Advanced Control Flows for Flexible Graphical User Interfaces*

or, *Growing GUIs on Trees*

or, *Bookmarking GUIs*

Paul T. Graunke
Northeastern University
Boston, MA, USA
ptg@ccs.neu.edu

Shriram Krishnamurthi
Brown University
Providence, RI, USA
sk@cs.brown.edu

ABSTRACT

Web and GUI programs represent two extremely common and popular modes of human-computer interaction. Many GUI programs share the Web's notion of *browsing* through data- and decision-trees. This paper compares the user's browsing power in the two cases and illustrates that many GUI programs fall short of the Web's power to clone windows and bookmark applications. It identifies a key implementation problem that GUI programs must overcome to provide this power. It then describes a theoretically well-founded programming pattern, which we have automated, that endows GUI programs with these capabilities. The paper provides concrete examples of the transformation in action.

1. INTRODUCTION

Consider software that installs modern desktop applications such as Microsoft Office or Adobe Acrobat. The installer permits users to customize their installed components, choosing features and adjusting for the available disk space and other resource constraints. Specifically, these installers allow users to "browse" back and forth between options before they commit to a final configuration, in a manner reminiscent of Web browsers.

This simple interface forces us to ask several questions: What is the relationship between this browsing and that in a Web browser? Does the browsing power of these installers compare to that of Web browsers? If not, are there beneficial features we can arbitrage between them? Specifically, can we "bookmark" a GUI? And how do we methodically construct these programs? This paper answers all these questions.

In the past few decades, interactive programs have greatly increased in complexity. Every decade appears to bring a whole new level of interface sophistication including full-screen ASCII-based graphics, pixel-based graphics, and the Web. While each improvement generally makes the quality of interaction better for the user, it also makes the programming task considerably more difficult. Much of the rigorous exposition on software design is still written for old-fashioned, inflexible textual interfaces [22].

Web software illustrates the problems that programmers must confront. Even though most Web programs avoid the complexity of having to employ large and intricate graphical libraries, they must still contend with a myriad of user actions. In addition to linear traversals of Web sites, users commonly use Back buttons, clone windows, create and visit bookmarks, and so forth. Every one of these actions imposes new burdens on the developer.

Utilities such as wizards only minimally assist interactive software developers. Most of these programs largely deal with the tedious task of laying out and instantiating visual elements. They still leave the programmer to deal with the difficult problems of designing and composing the actual behaviors that lie beneath the "callbacks". Unfortunately, most of the difficult tasks lie in the *interaction* between the visual elements and the underlying behavior. As a result, the wizards do not address many of the truly hard problems that arise in writing GUI programs.

The growing volume of interactive software makes these problems more urgent for software engineers and programmers. In particular, programmers need better tool support to assist in interactive software construction. As Myers says [22],

> Tools influence the kinds of user interfaces that can be created. Successful tools use this to their advantage, leading implementers towards doing the right things, and away from doing the wrong things.

In this paper, we offer three contributions to the conceptual and development toolkits of GUI programmers. First, we provide a simple yet fruitful analysis of the fundamental flows of control (and data) in modern interactive programs. We apply this to Web and to GUI programs to demonstrate that the former can often be much more complex than the latter. Specifically, we show that by cloning windows and bookmarking them, users can employ very powerful interaction patterns lacking in GUIs. Second, we present a programming pattern that helps developers provide these same features in GUIs, thereby enhancing the interaction facilities available to users. Third, we demonstrate that we can automatically transform programs written without these capabilities into ones that do provide them. Our prototype implementation allowed us to experiment with the new interaction styles.

The rest of this paper is organized as follows. Section 2 identifies similarities and differences in the browsing patterns engendered by Web and GUI applications. It also summarizes the features of the former that would be beneficial to the latter and outlines the difficulty in implementing the desirable GUI features. Section 3 presents the heart of our result: a programming pattern that overcomes this weakness. It then discusses the pattern's automation and correctness. Finally, it highlights additional sources of complexity that our pattern handles. Section 4 describes two paths of information flow through programs and how the pattern accommodates both. The remaining sections discuss related work and conclude with directions for future work.

2. USAGE PATTERNS IN INTERACTIVE PROGRAMS

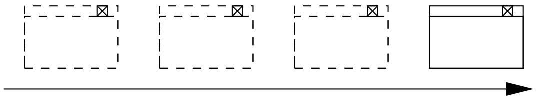
In this paper, we are primarily interested in *large-grained interactive* software, meaning programs that

- present results to the user and seek additional inputs at several points during their execution;
- encourage a user- and event-driven model of computation rather than one where the agenda is dictated by the application;
- maintain a relatively large granularity of interaction.

A software installer, for instance, maintains an extremely large interaction granularity (if we ignore individual character inputs, which libraries handle anyway); the granularity of a word-processor is extremely small (since its raison d'être is to handle character input). While, in theory, our results apply equally well to small-grained interactions, we have not applied them to such software, so we do not know how they scale to that level.

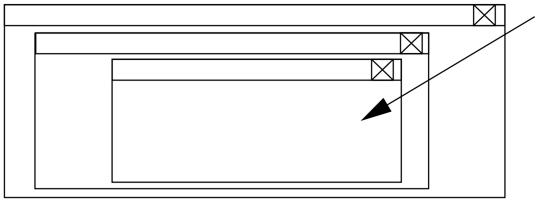
GUIs and Web-based programs are two of the most common and popular forms of interactive interfaces. To better understand interactive programs, we first describe typical user interactions with GUIs, then contrast these against Web interactions. We study user interactions patterns by examining the "shapes" of control flow that they engender.

At its simplest, a series of modal GUI panes sequentially consume user information:

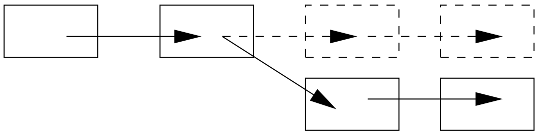


In this figure, each □ represents a single GUI window. The arrow represents the passage of time. The dashed boxes represent windows that no longer exist on screen, because the user has already entered data and selected options. This perversely inflexible GUI program is little different from a rigid, old-fashioned textual program, where the software pre-determines a sequence of forms and the user has to provide the appropriate responses.

Most modern GUI programs, however, give their users the choice of closing a window to return to the previous one and make a different selection:

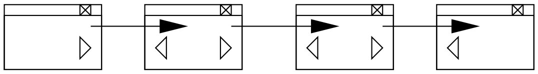


Here, time proceeds from the presentation of the largest window to the presentation of the smallest one; the ⊠'es allow users to kill a modal window and alter their choices on a prior window. We capture this interaction through a more abstract diagrammatic representation we shall call an *interaction diagram*, which depicts the entire history of the interaction:



Each box stands for a window presented to the user. The dashed region indicates that the user chose, then killed, one sequence of options, and has currently chosen a (possibly) different sequence, so the corresponding earlier windows are no longer visible on screen.

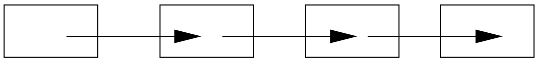
In some modern applications (including many installers), the GUI window includes explicit Forward and Back buttons. This gives the user the equivalent power of altering their decisions. While this has the benefit of offering a single window rather than crowding the screen with multiple ones, users need to use Back to see the options available and choices they made on previous versions of the window:



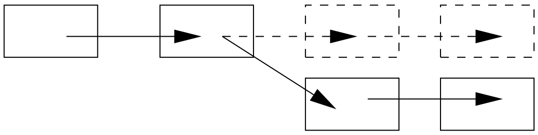
The resulting interaction diagram is the *same* as before (though earlier windows are effectively hidden beneath the Back button); the interface only provides a more convenient way of generating it, eliminating the need to pop up and kill numerous windows to explore different possibilities. In the end, while the user can explore a tree of choices, he can do this only one branch at a time.

In contrast to GUI programming, Web programming, at its simplest, is a fairly straight-forward task. A specification like the Common Gateway Interface (CGI) [23] simply requires Web programs to consume a set of name-value bindings and respond with a document of the appropriate type (which, currently, is often in HTML). This description, however, proves to be excessively simplistic for almost all interesting Web applications, which have grown up from being "scripts" to programs.

Many of the user interactions with Web programs are similar to those with GUIs. Studying the shape of these interactions helps us abstract away details of libraries and protocols. At its simplest, the user simply clicks through a series of Web pages, resulting in this interaction diagram:

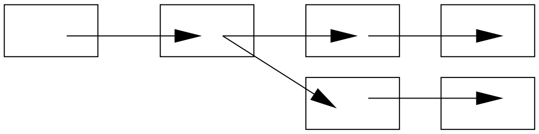


The difficulty arises when a user, unhappy with a set of choices, clicks the browser's Back button to return to an earlier page and make a different choice. This effectively wipes out one thread of interactions and creates a new one. In this respect the browsing history is a stack, which is represented explicitly by the Back and Forward buttons of many Web browsers, again capturing the fact that a user might have explored, but since abandoned, some sequence of options:



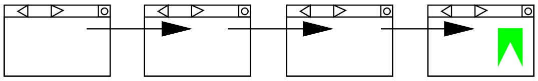
So far, Web programs seem to merely mimic the capabilities of GUIs. The Web, however, allows users to depart from, and enhance, these interaction styles in two important ways:

- First, users can clone the current window. The clones share a common past but, because they can now browse independently of one another, have different futures. Thus, cloning makes the tree structure of browsing explicit:

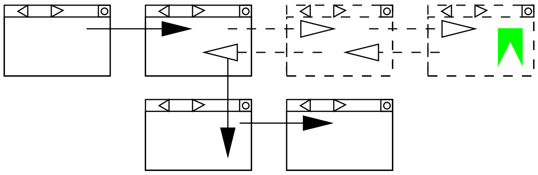


Importantly, each branch of the clone has solid lines and boxes, to indicate that both can run concurrently. Henceforth, we refer to this as *parallel exploration*.

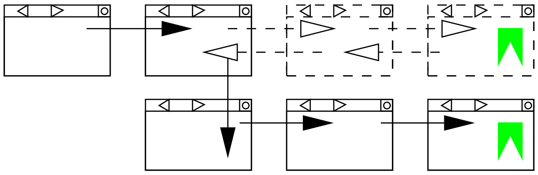
- Second, users can restore a prior exploration. That is, a user might initiate and bookmark a exploration as in this sequence of browser windows:



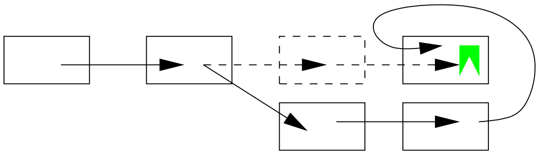
Then the user may explore an entirely different one,



but when invoking the bookmark restores the interface to its state at bookmark creation:



This results in the following interaction diagram:



Despite these interaction patterns, some people view Web applications as much simpler than GUI applications. We believe this view prevails for a few reasons:

- GUI programmers have to contend with staggeringly large interface toolkits [22]. Most Web applications generate little more than HTML using string processing.

- The relative youth of the Web means most applications have yet to reach the level of maturity of GUI programs. They perform fewer tasks, and thus don't deal with the same complexity of design demands.

- The complexity of GUI programs often ensues from the presence of explicit callbacks, concurrency and synchronization. These do not appear explicitly in Web programs, though failing to account for their implicit presence leads to erroneous interactions [25].

- Psychologically, users may be better conditioned to accepting erroneous behavior from Web applications, many of which run on remote sites and charge no direct fees, than from a GUI application that runs locally and charges the user a tangible up-front cost.

Web programs are, nevertheless, no simpler in *interaction flow* than GUI applications. For instance, every Web form submission is effectively a callback, with the CGI program [23] or servlet [6] referenced by the form's "action" field taking the place of a method invocation; cloning introduces concurrency. Furthermore, as our exposition illustrates, the Web already engenders *more complex* interaction flows than most GUI applications do, due to the features that browsers offer.

2.1 Desiderata

The complex interaction flows of the Web extract a price on the implementor, but translate into much greater convenience for the user. We believe users of GUI programs can benefit from these same features. Consider the software installer we mentioned in the introduction. Installers provide Forward and Back buttons for browsing the history of configuration choices. Large packages often include a "custom" installation mode, in which a user selects and de-selects individual components. When the user finishes choosing a configuration, the installer provides a summary of the space consumed for the components requested and confirms the configuration before it begins installation.

Suppose a user selects several components, but then finds the resulting installation would leave too little free disk space. As a result, he uses Back to return to an earlier menu, then navigates to a different configuration. With new choices made, perhaps the user wants to compare the disk usage to that consumed in the first configuration. Each time the user goes back and changes a configuration, however, he loses the previous one, as demonstrated by the GUI interaction diagrams. He therefore needs to have recorded the space usage for each prior configuration with some external utility.

Worse, suppose the user prefers an earlier configuration. He now must undo the currently active choices and somehow reproduce the preferred one. Therefore, he needs to have externally also recorded what components comprised each configuration. If the system includes several components, reproducing a configuration can be an extremely frustrating task.

This problem is not limited to installers. For example:

- Various "wizards" similarly guide users through application-specific choices. Many users find them frustrating because they limit user choices, but a flexible wizard need not do this: it can allow users to, at will, explore a large segment of the solution space, and to compare between solutions. Indeed, installers are merely simple forms of wizards.

- Many documentation browsers also don't permit users to clone a browser or to bookmark it.

- Readers may have noticed similar limitations in point-of-sale software, both in stores and in dial-in services (such as airline reservations), which keeps sales agents from effectively answering questions about multiple competing scenarios.

- This problem is even manifest in file browsing. Early versions of Windows opened a new filesystem browser each time a user entered a sub-directory. Because this can lead to a large number of open windows, later versions did not open a new window by default. A user can still open a new window on demand by Control+double-clicking on a folder, thus demonstrating the value of this interaction mode when made available in a controlled manner.

In short, this problem arises in numerous software contexts. The manual solution is time-consuming, error-prone, and ultimately distasteful because it forces the user to do what the computer ought to do instead. The problem is a mismatch between user need and software implementation. Our goal is to close this gap by bringing the implementation closer to the needs of the user.

In sum, we would like to give GUI users the same benefits that Web users currently enjoy. GUI users should be able to explore competing scenarios in parallel before committing to one (or more) of them. They should even be able to suspend the execution of a GUI by creating a "bookmark" and resume the computation whenever it is more appropriate or convenient.

2.2 Implementation Challenges

To understand the challenges a GUI implementor will face to provide these additional features, it is useful to study the problems Web programmers must confront [18]. Web traversal patterns make strenuous demands on software developers, such as:

- The use of the Back button, which corresponds to backtracking, is a client-side event that does not notify the server (because the previous page is usually in the cache). This means the application's view of the user's location in the browsing tree—represented by the last page the user requested—may differ from the user's actual location. Thus when the user submits a new request, the application may generate the wrong data, or may not even understand the request. Fortunately, a GUI programmer can usually detect backtracking events, such as closing a window or clicking on Back in an installer.

- Users may initiate many more Web computations than they complete. A few users might "log out" or otherwise explicitly terminate the transaction, but many users simply abandon it. This forces developers to contend with large-scale, distributed resource management problems. In contrast, GUI users normally explicitly terminate unnecessary applications, at which point the operating system reclaims resources.

- Cloning the browser window creates the potential for concurrency, because the user can submit requests from both clones at virtually the same time. The browser does not inform the application about cloning, so a developer must always anticipate the possibility of race conditions. When GUIs allow users to perform parallel explorations, programmers will have to attend to the same synchronization needs.

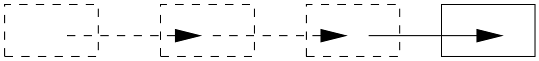
- When users resume a computation by visiting a bookmark, they expect the application to remember the information they had provided at the time they created the bookmark. The developer must therefore determine how and where to store these data.

More of the problems of the Web will infiltrate GUIs as researchers begin to build *distributed* GUIs [13]. Meanwhile, problems such as the possibility of race conditions and having to remember data in a bookmark remain difficulties that a programmer building GUIs with flexible interfaces must surmount.

2.3 Stack Patterns in Flexible GUIs

We illustrate challenges implementing the complex control flows necessary for GUIs with flexible interfaces by examining how representations of control information must evolve to accommodate additional functionality. We drive the exposition using an example that should be familiar to an academic audience: the paper submission portion of a conference manager. We have developed and deployed a Web version of this program to manage a conference co-chaired by the second author. Here, we present a greatly abridged form of its GUI counterpart. In each version of this example, the first window prompts for the author's personal information (name and email address). The second asks for a confirmation code (which is emailed to the address provided in the first window). The third window accepts information specific to the paper (title, abstract and filename). The last window prints an acknowledgment. We use the Swing API [8] of Java [16] in all of our examples, but our results are independent of the GUI library and the language.

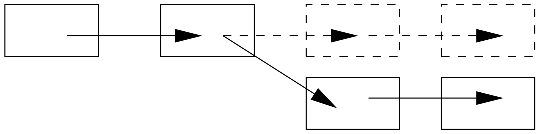
We begin by studying programs that implement the interaction diagrams of section 2. Figure 1 implements the first interaction diagram:



This program presents each of the submission windows in order. The program goes to lengths to avoid providing a convenient interface: each window simply returns the user's choice (by mutating a variable in its callback) instead of invoking a method that would generate the next input window. A given interaction with a user might result in the following stack snapshots (stacks grow upward), where each call to *promptRead* simply replaces the previous one:

promptRead	promptRead
main	main
(1)	(2)

Recognizing the weaknesses of this program, the developer chooses to implement the interaction diagram



As detailed in section 2.1, this style of interaction is extremely common in a large number of applications ranging from software installers to filesystem browsers. The sample program, shown in figure 2, permits the user to close windows to return to prior windows and re-enter information. Examining the stack, we see

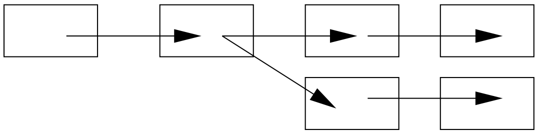
	Error		Paper
Confirm	Confirm	Confirm	Confirm
Contact	Contact	Contact	Contact
main	main	main	main
(1)	(2)	(3)	(4)

In the first snapshot, the user has entered the personal information, and is being prompted for the confirmation code. The second snapshot shows the error message frame that results from entering the wrong code. The user closes this window and returns to the previous one (3) to enter the correct code, which results in a paper submission dialog (4). This sequence demonstrates that the tree-shaped exploration maps to a stack by making sure only a linear thread of the exploration is active at any given time.

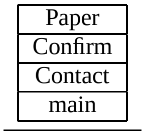
```
public class SubmitPaper {
    public static void main(String[] args) {
        System.out.println("Starting the Paper Submission Program.") ;
        enterPaperInDatabase() ;
        System.out.println("Finished the Paper Submission Program.") ; }
    private static void enterPaperInDatabase() {
        String[] contactInfo = DirectGUI.promptRead("Paper Submission: ", new String[]{"Name: ", "Email: "}) ;
        String name = contactInfo[0], email = contactInfo[1] ;
        String confCode = generateCode() ;
        Smtp.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode)) ;
        String[] codeAttempt = DirectGUI.promptRead("Confirm Email", new String[]{"Enter the confirmation code mailed to " + email}) ;
        if (codeAttempt[0].equals(confCode)) {
            String[] paperInfo = DirectGUI.promptRead("Paper Info", new String[]{"Title: ", "Abstract: ", "Paper Filename: "}) ;
            /* update database */
            DirectGUI.show("Thank you for your submission.") ; }
        else { DirectGUI.error("Incorrect confirmation code") ; }}
    private static String buildMessage(String name, String code) { /* ... */ }
    private static String generateCode() { /* ... */ }}
```

Figure 1: Inflexible Interface Version

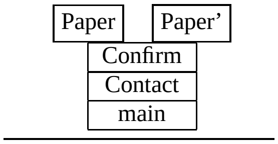
The next level of complexity in interaction diagrams is inspired by Web programs:



Suppose a programmer were able to implement this interaction mode. A user could begin to submit a paper

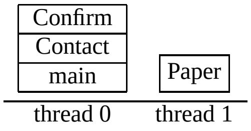


but then return to the window that spawns the paper information dialog and spawn a second such window, presumably to submit a second paper (the prime indicates that the frame refers to the same code, but may have different data):

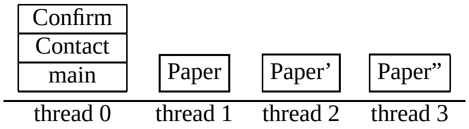


This stack snapshot demonstrates the difficulty in implementing such a feature: the simple mapping from the interaction diagram to the stack breaks down, since it violates the linear nature of the stack.

The interaction diagram demands that the programmer generate multiple branches of concurrently active stack fragments. This naturally suggests threads, which offer multiple concurrent stacks. It is, however, difficult to solve this problem by using threads alone. Spawning a thread in the class *Confirm* to run *Paper* might generate the following stack configuration:



That is, threads generate entirely new stacks with no direct control flow to the spawning stack. In contrast, the interaction diagram demands stack *fragments* with a *common base*. The programmer thus becomes responsible for coordinating between the completion of the frame *Paper* and the resumption of *Confirm*. If the user wants to submit multiple papers, the stack might resemble this:

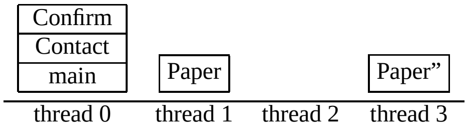


To address this problem, a sentinel in frame *Confirm* checks a shared variable. When one of the spawned threads populates the shared variable, the sentinel resumes control in the primary stack, returning the value to the frame *Contact*. The following procedural pseudocode illustrates this idea:

```
processConfirm() {
    ... build window ...
    Lock l = new Lock () ;
    JButton button = new JButton("Okay") ;
    button.addActionListener(new WakeOnClick(l)) ;
    ... show window ...
    l.wait() ;
    extract fieldValues ...
    return fieldValues ; }

class WakeOnClick {
    Lock l ;
    WakeOnClick(Lock lock) { l = lock ; }
    void actionPerformed(ActionEvent e) {
        spawn-thread {
            processPaper()
            l.notify() ; }}}
```

Besides its complexity, this solution has a *semantic* weakness. The sentinel returns when a thread (say the second one spawned) first writes to the shared variable, reducing the stack to this configuration:

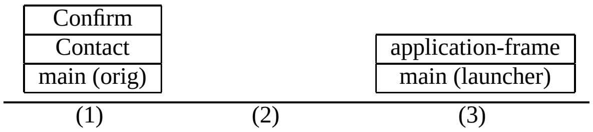


```
public class SubmitPaper2 {
  public static void main(String[] args) {
    System.out.println("Starting the Paper Submission Program.") ;
    Lock lock = new Lock() ;
    enterPaperInDatabase(lock) ;
    synchronized (lock) { lock.wait() ; }
    System.out.println("Finished the Paper Submission Program.") ; }
  public static void enterPaperInDatabase(final Lock lock) {
    GUI.promptRead("Paper Submission: ", new String[]{"Name: ", "Email: "}, new ProcessContact(lock)) ; }}
class ProcessContact extends Continuation {
  Lock lock ;
  ProcessContact(Lock l) { lock = l ; }
  void invoke(String[] contactInfo) {
    final String name = contactInfo[0], email = contactInfo[1] ;
    final String confCode = generateCode() ;
    Smtp.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode)) ;
    GUI.promptRead("Confirm Email", new String[]{"Enter the confirmation code mailed to " + email},
      new ProcessConf(name, email, confCode, lock)) ; }
  private String buildMessage(String name, String code) { /* ... */ }
  private String generateCode() { /* ... */ }}
class ProcessConf extends Continuation {
  String name, email, confCode ;
  Lock lock ;
  ProcessConf(String n, String e, String c, Lock l) { name = n ; email = e ; confCode = c ; lock = l ; }
  void invoke(String[] codeAttempt) {
    if (codeAttempt[0].equals(confCode)) {
      GUI.promptRead("Paper Info", new String[]{"Title: ", "Abstract: ", "Paper Filename: "},
        new ProcessPaper(name, email, lock)) ; }
    else { GUI.error("Incorrect confirmation code") ;
        synchronized (lock) { lock.notify() ; }}}}
class ProcessPaper extends Continuation {
  String name, email ;
  Lock lock ;
  ProcessPaper(String n, String e, Lock l) { name = n ; email = e ; lock = l ; }
  void invoke(String[] paperInfo) { /* update database */
    GUI.show("Thank you for your submission.") ;
    synchronized (lock) { lock.notify() ; }}}
```

Figure 2: Version with Backtracking Support

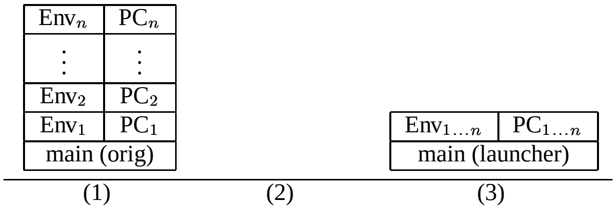
This means, however, that the sentinel is no longer present for the remaining threads. If the user decides to complete several of the spawned computations, only one of them can run to completion; the others languish for want of a sentinel. (While users will typically want to install only one configuration, in section *3.1.3* we explain why completing multiple scenarios is meaningful for many other applications.)

Inspired by the Web, the user might seek an even greater degree of convenience and power. Normally, aborting a program like a wizard or the paper submission suite forces users to restart the process from the beginning. Suppose, instead, a user enters the confirmation code, which leads to the paper information dialog, but realizes the paper is on a different filesystem. Rather than waste her labor, she may wish to bookmark the application itself, and quit its execution for now. To resume, she would use a special bookmark launcher. This generates the following stacks:



Snapshot (1) depicts the stack during bookmark creation. Snapshot (2) shows the empty stack, caused by exiting the application. Finally, snapshot (3) shows two frames. The bottom-most frame in a resumed program is always the bookmark launcher. The next frame is the first real frame of the resumed program.

This figure summarizes the responsibility that bookmarking computations thrusts on the programmer. They must ensure that the single resuming frame has enough information to perform the same computation that several frames did before termination. A more refined picture of the stack in the general situation is



The Env_i refer to the lexical environments [1] of each frame, while the PC_i are the return addresses, representing the remainder of the computation. The single frame in snapshot (3) must conceptually have a lexical environment representing all the environments; likewise, its return address must conceptually refer to a computation that reflects all the actions that would have been performed by returning through the stack in snapshot (1). Furthermore, the user may resume the bookmark multiple times. It is simply not clear how to implement this feature using a combination of function calls and threads. Felleisen proves this task is not possible with local transformations [10]. In the next section, we present a unified approach that tackles all these problems.[1]

3. IMPLEMENTING FLEXIBLE GUIS: CONTROL

To manage the complex control flows necessary for GUIs with flexible interfaces, we present a programming pattern that endows programs with these capabilities, and discuss its soundness and automation. We then briefly study the relationship between control flows internal to the program and those imposed by the user.

3.1 Transforming GUI Programs for Flexibility

The previous section demonstrates that increased complexity of flexible interaction flows results in a corresponding complexity of *control flows*. Programmers need a methodical discipline to systematically convert a program of the form in figure 1 to one that supports parallel exploration and bookmarking. This section presents a transformation that meets this need.

3.1.1 The Transformation

As we have demonstrated above, in the extreme case, the stack at the resumption point needs to effectively package all the stack frames that were active at the point of bookmark creation so that the launcher can correctly resume the computation. The programmer thus requires:

- a new source program such that the stack's return address on resumption refers to code that indeed completes the computation remaining when the user created the bookmark;
- data structures that preserve information, such as the values of lexical variables, current at the time of bookmark creation.

Object-oriented languages make it especially convenient to bundle data with program fragments that operate on the data. We exploit this to create a class of Continuation [11, 30] objects that encapsulate the necessary data structures with the modified source to restore and continue the computation. A Continuation object provides a single method, *invoke*, which resumes the computation. Using Continuation objects, the programmer can methodically *generate* the bookmarkable GUI program from a non-bookmarkable version:

1. He creates a concrete sub-class of the Continuation class for every call site in the program (except invocations of most system library methods, which remain unchanged). The Continuation class for a call site has a field for each lexical value live in the method body after the call returns.

2. The code in the class's *invoke* method contains all the code in the method body beginning from when the call returns until the next transformed method invocation.

3. Method invocations gain an addition argument containing the code to continue executing. Method returns become calls to Continuations' *invoke* methods.

The only exception to this rule is at a potential bookmarking point. Here, the program does not call *invoke* in the Continuation object, since this would continue the computation immediately. Instead, the programmer supplies the Continuation object to the callbacks for the various buttons. Most of them run *invoke* when the user selects the corresponding button. A bookmark button's callback, alone, marshals the Continuation object instead.

Figure 3 presents the transformed version of figure 1. The transformation applies the steps above, and also changes the interface of *promptRead* to accept a Continuation object. This latter change permits *promptRead* to spawn subsequent windows. When a user cancels one of these windows, *promptRead* distinguishes this from the submission of information, and permits the user to make another selection. This permits the user to backtrack through the windows.

On close inspection, we can see that the natural backtracking-friendly GUI code of figure 2 applies essentially the same transformation, partially and manually. The primary syntactic difference is that figure 3 uses inner classes to inline the class declarations. As we demonstrate and argue in the subsequent sections, the fully transformed program supports both parallel exploration and bookmarking.

3.1.2 Correctness and Automation

At this point, our presentation of the transformation has two significant weaknesses.

1. We have not offered any informal or formal reasoning of the two levels of correctness a programmer and user would care about:

   Rudimentary Correctness That the transformed programs preserve the semantics of the original if the user never spawns parallel requests or creates a bookmark.

   Extended Correctness That resuming a bookmarked program *does* correctly restore the state of the computation.

2. We have not discussed automation. The transformation involves a certain degree of manual labor. While it is methodical, it is tedious to perform manually, and the similarity between the original and the transformed version is clearly difficult to follow. This complicates both maintenance and debugging.

We address these in reverse order.

The transformation is clearly mechanical, and can therefore be implemented by a program rather than by a human. Indeed, we have implemented a prototype of this transformer for Java. Our prototype handles language features such as methods, conditionals and loops. As a result, programmers can develop and maintain the version in figure 1; the transformer generates code equivalent to figure 3 (we present it in a more readable form).

Automating the transformation addresses many important problems that arise when applying programming patterns. Avoiding direct manipulation of the transformed program simplifies software maintenance. It does not, however, immediately eliminate concerns about debugging. After all, many of the bugs in GUIs arise precisely from complex control flows and interactions.

We argue, however, that our transformations do not introduce *new* errors into the program. As we explain below, our transformation has a strong theoretical foundation that we can use to prove that it preserves the program's semantics. As we extend our transformer, we would need to correspondingly expand this reasoning. As a result, if the programmer is able to validate the behavior of the program with at most linear explorations, the transformer can extend this guarantee to parallel explorations and bookmarking.

[1] Note that many applications permit the bookmarking of *data*. This is relatively easy: in the simplest case, it may involve remembering no more than a filename or some comparable string datum. In contrast, we are interested in bookmarking *the running application itself*. We return to this issue in section *3.1.4*.

```
public class SubmitPaper3 {
  public static void main(String[] args) {
    System.out.println("Starting the Paper Submission Program.");
    enterPaperInDatabase(new Continuation_void() {
      void invoke() { System.out.println("Finished the Paper Submission Program."); }}); }
  public static void enterPaperInDatabase(final Continuation_void k) {
    GUI.promptRead("Paper Submission: ", new String[]{"Name: ", "Email: "},
      new Continuation() {
        void invoke(String[] contactInfo) {
          final String name = contactInfo[0], email = contactInfo[1];
          final String confCode = generateCode();
          Smtp.send(MAIL_SERVER, SMTP_PORT, FROM, email, buildMessage(name, confCode));
          GUI.promptRead("Confirm Email", new String[]{"Enter the confirmation code mailed to " + email},
            new Continuation() {
              void invoke(String[] codeAttempt) {
                if (codeAttempt[0].equals(confCode)) {
                  GUI.promptRead("Paper Info", new String[]{"Title: ", "Abstract: ", "Paper Filename: "},
                    new Continuation() {
                      void invoke(String[] paperInfo) { /* update database */
                        GUI.show("Thank you for your submission.");
                        k.invoke(); }}); }
                else { GUI.error("Incorrect confirmation code");
                    k.invoke(); }}}); }
        private String buildMessage(String name, String code) { /* ... */ }
        private String generateCode() { /* ... */ }}); }}
```

Figure 3: Transformed Version

This still leaves open the concerns about the fundamental correctness of the transformation itself. For this we defer to the literature since a careful reader will note that our transformation produces code essentially in *continuation-passing style* (CPS) [11, 24, 30], which preserves a program's meaning. This technique is commonly used in compilers for advanced functional languages [2]. While CPS is normally used by compilers to name intermediate terms (as observed by Sabry [26, 27]), it has the effect of making the continuations in the program explicit. These become our Continuation objects. Bookmarking becomes the act of saving and restoring continuations, which do indeed capture the state of the computation. Prior work on CPS can also guide the implementation of the transformer on control features such as exceptions [3]. We can, similarly, extend our transformer to handle multiple submission options in the form of *double-* and, in general, *multi-barrel* continuations [32].

3.1.3 Growing GUIs on Trees

With this machinery in hand, we can now return to the problem of permitting parallel GUI explorations. The transformational approach based on CPS handles this problem naturally. By providing a data structure representation of the stack, the transformation permits the use of a tree-shaped "stack" without interference from the underlying architecture. The event queue of the GUI automatically queues multiple submissions.

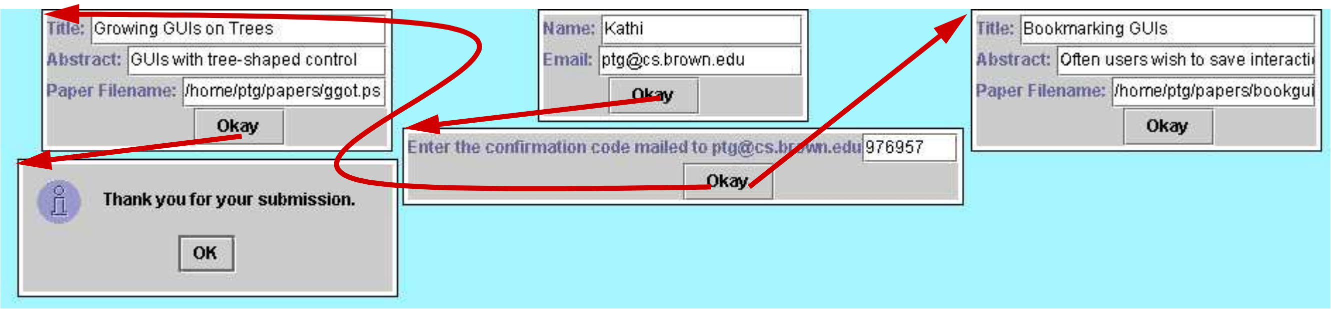
Besides the benefits of simplicity and formality, a programmer has another important reason to prefer our approach to the thread-based protocol of section 2.3. As we described earlier, the thread-based implementation most easily permits the user to continue only *one* interaction branch past a function return—an implementation detail invisible and possibly counterintuitive to an unsuspecting user. Subsequent selections will go unheeded because the sentinel has no spawning stack left to resume.

For some applications, such as installers, permitting only one final selection is usually the correct semantics. In this case, our transformer can easily generate the appropriate code to implement *one-shot* semantics [19]. For many other applications, however, users should be free to take multiple choices through to completion. For instance, perhaps the user wants to book several related flights (the cities may be the same, but the dates may differ), or to generate multiple, related network configurations using the wizard (the email address and personal information are the same, but the choice of SMTP server changes depending on the location).

Figure 4 demonstrates the use of multiple submissions in our running example. The order of forms in the conference submission program is deliberately chosen so an author needs to provide her contact information only once, and can then submit multiple papers. In the picture, she initiates the submission process by providing her contact information. She receives a confirmation code by email, which she enters into the validation window. She can now click on the Validate button as many times as necessary. Each click offers the opportunity to submit a (different) paper. In this instance, she chooses to take both submission windows to completion, resulting in two conference submissions. The transformed program shares common prefixes of the stack, but invocation essentially duplicates these shared frames on demand.

3.1.4 Bookmarking GUIs

The transformation of section *3.1.2* handles bookmarking also, and indeed is inspired by the need to do this. In particular, our transformer can automatically generate the callback for a Bookmark button. Clicking Bookmark prompts the user for a filename, to which the application writes the serialized [31] continuation object. For example, consider this modified fragment of *promptRead* used in figure 3. The transformer replaces the bookmark button's callback with a *Serializable* instance of Swing's *ActionListener* [8], where *frame* is the frame that contains the continuation object:



Title: Growing GUIs on Trees
Abstract: GUIs with tree-shaped control
Paper Filename: /home/ptg/papers/ggot.ps

Name: Kathi
Email: ptg@cs.brown.edu

Enter the confirmation code mailed to ptg@cs.brown.edu 976957

Title: Bookmarking GUIs
Abstract: Often users wish to save interacti
Paper Filename: /home/ptg/papers/bookgui

Thank you for your submission.

Figure 4: Multiple Submissions in Action

```
new SerialListener() {
    public void actionPerformed(ActionEvent e) {
        String fileName = /* pick bookmark file name */ ;
        FileOutputStream baos = new FileOutputStream(fileName) ;
        ObjectOutputStream oos = new ObjectOutputStream(baos) ;
        oos.writeObject(frame) ; }}}
```

The corresponding bookmark launcher is

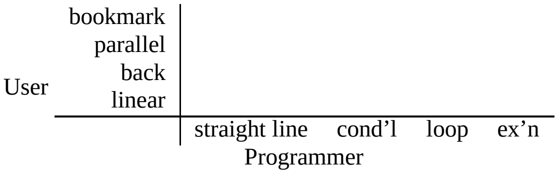
```
public class LoadBookmark {
    public static void main(String[] args) {
        FileInputStream fin = new FileInputStream(args[0]) ;
        ObjectInputStream ois = new ObjectInputStream(fin) ;
        JFrame frame = (JFrame)ois.readObject() ;
        frame.show() ; }}
```

Figure 5 shows screen-shots of this behavior. In the left panel, the user begins a paper submission and bookmarks the application (in this case, the bookmark file is roughly 15Kb in size). In the right panel, the user resumes the bookmark on the appropriate machine to finish submitting the paper.

3.2 Two Dimensions of Complexity

Our exposition above deals with the conversion of conventional GUI programs into more flexible variants. The program we present in figure 1 is, however, rather simplistic. The actual conference software has a loop, to check for the syntactic correctness of the email address, and catches exceptions raised by trying to email erroneous addresses. Our redaction for this paper has elided these details, but these and other features would be found in any realistic GUI program.

The simplicity of the source program, but the corresponding complexity of the control flows discussed in section 2.2, demonstrate that there are *two* sources of complexity in the interaction flow of flexible GUI programs. The internal dimension is controlled by the programmer, and usually handles the processing of data (for instance, a loop that iterates over elements in a collection). The external dimension is generated by the user's requests, and hence lies outside the control of the programmer:



As the complexity grows in the internal dimension also, it becomes extremely difficult to satisfy the demands of both dimensions. This makes a methodical means of program construction that accounts for the various kinds of external control flows especially valuable to programmers. Our technique offers programmers this support.

4. IMPLEMENTING FLEXIBLE GUIS: DATA

We have thusfar discussed the subtleties of control flow in programs. Programs also have different patterns of *information flow*. Broadly, there are two kinds of information, which are familiar to authors of interpreters and compilers: *lexically updated* and *globally mutated* information, which reside, respectively, in the *lexical environment* and in the *mutable store* [1, 12]. When we automatically generate code for parallel exploration, we must distinguish between these so there are multiple copies of the former, but only one of the latter.

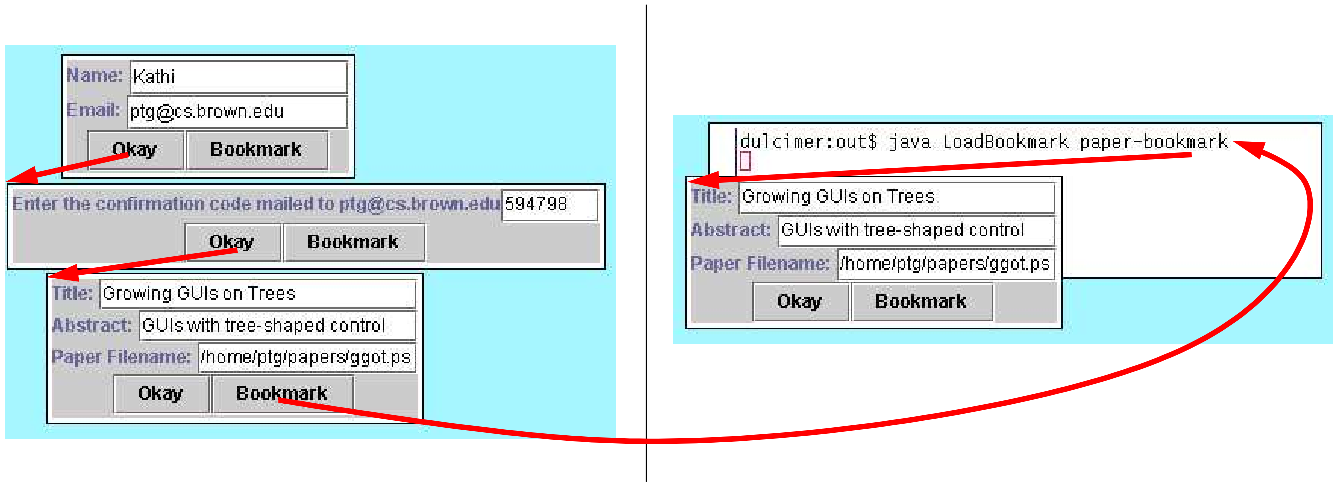
In a functional programming language such as ML [21], it is usually easy to distinguish between these two: by default information resides in the environment, unless it has type **ref**. Only variables of type **ref** are subject to mutation. In a language like Java, however, all variables receive values through mutation, even during initialization. This uniformity of syntax hides the fact that some variables are mutated only once, because of their lexical nature. We can identify this by examining which variables have static single assignment [7]; these are the lexical ones. This analysis is much simpler for Java than for languages supporting call-by-reference due to the lack of variable aliasing.

Our transformer is sensitive to this distinction between the environment and the store. Lexically updated variables reside in Continuation objects, which close over their values. There are hence as many copies of these variables (represented as Continuation object fields) as there are parallel explorations. In contrast, the transformation lifts mutated variables so that all continuations at a call site share the same ones. When we generate bookmarks, we must correspondingly maintain this distinction.

In a certain sense, all non-trivial GUI programs offer a weak version of bookmarking, in the form of configuration files. This common feature has two major shortcomings:

1. The configuration file is global, and thus shared by all executions of the program. (For instance, changing the default identity of the author in a word-processor affects all subsequent documents.) To work around this, a few programs permit multiple configurations ("profiles").

2. Programs sometimes incorrectly restore configuration data.



Name: Kathi

Email: ptg@cs.brown.edu

Enter the confirmation code mailed to ptg@cs.brown.edu 594798

Title: Growing GUIs on Trees

Abstract: GUIs with tree-shaped control

Paper Filename: /home/ptg/papers/ggot.ps

```
dulcimer:out$ java LoadBookmark paper-bookmark
```

Title: Growing GUIs on Trees

Abstract: GUIs with tree-shaped control

Paper Filename: /home/ptg/papers/ggot.ps

Machine 1 **Machine 2**

Figure 5: Bookmarking in Action

> They usually avoid this problem by not capturing complex configuration actions or ones that are not easy to marshal into files. This often forces users to manually recreate these configurations every time they use the program.

The second problem is automatically addressed by our transformations, because our bookmarks reflect the complete state of the program. Our analysis demonstrates that the first problem results from an incomplete understanding of the distinction between lexical and global information flows; our transformations automatically account for this distinction, saving the programmer from having to explicitly create support for profiles. Each bookmark thus behaves like a profile by keeping its lexical information separate from other bookmarks. For example, an academic reviewer could create a customized copy of a word-processor that hides her identity, which she can then use to annotate submissions with anonymous feedback.

5. RELATED WORK

Some related work develops novel methods of constructing GUIs without proposing new forms of user interaction. Other works provide alternate implementations of continuations on Java virtual machines without discussing GUIs at all. Earlier systems provided a mechanism for saving and resuming program state without separating the two kinds of information flow.

Elliot and Hudak's functional reactive animation [9] supports an unusual style of creating event-driven animations without explicitly dealing with timing details. In their system programmers define models of animation in terms of events that include the elapsing of specific amounts of time. The work focuses on a smaller-grained problem of handling graphics within windows rather than transitioning between windows.

GUIs written using Fudgets [4] also exhibit an unusual program structure. Programmers compose functions representing graphical components to create larger functions that thread streams of events from one component to the next. Although this may facilitate combining components, programmers must still weave events from one component to the next manually. The work does not attempt to provide more flexible interfaces to users.

Fuchs's Dreme [13] provides a mobile code system for Scheme [5]. It also addresses mobile user interfaces, and claims that callbacks in event-driven programs are a "twisted" form of continuations. The work does not address bookmarking GUIs, nor does it investigate new kinds of user interactions.

Most exception-based implementations of continuations on Java Virtual Machines fail to provide the flexibility of multiple resumptions our work requires. However, Fünfrocken [14] and Sakamoto, et al. [28] both provide an adequate mechanism for capturing and resuming control state on Java Virtual Machines using a global program transformation in conjunction with exceptions. Programmers could use their techniques to implement GUIs supporting parallel exploration, but they do not suggest doing so, nor does it extend to bookmarking.

Our work on GUIs developed out of studying Web programming. In particular, Hughes [20] and Queinnec [25] inspired our earlier work [17, 18] on interactive Web applications. These investigations exposed the additional flexibility provided by Web browsers and automated the construction of robust Web applications with matching flexibility.

The bookmarking portion of our work relies on Java serialization for object persistence. An object-oriented database system such as JavaSPIN [33] could replace serialization to better manage saved bookmarks. Packaging all of the remaining computation into a button callback greatly facilitates saving and restoring the user's interactions. A persistent store alone, however, does not enable parallel exploration or the ability to complete an operation multiple times.

Various programming environments for Common Lisp [29] and for Smalltalk [15] allow the user to save the entire application state and resume the program at a later time. This facility does not support parallel exploration easily since saving, quitting, and restarting the entire application for each switch between exploration branches is too cumbersome. This approach also copies any state shared between interactions.

6. CONCLUSION

This paper is motivated by the simple desire to compare the browsing power of different kinds of interactive software. In particular, we compare the patterns available to Web site users with those provided by many interactive graphical programs, using software installers as a representative example of these applications. By abstracting away the details of Web and GUI toolkits, we are able to reduce the use of these applications down to simple diagrammatic representations.

Our analysis demonstrates that, despite their numerous interface benefits, GUIs fall short of Web interfaces by failing to support key interaction patterns. In particular, Web users can both clone windows and create bookmarks while browsing, both of which are rarely if ever supported by graphical applications. We describe the implementation burdens a developer must overcome to provide this support. We then demonstrate that this support ensues automatically by transforming programs into an extension of *continuation-passing style*, a pattern more commonly found in the back-ends of compilers for advanced functional languages. We present this pattern in action by demonstrating its effect on a sample application.

Our work opens numerous avenues for future research. The most obvious question is how well it scales to interactive programs such as word-processors, where the granularity of interaction is at the level of a single keystroke. Depending on the structure of the code, bookmarks could either be too sparse (recording no past history) or too dense (recording all past keystrokes) with information. This may necessitate more forgiving forms of the transformation.

Second, we would want to use static a analysis to keep the size of bookmarks modest. We believe that in principle, it should be possible to reduce the size of bookmarks of the style described in section 4 to be very close to that of manually-constructed configuration files. As an initial effort, we can accomplish a similar end with much less sophistication by factoring out data common to multiple bookmarks, thus separating bookmarks into control and lexical data.

Ultimately, we believe our transformational ideas are best incorporated into interface-generation tools such as wizards. We believe the user traversal patterns an interface generates should be kept separate from its layout, and the synchronization and other needs imposed by the traversal should be generated automatically by a tool. We believe that our work presents a foundation for offering very general support of this form.

7. REFERENCES

[1] Aho, A. V., R. Sethi and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Inc., Reading, Mass., 1986.

[2] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Biagioni, E., K. Cline, P. Lee, C. Okasaki and C. Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, 1998.

[4] Carlsson, M. and T. Hallgren. Fudgets—a graphical user interface in a lazy functional language. In *Functional Programming and Computer Architecture*, 1993.

[5] Clinger, W. and J. Rees. Revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers*, pages 1–55, 1991.

[6] Coward, D. Java servlet specification version 2.3, October 2000. `http://java.sun.com/products/servlet/`.

[7] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Eckstein, R., M. Loy and D. Wood. *Java Swing*. O'Reilly, 1998.

[9] Elliot, C. and P. Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.

[10] Felleisen, M. On the expressive power of programming languages. In Jones, N., editor, *ESOP '90 3rd European Symposium on Programming, Copenhagen, Denmark*, volume 432, pages 134–151. Springer-Verlag, New York, N.Y., 1990.

[11] Fischer, M. J. Lambda-calculus schemata. In *Proceedings ACM Conference on Proving Assertions about Programs*, pages 104–109, Los Cruces, 1972.

[12] Friedman, D. P., M. Wand and C. T. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, MA, 1992.

[13] Fuchs, M. *Dreme: for Life in the Net*. PhD thesis, New York University, 1996.

[14] Fünfrocken, S. Transparent migration of Java-based mobile agents: Capturing and reestablishing the state of Java programs. In *Proceedings of the Second International Workshop on Mobile Agents*, pages 26–37. Springer-Verlag, September 1998. LNCS 1477.

[15] Goldberg, A. and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[16] Gosling, J., B. Joy and G. L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[17] Graunke, P., R. B. Findler, S. Krishnamurthi and M. Felleisen. Automatically restructuring programs for the web. In *IEEE International Conference on Automated Software Engineering*, November 2001.

[18] Graunke, P., S. Krishnamurthi, S. van der Hoeven and M. Felleisen. Programming the Web with high-level programming languages. In *European Symposium on Programming*, 2001.

[19] Haynes, C. T. and D. P. Friedman. Constraining control. In *12th ACM Symposium on Principles of Programming Langauges*, 1985.

[20] Hughes, J. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000.

[21] Milner, R., M. Tofte, R. Harper and D. MacQueen. The definition of Standard ML (revised), 1997.

[22] Myers, B., S. E. Hudson and R. Pausch. Past, present and future of user interface software tools. In Carroll, J. M., editor, *HCI In the New Millennium*, pages 213–233. ACM Press, Addison-Wesley, 2001.

[23] NCSA. The common gateway interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`.

[24] Plotkin, G. Call-by-name, call-by-value, and the λ-calculus. In *Theoretical Computer Science*, volume 1, pages 125–159, 1975.

[25] Queinnec, C. The influence of browsers on evaluators or, continuations to program web servers. In *ACM SIGPLAN International Conference on Functional Programming*, 2000.

[26] Sabry, A. *The Formal Relationship between Direct and Continuation-passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, 1994.

[27] Sabry, A. and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 1993.

[28] Sakamoto, T., T. Sekiguchi and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of the Joint Symposium on Agent Systems and Applications / Mobile Agents (ASA/MA)*, pages 16–28, September 2000.

[29] Steele Jr., G. L. *Common Lisp: The Language, 2nd Edition*. Digital Press, 1990.

[30] Strachey, C. and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, 1974.

[31] Sun Microsystems. Object serialization specification. `ftp://ftp.java.sun.com/docs/j2se1.3/serial-spec.pdf`.

[32] Thielecke, H. Comparing control constructs by typing double-barrelled CPS transforms. In *Third ACM SIGPLAN Workshop on Continuations*, 2001.

[33] Wileden, J. C., A. Kaplan, G. A. Myrestrand and J. V. Ridgway. Our SPIN on persistent Java: The JavaSPIN approach. In *First International Workshop on Persistence and Java*, September 1996.