Computation Scrapbooks

Richard Potter
PRESTO, Japan Science and Technology Corporation (JST)
potter@is.s.u-tokyo.ac.jp

Abstract
An important goal for end-user programming is to make gentle-slope systems that are
easy for novice users yet scale gracefully to general purpose programming. However,
understanding computation state introduces a difficult steep slope for non-
professional programmers because it is invisible and changing. I propose Computation
scrapbooks to collect, study, organize, and reuse multiple snapshots from the
computation state of general purpose programming languages. They provide new
opportunities for writing, testing, documenting, and understanding computer
programs. I am currently creating a simplified prototype system in Emacs Lisp to
explore high-level architectural issues, paving the way for a robust Java VM
implementation.

Introduction
As computers become more integrated into people's lives, computer skills of all levels
will take on more importance. The objective of this research is to give users
techniques that allow them acquire a full range of programming skills in an
incremental but continually useful way. These techniques will insure that small
increases in a user's programming skill will produce corresponding increases in
benefit. The current research term for systems that have this property is "gentle-slope"
systems, meaning that the learning curve for using the system does not have any steep
learning curves.

Figure 1: The goal of gentle-slope systems is gentle progress from novice to expert through
incremental learning.
This gentle-slope idea has played a role in the success of end-user programming systems like spreadsheets. A novice user can create real programs on a spreadsheet with a small amount of instruction. By learning incrementally by how coworkers use spreadsheets, the user can over time become an expert.

In contrast, general purpose programming languages are not gentle-slope systems. For example, a user may learn how to enter, compile, and run "hello world" in C, but it is not useful. This is a problem for web page designers and others who at times must write programs that are not possible with spreadsheets and other domain-specific easy-to-use programming techniques. Non-professional programmers who cannot take time to learn complicated programming knowledge will be unable to take full advantage of the opportunities provide by the computers in their lives.

There are many reasons why current programming systems are not gentle-slope systems. The one on which I wish to focus is the increase in temporary results as that as programs become more complex. The learning curve is made steeper because of the temporary and ever-changing nature of these intermediate results. Yet for users who wish to continually increase their programming skill, this understanding is essential because temporary values such as parameters, local storage, and return values are used in almost every programming language. They are essential for modular design.

Computation Scrapbooks are a new idea that can help users gradually master programming techniques that use temporary values. Computation Scrapbooks allow the users to collect, study, organize, and reuse computation states of their programs. Today such experience is impossible or at best prohibitively awkward to collect.

Computation Scrapbooks are exciting because they will offer a new foundation from which to build programming tools for all programmers, novice and expert alike. Experimental techniques in automatic programming and programming by demonstration will be possible on a smaller scale where they are more likely to be practical. Also, Computation Scrapbooks will be useful for improving program reliability, since they can be used to collect and reapply meaningful test cases.

**Alternative Approaches**

For many years, researchers have been trying to develop techniques that allow more types of users to write programs and make fuller use of computers. This area of research is called end-user programming. Many users who do not have the time to learn traditional programming skills could benefit from the ability to write programs that automate repetitive tasks, customize their work environment, or run "what-if" simulations to help in making decisions. Children can receive educational benefits by creating programs that simulate physical or biological systems they are studying.

Most of the research in end-user programming has focused on making programming simpler and more concrete. One technique for doing this has been programming by demonstration. Programming by demonstration allows a user to specify what a program should do by giving a specific example demonstration. The programming by demonstration system then automatically generates a program based on the user's demonstration. Programming in this way is easier because the user can give the demonstration concretely using typical applications like word processors or e-mail
programs. The user does not have to know the complex details of a programming language or use abstractions.

In some ways programming by demonstration has been successful, because it has allowed users with no programming experience to create useful programs. However, programming by demonstration does not scale beyond simple programs. Other end-user programming techniques have similar problems. Domain-specific languages can make programming simpler and more concrete, but they provide little benefit beyond the specific domains. For users who want to eventually advance to more powerful programming, these techniques are limiting and sometimes even a harmful distraction because they do not scale.

Researchers are now wrestling with the dilemma of how to make easy to use programming systems that also scale to complex and general programming. The research consensus is that, for the foreseeable future, users will have to acquire at least some traditional programming skills. Therefore the research focus has changed from eliminating the need for programming skills to providing effective ways to acquire them. The gentle-slope system philosophy is that people can acquire relatively complex skills if the short term benefits continually motivate additional learning.

**Computation Scrapbook Approach**

The high-level motivation for Computation Scrapbooks is that as users incrementally gain programming skills, they must understand more about computation state. Yet most of it is hidden. By analogy, computation state is like an iceberg because only a small part can be seen. The user must use their partial knowledge to imagine the hidden parts, which for a beginning user are a complete mystery. There is no effective direct solution to this problem because computation states are extremely complicated and do not have effective visualizations that are general purpose.

![User Interface](image)

**Figure 2**: Computation state is like an iceberg because most of it is hidden.

However, indirect solutions show promise. Even though computation state is mostly hidden and very complex, it is actually relatively easy to copy. It is *technically* becoming easier because of the use of virtual machines, e.g. Java, and inexpensive
storage. It is also conceptually simple because everything is copied. Even though the user may not understand all they need to understand about computation state, they can understand that it can be copied. While this alone does not directly make computation state understandable, it is a gentle step that leads to other gentle steps. Computation scrapbooks provide the tools to combine these steps into effective gentle-slope uses for computation snapshots.

One simple use is to annotate source code with snapshots. For example, a intermediate programmer could be reading through the Java SDK example code as in the figure below. The code could be annotated with complete computation states that demonstrate how particular lines of code are used. This would benefit the user because they could experiment with code without the added burden of setting breakpoints and determining what initial conditions and events are necessary to reach the particular line of code. The example snapshots could be specially chosen by a more experienced programmer to be simple enough to be understandable, but complex enough to show why a particular line of code is necessary.

```java
public void paint(Graphics g) {
    int np = lines.size();
    /* draw the current lines */
    g.setColor(getForeground());
    g.setPaintMode();
    for (int i=0; i < np; i++) {
        Rectangle r = (Rectangle) lines.elementAt(i);
        g.setColor((Color) color.elementAt(i));
        if (p.width != -1) {
            g.drawWire(p.x, p.y, p.width, p.height);
        } else {
            g.drawWire(p.x, p.y, p.x, p.y);
        }
    }
}
```

(Java SDK example code)

Figure 3: Source code linking to snapshots for detailed documentation of actual runtime use.

A second use of Computation Scrapbooks would be to organize lightweight test cases for arbitrary code segments. For a user just learning how to write procedures, learning effective ways to test newly written code is an additional burden that makes the learning curve less gentle. Each test case could consist of two snapshots of the computation state. Users could create these snapshots by directly editing computation states. This technique is more flexible than writing test routines, because it can be applied to arbitrary code segments and is not limited to complete procedures. Since computation states can be saved, the test cases could also be saved to test the code again if it is modified in the future. The test cases could also help document the code.
Figure 4: Pairs of snapshots can define test cases for arbitrary code segments.

Other creative uses of Computation Scrapbooks are possible. Programming by demonstration techniques could be used to generate code by treating editing operations on a computation state as a demonstration. This could make the slope much more gentle for users who have used programming by demonstration to create simple programs. Computation Scrapbooks also could allow techniques like automatic programming and genetic algorithms to find more practical applications since they could be applied to arbitrary code segments.

Work in Progress
Currently I am implementing a prototype Computation Scrapbook system in Emacs Lisp. The goal of this effort is to explore the high-level architectural issues. While the idea of saving complete computation states is the core technique, each way of using computation snapshots will have its own set techniques and issues. For example, using snapshots for documentation will require persistent ways to link from source code to where the snapshot is stored. Another issue is providing focus to certain parts of the snapshot so that visualization techniques will be more effective. Other uses, such as test cases and programming by demonstration, will bring their own set of issues.

Therefore, the goal of the Emacs Lisp prototype is to understand what core services the core snapshot mechanism has to provide to be effective for the various uses. Emacs is a suitable platform for such rough fast prototyping because it is a very-stable widely-used Lisp environment. It is also a rich environment with powerful functionality such that even small programs can be interesting and useful. Therefore, even a simple Emacs system has a good chance of being used by a sizeable user community and will likely generate interesting example to inform the design of Computation Scrapbooks.

A drawback to the Emacs Lisp environment is that Lisp provides limited access to the computation state. Therefore snapshots will be subsets. This is possible to overcome by creating a custom version of Emacs, but this solution would limit how easily the Computation Scrapbook system could be distributed. Another drawback is that Emacs provides limited GUI tools, so the Computation Scrapbook uses that benefit
from visualizations cannot be fully explored. Therefore, the once sufficient lessons from the Emacs prototype are gained, they will be used to design custom Java virtual machine that has robust snapshot capability and a full graphical user interface.

The current system can save and restore snapshots of the execution stack taken from inside a customized version Emacs' source debugger edebug. The standard version of edebug works by instrumenting functions so that all s-expressions call entry and exit hooks before and after evaluation. This has made it possible to implement the snapshot mechanism by customizing edebug to remember values on the execution stack. This technique works even if values are otherwise hidden by dynamic scoping. In order to support convenient browsing of restored snapshots, a backward tracing mechanism has been added to edebug. It works by simply restarting the execution from an initial expression and setting a global breakpoint that breaks at the previous time stamp.

![Emacs Lisp snapshot](image)

Figure 5: An Emacs Lisp snapshot with code state in the top window and execution stack below along with a time stamp. Current system allows saving and restoring of such states.

Currently, the prototype is being extended so that snapshots can be swapped out easily enough to support flexible test cases and programming by demonstration. Also, a flexible mechanism for specifying what information beyond the execution stack to include in the snapshot is being added. It is currently being investigated whether it is possible to add keyboard event histories to the snapshots, which would allow the replay and testing of interactive program segments.