.
7. Le Hors, A., Le Hégaret, P., Wood, L., Nicol, G., Robie, J., Champion, M. and Byrne, S. *Document Object Model (DOM) Level 2 Core Specification*, W3C, 2000.
8. Leshed, G., Haber, E.M., Matthews, T. and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise, in *Proceedings of CHI '08* (Florence IT, April 2008), ACM Press, 1719–1728.
9. Lieberman, H. (ed.). *Your Wish is My Command: Programming by Example*, Morgan Kaufmann, San Francisco CA, 2001.
10. Maes, P. Agents that reduce work and information overload, *Commun. ACM 37,* 7 (July 1994), 30–40.
11. Mahmud, J. and Lau, T. Lowering the barriers to website testing with CoTester, in *Proceedings of IUI '10* (Hong Kong CN, February 2010), ACM Press, 169–178.
12. Maulsby, D. and Witten, I. Inducing programs in a direct-manipulation environment, in *Proceedings of CHI '89* (Austin TX, April-May 1989), ACM Press, 57–62.
13. Quinlan, J.R. Induction of decision trees, *Mach. Learn. 1* (January 1986), 81–106.
14. Quinlan, J.R. *C4.5: Programs for Machine Learning*, Morgan Kaufmann, San Mateo CA, 1993.
15. Ruvini, J.-D. and Dony, C. APE: learning user's habits to automate repetitive tasks, in *Proceedings of IUI '00* (New Orleans LA, February 2000), ACM Press, 229–232.
16. Smith, D.C., Cypher, A. and Tesler, L. Programming by example: novice programming comes of age. *Commun. ACM 43*, 3 (March 2000), 75–81.
17. Teitelman, W. Toward a programming laboratory, in *Proceedings of IJCAI '69* (Washington DC, May 1969), Morgan Kaufmann, 1–8.