User Interface Continuations

Dennis Quan, David Huynh, David R. Karger, Robert Miller
MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square, Cambridge, MA 02139 USA
E-mail: {dquan,dfhuynh}@ai.mit.edu; {karger,rcm}@lcs.mit.edu

ABSTRACT
Dialog boxes that collect parameters for commands often create ephemeral, unnatural interruptions of a program's normal execution flow, encouraging the user to complete the dialog box as quickly as possible in order for the program to process that command. In this paper we examine the idea of turning the act of collecting parameters from a user into a first class object called a user interface continuation. Programs can create user interface continuations by specifying what information is to be collected from the user and supplying a callback (i.e., a continuation) to be notified with the collected information. A partially completed user interface continuation can be saved as a new command, much as currying and partially evaluating a function with a set of parameters produces a new function. Furthermore, user interface continuations, like other continuation-passing paradigms, can be used to allow program execution to continue uninterrupted while the user determines a command's parameters at his or her leisure.

KEYWORDS: Continuations, dialog boxes

INTRODUCTION
Countless applications use dialog boxes to prompt the user for additional information needed to complete commands. Many dialog boxes are presented modally such that the user cannot use other functionality in the application until the dialog box is dismissed. Haphazard use of modal dialogs can inhibit the usability of a program. For example, some e-mail clients have a button that allows the user to look up a destination address in the address book, presented as a modal dialog box. The user experiences trouble when the person's name being looked up is actually in the body of the e-mail, obscured by the dialog box. Modal dialog box versions of base program functionality, such as address books, also tend to be less functional than their non-modal counterparts.

Although not modal, modeless dialog boxes and property inspectors can be similarly troublesome when users try to use them on more than one object at a time. For example, if an application exposes a font property inspector that allows the user to inspect the formatting of whatever text is selected, the user will have trouble trying to compare or copy the formatting of two different pieces of text at the same time. Furthermore, going back to the previous example, simply making the destination address lookup dialog box modeless is not the solution because the user may wish to select addresses for more than one e-mail at once, and any navigation state attained by browsing for a destination address for one e-mail (e.g., looking through contact groups, doing searches from a corporate directory, etc.) would be lost when the user temporarily switches to addressing one of the other e-mails.

These problems arise from the fact that applications do not treat the state of a command as a first class object. Unlike documents, which can be opened, saved, copied, and manipulated, dialog boxes are usually singleton and ephemeral. In this paper we propose that the in-progress state of a command be given first class status in a program. This is accomplished by packaging the code that will be executed upon completion of the dialog box as a *continuation* and attaching it to the in-progress state. The dialog box then becomes a manifestation of the first class continuation on the screen. Together, the dialog box and the continuation are referred to as a *user interface continuation*.

The definition of continuation we adopt here arises from the literature on continuation-passing style [7]. Conventional programs use stack frames to keep track of which function is currently being executed. A function completes when it releases its stack frame and returns to the calling function (the parent stack frame). In contrast, continuation passing style does not use a stack; instead, functions are called with an extra parameter known as a continuation. As the name implies, a continuation is a function that represents the remaining flow of execution of a program. Instead of returning a value, a function written in continuation passing style calls the supplied continuation with the return value. By analogy, a dialog box under our scheme calls the supplied continuation with the data gathered from the user.

Our approach brings about a number of advantages. First, user interface continuations gain many of the features of documents: the same sets of tools that can be used to look up information for insertion into a document can be employed

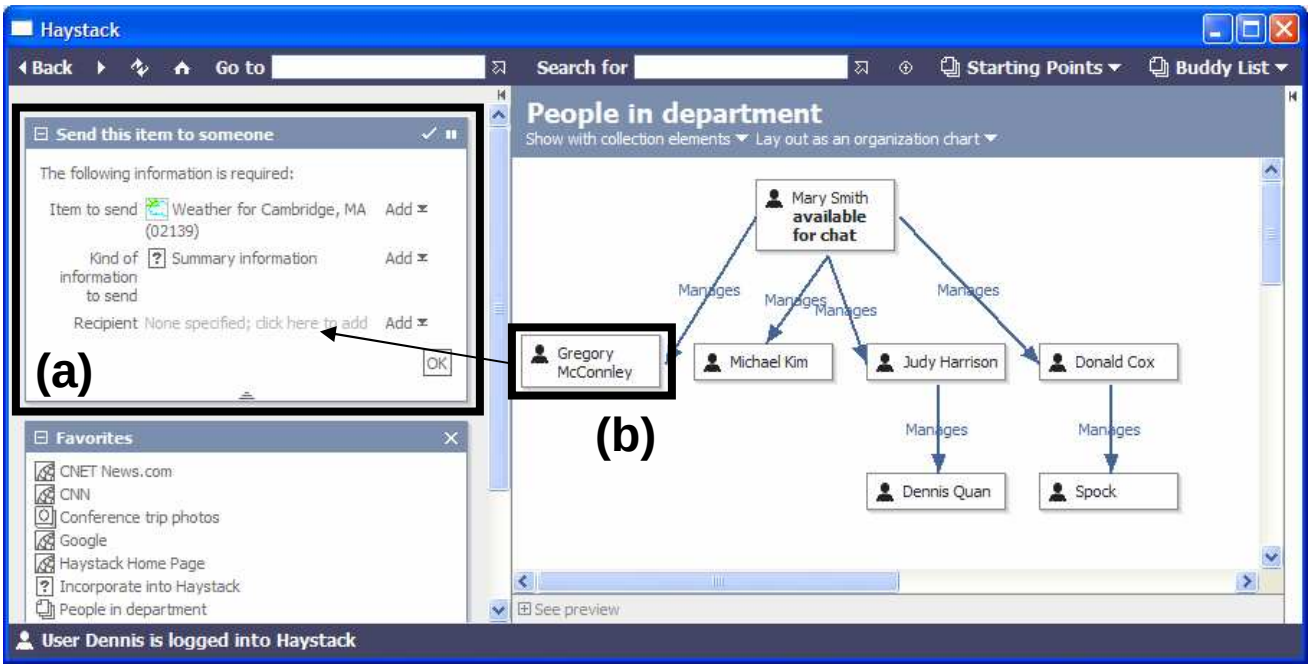

Figure 1: Screenshot of user interface continuation: (a) the user interface continuation; (b) dragging an item from a document into a user interface continuation

to satisfy continuations. There is also little urgency to complete a user interface continuation since the program does not need to suspend its state while waiting for the user to complete the command; the state needed to continue the program is encapsulated within the continuation. Finally, user interface continuations can be saved by means of a process called partial evaluation, which in effect creates a copy of an existing command in which some of the parameters have already been filled in. These partially completed commands can then be organized and searched like other documents.

We demonstrate these techniques in the context of Haystack, an environment designed to help users manage all their information, including e-mails, appointments, documents, and Web pages [1]. Haystack employs user interface continuations and other supporting abstractions to let users begin commands and complete them at their convenience. More information about Haystack, including a downloadable version of the system, can be found at the project home page: http://haystack.lcs.mit.edu/.

OPERATION ABSTRACTION

Haystack abstracts most user interface commands into operations—object-oriented pieces of functionality with defining metadata such as name, icon, and the types of the parameters. This metadata is defined using Haystack's flexible data model, which is based on the Resource Description Framework (RDF) [3]. RDF is a generalized directed graph representation that models metadata in terms of nodes (objects) and directed arcs (relationships between objects) and is used to model all of the metadata concerning the user's documents and other objects [2]. We have modeled operations in RDF, but any key-value pair metadata scheme (e.g., s-expressions, XML, etc.) can be employed. Indeed, the use of declarative specifications for commands is not new and has been investigated in the past; Myers et al. applied the technique of treating commands as objects for the purposes of supporting undo [5].

To illustrate the user interface continuation concepts discussed in this paper, we will describe the implementation of a command that allows information about any object in the system to be sent to some recipient. Pseudocode for this operation's metadata is given below:

```
MailAnObject
    type Operation
    title "Send this item to someone"
    params Recipients,ItemToSend,WhatPartsToSend

Recipients
    type Parameter
    title "Recipients"
    parameterType Person

ItemToSend
    type Parameter
    title "Item to send"
    parameterType Anything

WhatPartsToSend
    type Parameter
    title "Kind of information to send"
    parameterType InformationExtractor
```

To model the current state of an operation in Haystack we use an *operation closure*, which is an object that has, as properties, the parameters for an operation in progress.

Closures are also modeled in RDF. An example closure for our send object command is as follows:

```
closure20
    type Closure
    operation MailAnObject
    Recipients DonaldCox,MarySmith
    ItemToSend DepartmentMeeting
    WhatPartsToSend SummaryExtractor
```

USER INTERFACE CONTINUATIONS

When an operation that requires parameters is activated by means of a menu, a toolbar, or context menu, Haystack checks to see if the selection unambiguously satisfies any of the operation's parameter types. If there are unresolved parameters or the selection type checks against multiple parameters, Haystack exposes the in-progress operation closure as a user interface continuation. Like a dialog box, a user interface continuation prompts the user for needed information—in this case, the unresolved parameters.

Unlike modal dialog boxes, user interface continuations are modeless, allowing the user to use whatever tools in the system he or she is most familiar with to find the information needed to complete the operation. Our interface is similar to a shopping cart on an e-business website: the user can drag and drop relevant items into the "bins" representing the operation's parameters, as shown in Figure 1. The user can even decide to perform other tasks and come back to the operation later. When the user has finished obtaining the necessary information and is ready to perform the operation, he or she clicks the "OK" button on the user interface continuation. The system then invokes the continuation, which in the case of an operation invocation, is a function that performs the operation using the parameter bindings specified in the operation closure.

Support for user interface continuations is not dependent upon the software environment making use of declarative specifications of commands. The essence of a user interface continuation, as mentioned earlier, is a user interface for accepting values from the user (e.g., a dialog box) and a function to call when the user has finished supplying the needed information. However, user interface continuation support is especially well suited for use with declaratively-specified operations. With an operation abstraction, the act of presenting a dialog box is reduced to the job of displaying an editor for the operation closure. The presentation of a user interface continuation is automatically produced from an operation's declarative specification with widgets specialized for the kind of input required. (The problem of laying out dialog box widgets has been further explored in previous work [8].) In this way, Haystack frees the developer from needing to design specialized, miniature user interfaces for retrieving information from within modal dialog boxes, reusing the existing browsing environment and at the same time providing the user with a seamless experience. The operation's implementation, a function, is already written in a form that makes it suitable to be called from a continuation. Furthermore, the presentation of the user interface continuation can be customized by implementing a custom view (cf. Model-View-Controller) for the continuation [4].

CURRYING

Finally, users are able to save an in-progress operation closure and turn it into a new operation by selecting the option from the user interface continuation's context menu that instructs the system to bind the state of the current operation together with the already specified parameters. This binding process can be described as currying followed by partial evaluation, but we will refer to the entire process as currying as a shorthand. Currying is a term used in programming languages such as Haskell and ML that refers to the conversion of a function

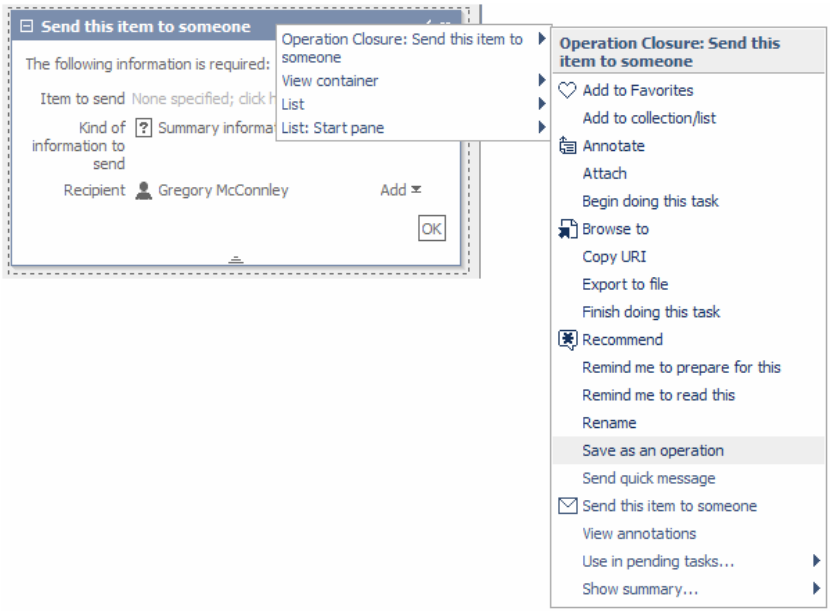

Figure 2: Currying an operation from the context menu

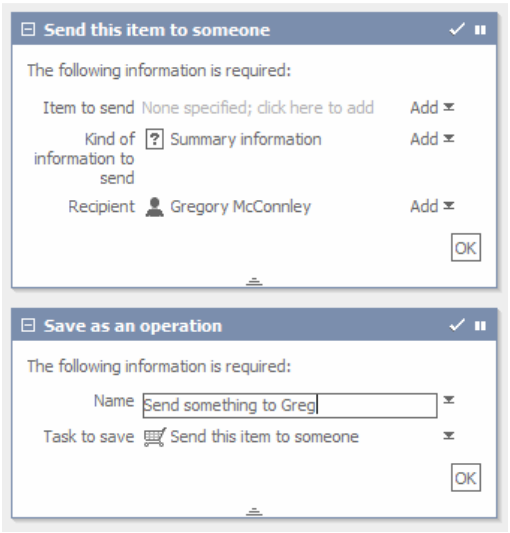

Figure 3: Creating a curried operation

that takes *n* parameters into a "curried" function that takes the first original parameter and returns a function that accepts the remaining *n - 1* parameters (also in a curried fashion). In other words, currying takes a function *f* of the form:

$f : a_1 \times a_2 \times \ldots \times a_n \rightarrow b$

and turns it into a function of the form:

curry$[f] : a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow b$

Similarly, when a user creates a new operation through currying, he or she is creating a new function from the curried form of the original function in which the parameters that have already been specified have been applied to the curried function. Another way to put this is if a user wishes to curry an operation *f* with the parameters a_1 through a_m already specified, then the resulting curried operation *g* has the following form:

$g =$ **uncurry**[**curry**$[f]\ a_1\ a_2\ \ldots\ a_m] : a_m \times a_{m+1} \times \ldots \times a_n \rightarrow b$

There is nothing special about the way in which currying in implemented in Haystack. Instead, currying is exposed to the user as simply another operation that takes an existing operation closure and a name as parameters. Figure 2 illustrates the use of a context menu for saving an operation closure as a new operation. Additionally, the screenshot given in Figure 3 depicts a user filling in a user interface continuation to create a new operation from an existing user interface continuation.

One benefit of currying is its ability to allow users to create specialized commands suited for their own purposes. In the example depicted in Figure 2, the user has likely observed that he or she sends summaries to Gregory McConnley frequently enough to warrant its own command. Most existing environments support this level of customized functionality only through macros or most recently used (MRU) lists. Furthermore, because curried operations and user interface continuations are described in the data model, they can be organized, searched, and shared with others just as documents can (e.g., placed in folders, sent as e-mail attachments, etc.).

Finally, currying can be used to construct first class support for command customizations. For example, the Print dialog box in Windows remembers settings such as copy count, which printer to use, and collation options only for the lifetime of an application. This simple approach obscures two important user interface problems. First, unless the user has observed the Print dialog box's behavior over a long period of time, the circumstances under which a program retains the last used print options may be unclear to the user at first. Second, there is no support for remembering more than one of the user's frequently-used configurations (e.g. double-sided duplicate copies with staples, one-sided single copy to the color printer, etc.). By storing these settings in curried operations, applications can give users first class support for commonly-used groups of settings while removing the ambiguity surrounding an application's policy on maintaining default options. (Developers would have to expose settings such as double-sidedness as parameters to the print operation rather than as properties of the printer. Also, any curried form of an operation used widely enough would likely gain built-in support from the application, e.g., "Print on Standard Paper"; our approach enables users to create such customizations without developer realization.) Although we have not implemented support for automatic generation of MRU lists, previous work has explored the notion of exposing such MRUs in the user interface [6]. In Haystack, users are free to place their curried operations into the system's menus and toolbars.

CONCLUSION

User interface continuations enable first class support for saving the state of a command and presenting it in a modeless fashion. Users benefit from using the tools already present within an application for locating the relevant parameters to the command instead of being restricted to the more limited functionality provided in special-purpose modal dialog boxes. Like documents, continuations can be completed at the user's leisure, saved as curried operations, and sent to colleagues. Finally, curried operations provide an elegant means for implementing customized commands without macros.

ACKNOWLEDGMENTS

This work was supported by the MIT-NTT collaboration, the MIT Oxygen project, and IBM.

REFERENCES

1. Huynh, D., Karger, D., and Quan, D. Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. In *Proceedings of the Semantic Web Workshop, The Eleventh World Wide Web Conference 2002*.
2. Quan, D., Huynh, D., and Karger, D. Haystack: A Platform for Authoring End User Semantic Web Applications. To appear in the *Proceedings of the International Semantic Web Conference 2003*.
3. Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/ REC-rdf-syntax-19990222/.
4. Quan, D., Karger, D., and Huynh, D. RDF Authoring Environments for End Users. *In Proceedings of Semantic Web Foundations and Application Technologies 2003*.
5. Myers, B., and Kosbie, D. Reusable Hierarchical Command Objects. *In Proceedings of CHI '96*.
6. Terry, M. and Mynatt, E. Side Views: Persistent, On-Demand Previews for Open-Ended Tasks. In *Proceedings of UIST '02*.
7. Steele, G. and Sussman, G. *LAMBDA: The Ultimate Imperative.* MIT Artificial Intelligence Laboratory Memo 353.
8. Vander Zanden, B. and Myers, B. Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces. In *Proceedings of CHI '90*.