Typed Output and Programming in the Interface

Francesmary Modugno Brad A. Myers

March 1993
CMU-CS-93-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work supported by the National Science Foundation under grant number IRI-9020089. The first author is supported by a fellowship from the Fannie and John Hertz Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: User Interfaces, Intelligent Interfaces, End-User Programming, Programming by Demonstration, Demonstrational Systems, Interaction Techniques

Abstract

A visual shell, (e.g., Apple Macintosh), is a direct manipulation interface to an operating system. Although such shells are easier to use than Unix, they have limitations: they are not programmable; the output of utilities can not be directly input to other utilities; and they lack the functionality of utilities such as `awk`.

PURSUIT is a visual shell that introduces a novel interface model, *typed utility output*, to address some of these limitations. In PURSUIT, *all* objects are typed. A typed object is an iconic representation of an object in the system, such as a file, user id, network address, etc., that users can manipulate in the traditional direct manipulation way.

Because all objects in PURSUIT are typed, all utility output is typed. By selecting part of the output of one utility and using it as input to another utility, users can create straight-line programs (i.e., Unix pipes) directly in the interface. By manipulating the contents of a typed output window, users can access the common functionality of utilities, such as column selection in `awk`, without knowing any special utility names and flags.

PURSUIT thus provides users with programming abilities, the common functionality of utilities, and the functionality of pipes *directly in the interface* without requiring them to learn obscure programming languages, commands or interaction techniques.

1 Introduction

Direct manipulation interfaces to operating systems ("visual shells"), such as the Macintosh Finder, are easy to use because they make users feel as if they are directly engaging with the objects in the interface rather than with a computer [7]. Unfortunately, this ease of use comes at the cost of losing the power and flexibility that has made Unix so successful. Specifically, visual shells do not have pipes, which allow users to connect the output of one command to the input of another; they do not provide the functionality of powerful utilities such as `awk`; and they are not programmable.

PURSUIT [8] is a novel visual shell that is addressing these problems. To add programmability, PURSUIT contains a programming by example (PBD) system. To write programs in PURSUIT, users execute actions on data and the PBD system constructs a general procedure. The system also produces an editable, graphical representation of the program, so that users can view and edit the code. This aspect of PURSUIT is discussed elsewhere [9].

Another key idea in PURSUIT is that *all* elements in the interface are typed objects. This enables users to manipulate these objects in meaningful ways, similar to the way they manipulate file and folder representations. Since all objects are typed, the output of utilities is also typed. For example, file names output by a utility become references to the actual file objects. Users can then input these objects to utilities that manipulate files, creating straight-line programs directly in the interface.

By creating typed utility output, PURSUIT brings two strengths of Unix into the visual domain: pipes and the common functionality of powerful utilities. Most importantly, PURSUIT provides this power in a way that is consistent with the direct manipulation paradigm.

2 Visual Pipes

Unix pipes enable the output of one utility to be input directly to another utility. They are one of the most frequently used Unix features [4]. The pipe mechanism gets its power from the fact that all input and output from utilities is an untyped stream of characters. However, it is *impossible* to

construct successful pipelines in Unix without some knowledge of what the input arguments represent and how the output is formatted.

For example, consider the following Unix command sequence: `rm ‘grep ‘‘typed output’’ * | awk -F: ’{print $1}’ | sort -u‘`. It removes all those files containing the string "typed output" from the current directory. Because the output of `grep` has the form *filename:text*, it cannot be piped directly into `rm`. To remove the extra colon and text, the output is filtered through an `awk` script. It is then piped through `sort` to remove duplicate file names.

Although Unix I/O is just a stream of characters, the sequence of commands succeeds only because the output of the `grep`, `awk`, `sort` pipeline and the input to `rm` are both lists of file names and are specified by strings of characters. That is, they have an implicit type (file-name) and identical specification (string). To successfully write the above sequence, one *must* know the form of the command outputs and what the outputs represent. Hence, Unix pipes are successful because of the implicit typing of command arguments and the symmetry of input and output.

PURSUIT applies these observations to the visual shell domain. All interface objects, and therefore all utility output, are typed. This provides a symmetry of input and output forms for PURSUIT utilities, just like Unix. Existing visual shells lack this symmetry. Usually, programs take as input iconic representations of objects (such as files), which are typed, but often output streams of text (such as dialog boxes), which are untyped. Thus, they can not use the same mechanism for pipes as for normal command invocation. Typing output objects enables users to construct visual pipelines because the output of a utility can be understood as input by other utilities.

3 Utility Functionality

Unix provides hundreds of utilities to manipulate data. However, only a small subset of utilities are ever used by most users [3]. In addition, most users access only simple, common functions of utilities [2].

PURSUIT provides the functionality of utilities in two ways: explicitly and implicitly. Some utilities, such as `sort` and `search` are invoked by explicitly selecting a menu item, icon, etc. However, the common functionality

of many utilities can be accessed implicitly by manipulating typed output. That is, interaction techniques applied to typed objects provide a functionality equivalent to some Unix utilities. Since users are familiar with both the meaning of the objects and the interaction techniques, this power is provided without requiring them to learn anything new. In addition, this power is provided in a direct way, rather than by explicitly invoking utilities. For example, by deleting a column of typed output, users have performed the same function as column selection in `awk`. Examples of other functionality are given below.

4 Related Work

Several existing graphical shells to Unix (e.g., [12], [5], [6], [11]) are successful in providing some file and directory management commands (e.g., `ls, cd, cp, rm, pwd`) in a direct way. This directness, however, is not extended to include pipes and utilities. Users specify pipes by typing a sequence of Unix commands in a special shell window. To access utilities, they must still write code in special languages or specify complicated sets of switches.

IShell [1] is a visual shell for Unix that contains iconic representations of applications. An application is invoked by dropping data onto its icon. The output is then placed into the cursor and can be "piped" into another application by dropping it onto that application's icon. IShell has several limitations. Users must learn the meaning and functionality of each icon: how it works, what input it accepts and what it outputs. This is no different than learning all flag options, argument formats and output forms in Unix. Also, an application can not have options, so it has only one behavior. The model is also space inefficient. As the number of applications grows, the desktop can get cluttered. Users may have difficulty organizing, locating and remembering applications.

5 Two Examples

The following two examples illustrate how typed objects easily enable users to construct pipelines and to access some of the common functionality of utilities. They illustrate how typed output enhances the visual shell without

increasing its complexity.

The first example shows how the above Unix command sequence is easily accomplished in PURSUIT. PURSUIT contains a `search` utility similar to the Unix `grep` utility. The user begins by invoking the utility and giving it the string "typed output". The utility creates a window containing a sequence of file and string objects, as shown in figure 1. The files in the window are references to the actual files they represent. To remove the actual files, the user clicks on the icons in the `search` output window, and drags them to the trash – the same sequence of actions the user would use on the actual files to remove them. Unlike the Unix sequence, the user did not need to know any special languages (like `awk`) or special flags to utilities. The user accesses the functionality of these utilities using a simple sequence of clicking, dragging and dropping actions. No extra knowledge is required.

As a second example, consider the command sequence `who | awk '{print $1 $5 }' | sort`. It prints the name and login time of all users currently logged on sorted by their user id's. To accomplish this in PURSUIT, users invoke the `who` utility, which produces the window shown in figure 2. They then select the three inner and the last "columns" of the output (figure 3) and drag them to the trash. Finally, they select the column of user id's and invoke the sort routine. The final window display is shown in figure 4. Again, the user has achieved some of the functionality of Unix utilities using a sequence of clicking, dragging and dropping actions that are the same ones they learned to operate the basic system functions. This power is made accessible without requiring the user to learn obscure commands, languages and switches.

6 Issues to Address

A prototype of the PURSUIT interface has been implemented using the Garnet [10] toolkit. The previous two examples illustrate some of the important issues uncovered by the prototype: defining the meaning of actions on typed objects; presenting polymorphic utilities to users; communicating user intent; and doing type coercion.

What does it mean?

In the first example, users drag the file references to the trash and the *orig-*

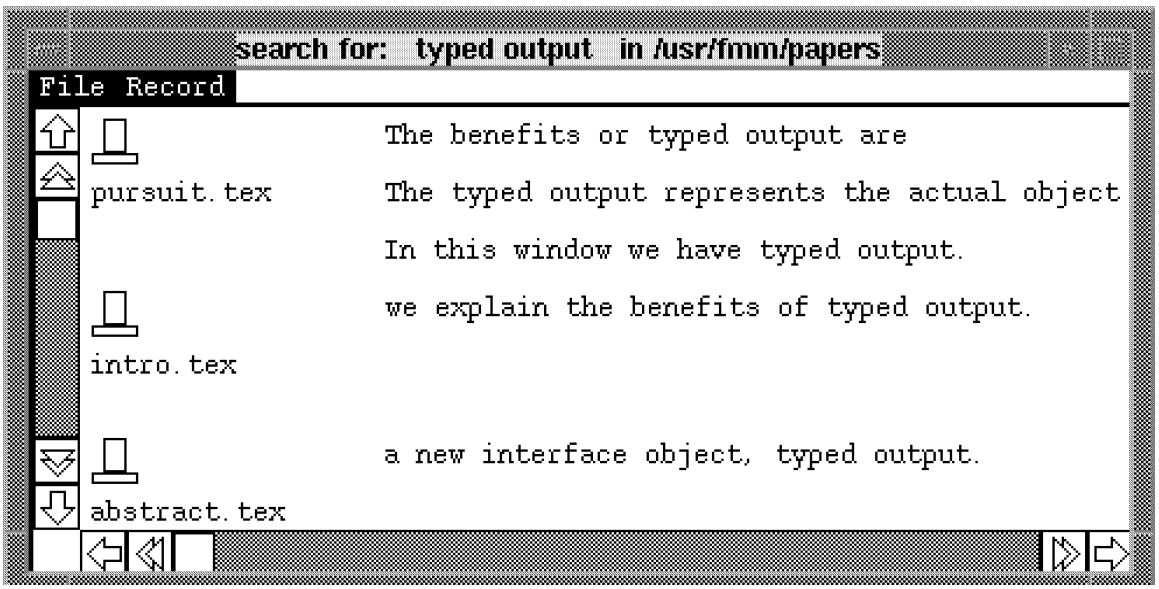

Figure 1: The output of the `search` utility containing both file and string objects. The file objects are references to the files they represent. The strings are the places in those files where the searched string was found. Users can perform operations on the referenced files and have the operations affect the actual files. They can also perform operations on the string objects. For example, double clicking on the string object with value "`The benefits of typed output are`" opens the *pursuit.tex* file scrolled to the position in the file containing that string.

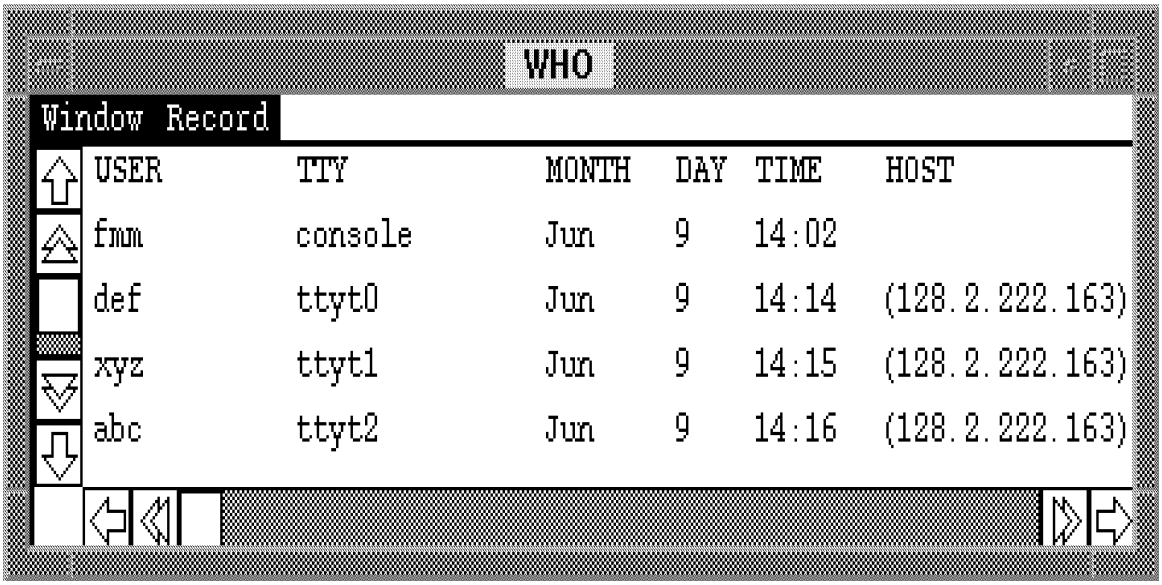

Figure 2: The output of the `who` utility.

inal files are deleted. What if they only wanted to deleted those references, and leave the original files unchanged? What if they dragged the references to another window? Would that move the original files, create copies of them, or created links to them? In the second example, supposed the user dragged the user-id's to the trash. Would that log all the users out? Or, would that just remove the user-id's from the window?

The problem here is clearly and precisely defining the meaning of actions on typed objects. The same action on objects of different types might have different meaning. This is consistent with the object oriented model. However, the same actions on objects of the same type may have different meanings depending on the location of the object. For a particular object in a given location, what does an action mean? Should the meanings of actions be pre-defined by the system, or should users define them?

Polymorphic Utilities and Inferring User Intent

The second example illustrates another problem: utilities must be polymorphic. In the example, the user wishes to sort the column of user id's. To do this the `sort` utility is invoked on objects of type user-id. But what does it mean to sort user-id objects? Normally, one sorts text. Does it confuse

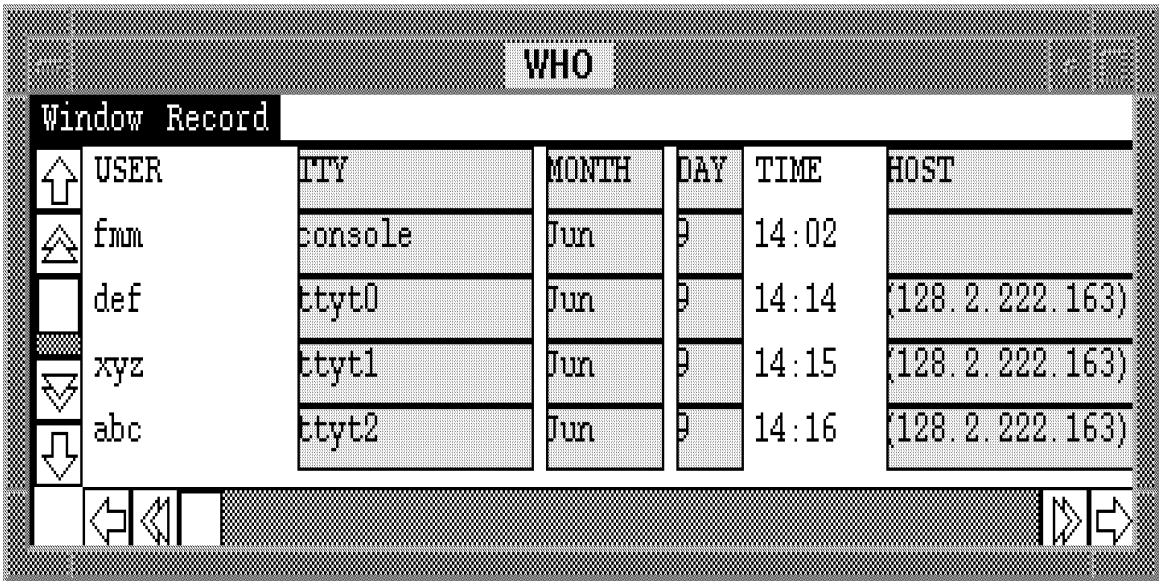

Figure 3: To delete the inner three and final columns, the user selects them and drags them to the trash.

the user to sort typed objects or is it clear that the utility does the "right thing".

A related problem is determining the best way for users to indicate their intentions. In the `who` example user id's were sorted by their login strings. What if the intent was to sort the objects by last names? How can this be determined? Must users explicitly specify their intentions, or can utilities guess the user's intentions, relying on heuristics to decide the best way to operate in a given situation? For example, can objects be sorted based on the value visible on the screen? Or, should the sort utility determine how to sort objects based on their location (e.g., a who window versus a folder window, etc.)?

In order for `sort` and other utilities to be polymorphic, utilities must operate on any typed object in any situation and users must understand the meaning of operating on typed objects in a particular situation. That is, the semantics of a utility's effects and the users' conceptualization of these effects must be the same. Further user testing is needed to determine how to best insure this.

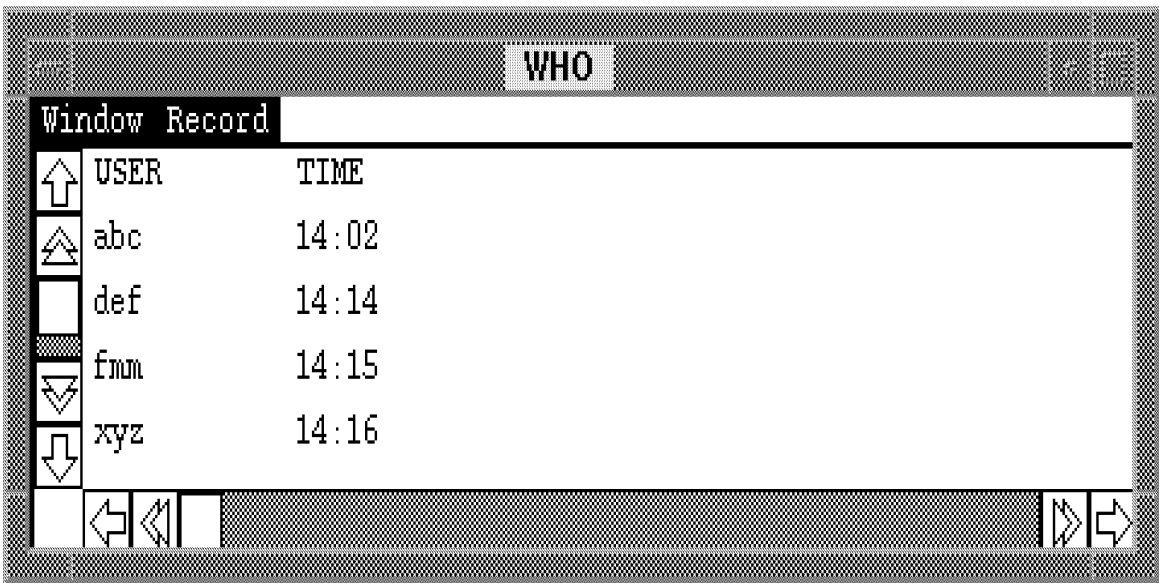

Figure 4: The `who` output window after the user invokes the `sort` routine on the column of user id's. The user has demonstrated the equivalent of the program "`who | awk '{print $1 $5 }' | sort`" using two explicit utility invocations and some familiar interaction techniques.

Type Coercion

A typed object may have different values depending on its location. For example, a user object may have a login id value in a `who` window. The same object may have a network address value in a `mail` utility window. A user may want to use the value of an object as input to a utility, although the object may not be the type of the input required by the utility. For example, the user may wish to input to the mail utility an object other than a user whose value is a valid mail address. Must applications be aware of the different types of objects they can be passed and coerce types where necessary, or should the shell do the type checking and any coercing before invoking applications? Do users need to be aware of the type conversion or should it be invisible to them? Is the "magic" of automatic type conversion confusing?

7 Conclusions and Future Work

We have presented a new idea for adding programmability to visual shells. Adding types to the interface seems natural. In Unix, users interpret text strings as representing objects, not just text. Similarly, in the graphical domain users often view interface elements as more than just shapes or strings of text. By making all interface elements typed, they become "real" objects that the user can manipulate in natural ways. This extends to users the ability to write programs and access the common functionality of utilities without requiring that they have any programming skills or that they know any special commands, languages, etc.

Using the PURSUIT prototype, we are exploring the many interesting questions presented in this paper. User testing is being done to determine the "natural" meaning of actions on objects of different types. Further testing will determine how to best present the ideas of type coercion and polymorphic utilities, as well as determine ways to communicate user intent. The results of these studies will be incorporated into the final PURSUIT design.

References

[1] Kjell Borg. Ishell: A visual unix shell. In *Proceedings of CHI '90*, pages

201–207, Seattle, Washington, April 1990.

[2] Keys Botzum. An empirical study of shell programs. Technical Report in Progress, Bell Communications Research, 1992.

[3] S. Greenberg and I. H. Witten. Directing the user interface: How people use command-based computer systems. In *Third IFAC/IFIP/IEA/IFORS Conference*, pages 349–355, Oulu, Finland, June 1988.

[4] Stephen Jose Hanson, Robert E. Kraut, and James M. Farber. Interface design and multivariate analysis of unix command use. *ACM Transactions on Office Information Systems*, 2(1):42–57, March 1984.

[5] Hewlett Packard. *HP Visual User Environment Version 2.0 Field Training Manual*, July 1990.

[6] Scott E. Hudson and Shamim P. Mohamed. A graphical user interface server for unix. *Software-Practice and Experience*, 20(12):1227–1239, December 1990.

[7] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design*, chapter 5, pages 87–124. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[8] Francesmary Modugno. *Pursuit: Adding Programming in the Interface.* PhD thesis, Carnegie Mellon University, In progress.

[9] Francesmary Modugno and Brad A. Myers. Graphical representation and feedback in a pbd system. In Allen Cypher, editor, *Watch What I Do: Programming By Demonstration*, chapter 20. MIT Press, 1993.

[10] Brad A. Myers et al. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[11] NeXT Inc.,. *The NeXT User's Reference Manual*, 1989.

[12] Isaac J. Salzman. A look at graphical shells. *Unix Review*, 8(11):103–113, September 1990.