In Proceedings of the 2003 IEEE Symposia on Human Centric Computing Languages and Environments (HCC03), Auckland, New Zealand, October 28-31, pp. 50-52.

Additional Context for Gentle-Slope Systems

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

Yasunori Harada
NTT Communication Science Laboratories, NTT Corporation
hara@acm.org

Abstract

Interpreting general purpose programming constructs can be difficult because it requires context, such as knowledge of language syntax or idioms, which users may not have readily available. In such cases other source of context may complement or substitute. This paper proposes annotating program comments with hyperlinks that the user can select to restore the complete runtime state of the program, a technique we call Snapshot Documentation. The generality of Snapshot Documentation across user skill and system complexity suggests that it may be useful for gentle-slope systems, i.e. systems that are designed to allow users to acquire a full range of programming skills in an incremental yet continually useful way.

1. Introduction

Saving and restoring runtime computation state has found previous uses in fault tolerant computing, continuations of interactive sessions, process migration, and debugging. This research is part of work to explore the potential for saving and restoring computation state to make programming easier. We have been exploring these possibilities with SBDebug, a Computation Scrapbook [4] system that allows multiple runtime states of Emacs Lisp to be saved, organized, and restored. SBDebug provides an infrastructure for exploring new programming tools. This paper will explain one of these tools, Snapshot Documentation, and its potential for gentle-slope systems.

2. Snapshot Documentation

The basic idea of Snapshot Documentation is that runtime state can be a helpful source of context for understanding computer programs. Snapshot Documentation shows potential for novice, intermediate, and expert users.

For an intermediate example, assume a user is using a text editor that has a feature to left justify a single column of monospaced text. The user finds it useful, but sometimes wishes the editor had a similar feature to right justify a single column of monospaced text. The user is considering the possibility of implementing this feature by programming. The user must first understand how the existing code works, perhaps by looking at various lines that look something like this:

```
(insert (make-string shift ? ))
```

Clearly context is necessary because code like this is cryptic in isolation. Many different types of context can help. Knowledge of English could hint that something is being shifted or inserted. Knowledge of Emacs Lisp itself is helpful and could tell the user that "shift" is an integer variable and that the line inserts "shift" number of spaces after the Emacs text cursor. Knowledge of the overall purpose of the program might suggest that this is shifting a column of text. Surrounding code (Figure 1) can provide context to reason about how this line works with other lines to left justify and maybe give clues of how to change the code so that it right justifies. A full listing of the program is about 150 lines, and would definitely be enough context, though it may require more skill or time than the user can muster.

```
(defun straighten-column ()
  (interactive)
  (let ((record-list (collect-record-around-point)))
    (if (null record-list) (error "No column found at point"))
    (let* ((line (car record-list))
           (scan (cdr record-list))
           (max-left-possible-position (aref line 1))
           (min-current-begin (aref line 2))
           (min-current-end (aref line 4))
           (max-length (- (aref line 3) (aref line 2))))
      (while scan
          (setq line (car scan))
          (setq max-left-possible-position (max max-left-possible-position (aref line 1)))
          (setq min-current-begin (min min-current-begin (aref line 2)))
          (setq min-current-end (min min-current-end (aref line 4)))
          (setq max-length (max max-length (- (aref line 3) (aref line 2))))
          (setq scan (cdr scan)))
      (let* ((target-left (max min-current-begin max-left-possible-position))
             (target-right (+ target-left max-length))
             (pad (- target-right min-current-end -2)))
          (save-excursion
            (while record-list
              (let* ((record (car record-list))
                     (bol (aref record 0))
                     (left-gap-end (aref record 2))
                     (shift (- target-left left-gap-end))
                     (pad-right (- pad shift)))
                (goto-char (aref record 0))
                (forward-char (aref record 3))
                (if (looking-at " *$") ;; rest of line is blank
                    (progn
                      (delete-region (point) (progn (end-of-line) (point)))
                      (insert "  "))
                  (if (< pad-right 0)
                      (delete-char (- pad-right))  ;; e.g.  <S-209-953>
                    (insert (make-string pad-right ? ))))  ;; e.g.  <S-208-1258>
                (goto-char (aref record 0))
                (forward-char (aref record 1))
                (if (< shift 0)
                    (delete-char (- shift))  ;; e.g. <S-211-1807>
                  (insert (make-string shift ? ))))   ;; e.g. <S-210-1417>
              (setq record-list (cdr record-list))))))))
```

Figure 1. With the proper skills, source code can provide useful context.

Snapshot Documentation provides context that can complement these and other sources of context. The basic idea is that the person who originally wrote the code

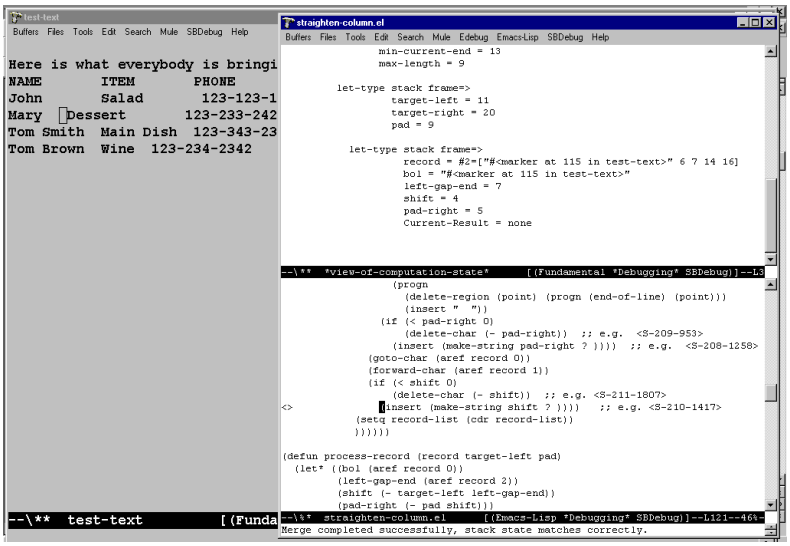

Figure 2. Selecting a Snapshot Documentation link in SBDebug restores a debugging session where the user can explore connections between the concrete runtime context and the abstract code.

can create snapshots of interesting debugging sessions while developing the code. After saving the snapshot in a Computation Scrapbook, the programmer can insert a link to the snapshot into a source code comment. Then people who are reading the code later can access the debugging session by simply selecting the link.

The code in Figure 1 shows how Snapshot Documentation appears for SBDebug, our prototype Emacs Lisp Computation Scrapbook. Links appear as short textual codes that can easily be inserted into source code comments. The `(insert ...)` line has the link `<S-210-1470>`, and if the user selects it, the system jumps directly to the debugger configuration in Figure 2.

Now the user has various pieces of information for reasoning about and verifying understanding of the program. The user interface of the program is in the left and already has simple (but not too simple) data. From this the user can see that the cursor is before the letter "D" in "Dessert". In the upper right area, the user can see that the present value of "shift" is 4. The lower right area shows the source code, and since SBDebug restored the complete execution state, the user can step through to the end of the line and watch the word "Dessert" become left-aligned with "Salad".

One might argue that the user could get the same context by running the program in a debugger from scratch. However, Snapshot Documentation has the advantage that the original programmer's expertise is preserved by the selection of the example data and break points. The programmer can make sure that the example data is simple enough to make stepping through the program practical, yet complicated enough to be interesting. For example, some of the lines of code in Figure 1 are not executed if the data is not selected carefully. Another advantage is that selecting a link is much easier than setting up a debugger, generating example data, setting break points, and running a program from scratch. Snapshot Documentation therefore makes the extra context available to users who would otherwise have too little skill or time to obtain it, exactly the people who might benefit the most.

Snapshot Documentation compliments and sometimes can substitute for other types of context, which could be valuable for novice users whose knowledge of the programming system is incomplete. It is full of detail that complements the abstract code. This gives novice users a better chance of connecting with something that they already understand. They can then begin to explore less familiar aspects. For example, even though the Lisp program processes columns of text, a novice programmer would have difficulty recognizing how they are expressed in the source code. In contrast, a novice could notice the columns of text easily in the debugger configuration and could then start to explore how they change as the program is executed step by step.

The context from Snapshot Documentation also makes it easier for the user to experiment. For example, a user might not know the `(make-string )` function or the "?" syntax, but tracing through the code on a specific example makes it clear that the code generates a string of spaces. If the user conjectures that the string was a string of spaces because a space character follows the "?", the user can change it to "`?!`" or some other character, click on the snapshot and run the code on the exact same data to confirm the conjecture. This experimentation is easy because the Snapshot Documentation feature of SBDebug can do code substitution that makes it possible to use source code that is different from that used when the

snapshot was made.

3. Simple and Complex Examples

The basic idea of Snapshot Documentation should generalize to other type of programming systems. In fact, in some sense this is what spreadsheets already do and could provide one alternative explanation for the success of spreadsheets, because spreadsheets always have an implicit snapshot of the runtime state, at least for simple spreadsheets. The concrete values provide documentation to explain the abstract formulas in the spreadsheet. The idea might be extended to more complex spreadsheets by allowing complicated cells with conditionals to contain links to complete spreadsheets that illustrate when various cases of the complex conditional are required. Procedural macros in spreadsheets could use Snapshot Documentation similar to how it is used in SBDebug with Emacs Lisp, i.e. taking the user quickly from a line of code to a debugger scenario showing the line of code processing actual data.

Another simple application of Snapshot Documentation for novice programmers would be to document specific language or application features. This could be similar to the "What is this?" feature that many applications have that allows a user to point at a menu item or tool bar and receive a help window that describes what the feature does. Snapshot Documentation could add a similar feature that answers "When would somebody use this?" For example, the user could select the feature and point to the summation tool icon in a spreadsheet application. A new spreadsheet would then open up that has a column of numbers set up for using the summation tool along with annotations that explain the use.

Snapshot Documentation also generalizes to complex programs, because virtual machines make it practical to take snapshots of entire operating systems. We are currently developing a Computation Snapshot system for Linux, which will make it possible to create Snapshot Documentation for almost any programming language that runs inside of Linux. Such snapshots can be small enough to be practical. For example, with our current system it is possible to save a snapshot of two debuggers in a GUI demonstrating the Unix sockets API with a snapshot that is smaller than 1MB, because the snapshot can be compressed in relation to a snapshot of a freshly booted virtual Linux.

4. Gentle-Slope Systems

Snapshot Documentation seems particularly suitable for gentle-slope programming systems [1] that are intended to be used by novice or intermediate programmers, yet still scale to general purpose programming. Users could be introducing to it when working with simplified or domain specific programming constructs and continue to use it as they advance to more advanced programming techniques.

Snapshot Documentation might conceivably help remove some of the "cliffs" in the learning curve. For example, a novice user trying to figure out what the `(insert …)` line does only looking at Figure 1 may feel like they must learn a lot about Emacs Lisp first. Learning so much might seem impractical and convince that they have hit a cliff and have no practical alternative but to give up. However, Snapshot Documentation can give even a novice user an alternative context to reason about what the line does.

Even though additional concrete context is being used, the same general purpose programming constructs are still the focus of the user's attention. Therefore, while the user is reaching conclusions and thus achieving short term benefits, they may also be gaining insights into some of the general purpose behavior of the language constructs. Instead of hitting a "cliff" and making no progress in learning general purpose programming, the chance for progress of long-term significance is increased.

5. Future Directions

Some systems, such as Squeak [2], can regenerate runtime state along with tools to inspect it. These might easily be extended to create Snapshot Documentation features.

One consequence of Snapshot Documentation is that it greatly increases the value of software visualization, algorithm animation, and advanced debugging techniques [3]. An area of future research is to explore ways that the benefits of this symbiotic relationship can be maximized.

6. References

[1] M. Dertouzos, "Creating the People's Computer", *Technology Review*, MIT, Cambridge, MA, 1997, pp. 20-28.

[2] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself", in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Atlanta, GA, 1997, pp. 318-326.

[3] H. Lieberman, and C. Fry "Bridging the Gulf between Code and Behavior in Programming", in *Proceedings of the SIGCHI conference on Human factors in computing systems*, Denver CO, 1995, pp. 480-486.

[4] R. Potter, and M. Hagiya, "Computation Scrapbooks for Software Evolution", in *Proceedings of the Fifth International Workshop on Principles of Software Evolution*, Orlando, FL, 2002, pp. 143-147.