Hyperbole Manual

Hypertext for Everyday Work

Bob Weiner
Brown University

Edition 2.02.2

December 13, 1991

The author's work on this project has been sponsored by Motorola Inc. under the Distinguished Student-Employee program.

Preface

This manual explains user operation and summarizes developer facilities of Hyperbole. It is for use with any version 2.02 or greater of Hyperbole.

This major release of Hyperbole concentrates on providing convenient access to information. The next major release will emphasize flexible views and structure manipulation within bodies of retrieved information.

Acknowledgments

Thanks to Mark Eichin, Dadong Wan, and Craig Smith for their feedback on the Beta test release of Hyperbole.

Carolyn Duby, David Niguidula, and Sumeet Singh provided lively discussions on the first version of Hyperbole. Peter Wegner has encouraged the growth in this work. Morris Moore and Q. Balzano have helped us pursue our own research visions and keep us striving for excellence.

Doug Engelbart has shown us the bigger picture and continues to be an inspiration. His work provides a model from which we are beginning to draw.

Introduction

This unit outlines the structure of this document, describes what Hyperbole is and lists some of its potential applications.

Manual Overview

Appendix A [Glossary], page 27, provides definitions of Hyperbole terms for quick reference, so terms are not precisely defined within the text. Be sure to reference the glossary if a term is unclear to you. Although you need not have a keen understanding of all of these terms, a quick scan of the Glossary should help throughout Hyperbole use.

Chapter 1 [Button Concepts], page 7, and Chapter 2 [Hyperbole Views], page 11, lay a conceptual foundation needed before the material in Chapter 3 [Using Hyperbole], page 13, can be utilized. See Appendix C [Questions and Answers], page 33 for responses to a number of frequent questions.

Developers comfortable with Emacs Lisp will want to continue on through Chapter 4 [Developing with Hyperbole], page 21. Appendix D [Ongoing Work], page 35 describes future directions in Hyperbole's evolution.

Hyperbole Overview

Hyperbole is an open, efficient, programmable prototype hypertext system that can be used for everyday work on any UNIX-type platform supported by GNU Emacs. It also works with the X Window-based version of Emacs from the University of Illinois, called Epoch. It is easy to pick up and use since it introduces only a few new mechanisms and provides user-level facilities through a menu interface, which you control from the keyboard or the mouse. Hyperbole is wholly written in Emacs Lisp for ease of modification. It includes a mouse interface for browsing documentation in GNU Info format.

Hyperbole consists of three parts:

1. an interactive hypertext management interface which anyone can use;
2. a set of programming library classes for system developers who want to integrate Hyperbole with another user interface or as a back-end to a distinct system; (Although Hyperbole is a prototype, it has been engineered for real usage and is well structured.)
3. a set of hyper-button types that provides core hypertext and other behaviors; Users can make simple changes to button types and those familiar with Emacs Lisp can quickly prototype and deliver new types.

Hyperbole may be used simply for browsing through documents pre-configured with Hyperbole buttons, in which case, this document explains more than one need master. Many users, however, will want to create their own buttons. The standard Hyperbole button editing user interface is GNU Emacs-based, so a basic familiarity with the Emacs editing model is useful. The material covered in the GNU Emacs tutorial, normally bound to `{C-h t}` within Emacs, is more than sufficient as background. If some GNU Emacs terms are unfamiliar to you, see section "Glossary" in *the GNU Emacs Manual*.

A Hyperbole user works with *buttons*; he may create, modify, move or delete buttons. Each button performs a specific action, such as link to a file or execute a shell command.

Presently, there are two distinct categories of Hyperbole buttons: *explicit buttons* (the ones that Hyperbole creates) and *implicit buttons* (buttons created and managed by other programs or embedded within the structure of a document and recognized contextually by Hyperbole). Explicit Hyperbole buttons may be embedded within any type of text file. Implicit buttons may be recognized anywhere within a text file, depending on the implicit button types that are available.

Hyperbole buttons may be clicked upon with a mouse to activate them or to describe their actions. Thus, a user can always check how a button will act before activating it. Buttons may also be activated from a keyboard. (In fact, all Hyperbole operations, including menu usage, may be performed from any standard character terminal interface, so one need not be anchored to a workstation all day). See Section 3.2 [Smart Keys], page 13.

Hyperbole does not enforce any particular hypertext or information management model, but instead allows you to organize your information in large or small chunks as you see fit. Some of its most important features include:

- Buttons may link to information or may execute procedures, such as starting or communicating with external programs;
- Buttons may be embedded within electronic mail messages;
- Other hypertext and information retrieval systems may be encapsulated under a Hyperbole user interface (a number of samples are provided).

Typical Hyperbole applications include:

personal information management
: Overlapping link paths provide a variety of views into an information space. A search facility locates buttons in context and permits quick selection.

documentation browsing
: Cross-references may be embedded within documentation. One can add a point-and-click interface to existing documentation, link code with associated design documents, or jump to the definition of an identifier from its use within code or its reference within documentation.

help/training systems
: Tutorials with embedded buttons show students how things work while explaining the concepts, e.g. an introduction to local commands. This technique can be much more effective than descriptions alone.

archive managers

Programs that manage archives from incoming information streams may be supplemented by having them add topic-based buttons that link to the archive holdings. Users can then search and create their own links to archive entries.

1 Button Concepts

This chapter explains the user-level notion of Hyperbole buttons. You should read it before reading Chapter 3 [Using Hyperbole], page 13.

Hyperbole buttons which are stored in files persist across Emacs sessions, so they provide a convenient means of linking from one information source to another.

1.1 Explicit Buttons

Hyperbole creates and manages *explicit buttons* which look like this `<(fake button)>` to a Hyperbole user. They are quickly recognizable, yet relatively non-distracting as one scans the text in which they are embedded. The text between the `<(` and `)>` delimiters is called the *button label*. Spacing between words within a button label is irrelevant to Hyperbole, so button labels may wrap across several lines without causing a problem.

Hyperbole stores the *button data* that gives an explicit button its behavior, separately from the button label, in a file named '`.hypb`' within the same directory as the file in which the button is created. Button data is comprised of individual *button attribute* values. A user never sees this data in its raw form but may see a formatted version by asking for help on a button.

Explicit buttons may be freely moved about within the buffer in which they are created. (No present support exists for moving buttons between buffers, but this is a high priority item for development.) A single button may also appear multiple times within the same buffer.

Explicit buttons are associated with action types which determine the actions they perform. *Link action types* connect buttons to particular types of referents. *Activation* of such buttons then displays the referents.

Hyperbole does not manage referent data; this is left to the applications that generate the data. This means that Hyperbole provides in-place linking and does not require reformatting of data to integrate it with a Hyperbole framework.

New forms of explicit buttons may be created simply by adding new action types to a Hyperbole environment. Look through the examples in '`hactypes.el`' and see `(actype:create)` in '`hbut.el`' for details.

1.2 Implicit Buttons and Types

Implicit buttons are those defined by the natural structure of a document. They are identified by contextual patterns which limit the locations in which they can appear. Their semantics are provided by action type specifications. They may use the same action types that explicit buttons do.

Implicit buttons never have any button data associated with them. They are recognized in context based on predicate matches defined within implicit button types. For example, Hyperbole recognizes file names enclosed in double quotes and can quickly display their associated files in response to simple mouse clicks.

See '`hibtypes.el`' for complete examples. Standard implicit button types include:

`annot-bib`
: Displays annotated bibliography entries referenced internally, delimeters = [].

`dir-summary`
: Detects filename buttons in files named "MANIFEST" or "DIR".

`hyp-source`
: Makes source entries in Hyperbole reports into buttons that jump to source.

`kbd-key`
: Executes a key sequence delimited by curly braces.

`man-apropos`
: Makes man apropos entries display associated man pages when selected.

`pathname`
: Makes a delimited, valid pathname display the path entry.

1.3 Action Types and Actions

Action types provide action procedures that specify button behavior. The arguments needed by an action type are prompted for at button creation time. When a button is activated, the stored arguments are fed to the action type's *action* body to achieve the desired result. Hyperbole handles all of this transparently.

Standard action types include:

`annot-bib`
: Follows internal ref KEY within an annotated bibliography, delimiters=[].

`eval-elisp`
: Evaluates a Lisp expression LISP-EXPR.

`exec-kbd-macro`
: Executes KBD-MACRO REPEAT-COUNT times.

`exec-shell-cmd`
: Executes a SHELL-CMD string asynchronously.

`hyp-source`
: Displays a buffer or file from a line beginning with 'hbut:source-prefix'.

`kbd-key`
: Executes the function binding for KEY-SEQUENCE, delimited by `{}`.

`link-to-Info-node`
: Displays an Info NODE in another window.

`link-to-buffer-tmp`
: Displays a BUFFER in another window.

`link-to-directory`
: Displays a DIRECTORY in Dired mode in another window.

`link-to-ebut`
: Performs action given by another button, specified by KEY and KEY-FILE.

`link-to-file`
: Displays a PATH in another window scrolled to optional POINT.

`link-to-mail`
: Displays mail msg with MAIL-MSG-ID from MAIL-FILE in other window.

`man-show`
: Displays man page on TOPIC, which may be of the form `<command>(<section>)`.

The use of action types provides a convenient way of specifying button behavior without the need to know how to program. Expert users who are familiar with Emacs Lisp, however, may find that they often want to tailor button actions in a variety of ways not easily captured within a type system. In such cases, *hui:ebut-prompt-for-action* should be set non-nil. This will cause Hyperbole to prompt for an action to override the button's action type at each explicit button creation. For those cases where the action type is sufficient, a nil value should be entered for the action. An action may be any Lisp form that may be evaluated.

1.4 Button and Type Precedences

Explicit buttons always take precedence over implicit buttons. Thus, if a button selection is made which falls within both an explicit and implicit button, only the explicit button will be selected. Explicit button labels are not allowed to overlap; Hyperbole's behavior in such cases is undefined.

If there is no explicit button at point during a selection request, then each implicit button type predicate is tested in turn until one returns non-nil or all are exhausted. The types are used in **reverse** order of definition, so that personal types defined after standard system types take precedence. It is important to keep this order in mind when defining new implicit button types. By making their match predicates as specific as possible, one can minimize any overlapping of implicit button type domains (those contexts in which a type predicates match).

Once a type name is defined, its precedence relative to other types remains the same even if you redefine the body of the type, as long as you don't change its name. This allows incremental modifications to types without having to worry about shifts in type precedence. See Section 4.2 [Creating Types], page 22, for information on how to develop or modify types.

2 Hyperbole Views

This chapter explains the role of views in the display of information accessed via Hyperbole. You should read it before reading Chapter 3 [Using Hyperbole], page 13.

This release of Hyperbole does not contain **any** support for views. A general view model is in development. The descriptions here are meant to familiarize you with the notion of views to be available in future releases.

A *Hyperbole view* is a perspective on a pool of information. An outline of a document, a document extract, and a decrypted document all represent views of an original document (which may just be a file).

View specifications or view specs, for short, control the application of views to displayed information. An action type which displays a buffer of text, the *result buffer*, can apply a view spec to the buffer before display. A user can then manipulate the view by interactively altering the view spec associated with the buffer. A 'V' in an action type's interactive call string indicates that the view spec of any button which uses the action type should be applied to the result buffer of the action type.

3 Using Hyperbole

This chapter covers user interaction with Hyperbole. Often a number of overlapping interaction methods are provided to support different work styles and hardware limitations. You need learn only one with which you can become comfortable, in such instances.

3.1 Initializing

Hyperbole must be installed at your site before you can use it. The `README` file that comes with the Hyperbole distribution explains how to install it and how to join the Hyperbole interest mail lists.

When Hyperbole is run under Epoch, it can automatically highlight any explicit buttons in a buffer and make them flash when selected. The only configuration necessary is selection of a color (or style) for button highlighting and button flashing. See the `hui-ep*.el` files for lists of potential colors and the code which supports this behavior. A call to `(ep:cycle-but-color)` within a Hyperbole initialization sequence changes the color used to highlight and flash explicit buttons by selecting the next color from a default list of color names.

Each time you start a new Emacs session, the Hyperbole code must be loaded for you to use it. The easiest way to achieve this is to add a line to your `~/.emacs` file, which Emacs reads automatically on startup. The line should be of the form: `(load "<HYP-DIR>/hsite")` where the local Hyperbole installation directory is substituted for `<HYP-DIR>`. If you use mouse keys, be sure to add this after any setup of mouse key bindings to ensure that Hyperbole's mouse keys are initialized. See Section 3.2 [Smart Keys], page 13, for further details.

If you want to customize the basic Hyperbole initialization sequence, you should make a personal copy of the `hsite.el` file, modify it the way you want, and then load it. (If you are familiar with Emacs Lisp, see also Section 4.1 [Hook Variables], page 21.) Once this is done, you may use all Hyperbole operations whenever you use Emacs or Epoch.

3.2 Smart Keys

Hyperbole provides two special keys that perform different operations in different contexts, as explained below. By default, these two keys are bound to your middle and right mouse keys when Hyperbole is run under an external window system (mouse configuration is automatic for the X Window System, SunView, and Apollo's Display Manager if your GNU Emacs program has been built with support for any of these window systems).

The middle key is called the *primary Smart Key* (or just Smart Key) and the right is called the *secondary Smart Key* (or Smart Meta Key). By default (if *smart-key-init* is left set equal to `t` in `hsite.el`), then `{C-u M-RET}` may also be used as the primary Smart Key and `{M-RET}` may be used as the secondary Smart Key. These key bindings allow context sensitive operation from any keyboard.

The primary Smart Key generally selects entities and activates buttons. The secondary Smart Key generally provides help, such as reporting on a button's attributes, or serves a complementary function to whatever the primary Smart Key does within a particular context.

When Hyperbole is installed, a key may be bound which allows you to switch between the Smart Key mouse bindings and your prior ones. `C-h w sm-mouse-toggle-bindings RTN` should show you any key which performs this command. If no key binding has been established or if you prefer one of your own, simply select a key and bind it within your '`~/.emacs`' file. For example, `(global-set-key "\C-ct" 'sm-mouse-toggle-bindings)`.

A prime design criterion of Hyperbole's user-interface is that one should be able check what an operation will do before applying the operation. Thus, one can get help on buttons and menu items before activating them by using the secondary Smart Key. Hyperbole also shows the result of a Smart Key direct selection of an argument value before it accepts the value for the argument.

When you use a mouse and you want to find out what either of the Smart Keys does within a context, depress the one you want to check on and hold it down, then press the other and release as you please. A help buffer will pop up explaining the actions that will be performed in that context, if any. A press of the primary Smart Key at the end of that help buffer will restore your display to its configuration prior to invoking help.

You can also get a complete listing of what the Smart Keys do in different contexts by pressing the secondary Smart Key within a context that has no special Smart Key function, e.g. the middle of a paragraph. The following table is an example of what you will be shown. Much of the browsing power of Hyperbole comes from use of the Smart Keys, so you should spend some time practicing how to use them.

```
==============================================================================
Context   Smart Key
Primary Secondary
==============================================================================
On Hyperbole menu item Item is activated Item help
On an explicit button Button is activated Button help
Reading Hyperbole argument
   1st press at an arg value      Value copied to minibuffer
   2nd press at an arg value      Value used as argument
   In minibuffer Minibuf arg is applied  Completion help
End of line, not end of buffer  Scrolls up a windowful Scrolls down
On an implicit button Button is activated Button help
End of Any Help buffer               Screen restored to previous state
Emacs occur-mode match      Jumps to match source line.
Explicit button search match      Jumps to match source line or file.
In the Emacs Info Viewer
   Menu Entry or Cross Ref Jumps to referent
   Up, Next or Prev Header Jumps to referent Jumps to prior node
   File entry of Header Jumps to top node Jumps to (DIR) node
   End of current node Jumps to next node Jumps to prev node
   Anywhere else Scrolls up a windowful  Scrolls down a wind
Any other context Hyperbole top menu      Smart Key summary
```

3.3 Operating Menus

Hyperbole menus provide access to user-level commands. Only one key binding is used for the top-level menu, but menu items may be selected from either the keyboard or via mouse clicks. When used with the keyboard, they provide rapid command access similar to key sequence bindings.

The top level menu is invoked from a key given in your '`hsite.el`' file (by default, `{C-h h}`) or via a Smart Key press in a random context. The menu will appear in the minibuffer and should look about like so:

```
Hyperb>  Act  ButFile...  EBut...  HistJump  IBut...  Report...  Types...
```

All button items are selected via the first character of their names (letter case does not matter) or via a press of the Smart Key. "..." at the end of an item name indicates that it brings up a sub-menu.

You can browse around the menus by hitting `{C-t}` to return to the top menu or `{q}` when you want to quit instead of selecting an item. `{C-g}` aborts whether you are at a menu prompt or any other Hyperbole prompt.

The above menu items can be summarized as follows:

Act Perform the action associated with any button at point.

ButFile... Display a local or global file of buttons, providing easy access. '`HYPB`' for a local button file and '`~/.hyperb/HYPB`' for your global file. These are good places to start your button creation testing.

EBut... All explicit button commands.

HistJump Jump back to last position in button traversal history.

IBut... All implicit button commands.

Report... Summarize buttons for current buffer.

Types... Summarize available types.

3.4 Entering Arguments

Many Hyperbole commands prompt you for arguments. The standard Hyperbole user interface has an extensive core of argument types that it recognizes. Whenever Hyperbole is prompting you for an argument, it knows the type that it needs and provides some error checking to help you get it right. More importantly, it allows you to click within an entity that you want to use as an argument and it will grab the appropriate thing and show it to you at the input prompt within the minibuffer. If you click on the same thing again, it accepts the entity as the argument and moves on. Thus, a double click registers a desired argument. Double-quoted strings, pathnames, mail messages, Info nodes, dired listings, buffers, numbers, completion items and so forth are all recognized at appropriate times. All of the argument types mentioned in the documentation for the Emacs Lisp `(interactive)` function are recognized. Experiment a little and you will quickly get used to this direct selection technique.

Wherever possible, standard Emacs completion is offered, see section "Completion" in *the Gnu Emacs Manual*. Remember to use `{?}` to see what your possibilities for an argument are. Once you have a list of possible completions on screen, you can double click the Smart Key on any one to enter it as the argument.

3.5 Working with Explicit Buttons

Explicit buttons provide the building blocks for creating personal or organizational hypertext networks with Hyperbole. This section summarizes the user-level operations available for managing these buttons.

3.5.1 Creating

The most efficient way to create an explicit button interactively is to mark a short region of text in any fashion allowed by GNU Emacs and then to select the Hyperbole menu item sequence, Ebut.../Create. You will be prompted for the button's label with the marked region as the default. If you accept the default and enter the rest of the information you are prompted for, the button will be created within the current buffer and Hyperbole will surround the marked region with explicit button delimiters to indicate success. The process becomes quite simple with a little practice.

If you do not mark a region before invoking the button create command, you will be prompted for both a label and a source buffer for the button and the delimited label text will be inserted into the chosen buffer after a successful button creation.

If a previous button with the same label exists in the same buffer, Hyperbole will add an *instance number* to the label when it adds the delimiters so that the name is unique. Thus, you don't have to worry about accidental button name conflicts. If you want the same button to appear in multiple places within the buffer, just enter the label again and delimit it yourself. Hyperbole will interpret all occurrences of the same delimited label within a buffer as the same button.

After Hyperbole has the button label and its source buffer, it will prompt you for an action type for the button. Use the `{?}` completion key to see the available types. The type selected determines any following values for which you will be prompted.

When creating link buttons, the best technique is to place the buffer in which you want your button and the entity to link to on screen at the same time. Mark the region of text to use for your button label, invoke the button create command from the menu, choose an action type which begins with `link-to-` and then use the direct selection techniques mentioned in Section 3.4 [Entering Arguments], page 16, to select the link referent.

3.5.2 Renaming

Once an explicit button has been created, its label text must be treated specially. Any inter-word spacing within the label may be freely changed, as may happen when a paragraph is refilled. But a special command must be invoked to rename it.

The rename command operates in two different ways. If point is within a button label when it is invoked, it will tell you to edit the button label and then invoke the rename command again. The second invocation will actually rename the button. If instead the command is originally invoked outside of any explicit button, it will prompt for the button label to replace and the label to replace it with and then will perform the rename.

The rename command may be invoked from the Hyperbole menu via Ebut.../Rename. A faster method is to use a key bound to the `hui:ebut-rename` command. Your site installation may include such a key. `C-h w hui:ebut-rename RTN` should show you any key it is on. If no key binding has been established or if you prefer one of your own, simply bind it within your '`~/.emacs`' file. We recommend the `{C-c r}` key, as in: `(global-set-key "\C-cr" 'hui:ebut-rename)`.

3.5.3 Deletion

Ebut.../Delete works similarly to the Rename command but deletes the selected button. The button's delimiters are removed to confirm the delete. Presently there is no way to recover a deleted button; it must be recreated. This will be remedied in the future.

3.5.4 Modification

Ebut.../Modify prompts you with each of the elements from the button's data list and allows you to modify each in turn.

3.5.5 Location

The Report... menu can be used to summarize all of the explicit buttons within a single buffer. The buttons may then be selected directly from the summary.

Ebut.../Search prompts for a search pattern and searches across all the locations in which you have previously created explicit buttons. It asks you whether to match to any part of a button label or only complete labels. It then displays a list of button matches with a single line of surrounding context from their sources. Any button in the match list may be activated as usual. A Smart Key press on the surrounding context jumps to the associated source line or a press on the filename preceding the matches jumps to the file without selecting a particular line.

There are presently no user-level facilities for globally locating buttons created by others or for searching on particular button attributes.

3.5.6 Button Files

It is often convenient to create lists of buttons that can be used as menus to provide centralized access to distributed information pools or for other purposes. These files can serve as useful roadmaps to help efficiently guide a user through both unfamiliar and highly familiar information spaces. Files that are created specifically for this purpose, we call *button files*.

The Hyperbole menu system provides quick access to two types of these button files: personal and directory-specific, through the ButFile... menu. (The variable, *hbmap:filename*, contains the base name of these standard button files. Its standard value is 'HYPB'.)

A personal button file may serve as a user's own roadmap to frequently used resources. The default personal button file is stored within the directory given by the *hbmap:dir-user* variable. Its standard value is '`~/.hyperb`'.

A directory-specific button file may exist for each file system directory. Such files are useful for explaining the contents of directories and pointing readers to particular highlights within the directories. Selection of ButFile.../DirFile menu item displays the button file for the current directory; this provides an easy means of updating this file when working on a file within the same directory. If you want to view some other directory-specific button file, simply use the normal Emacs file finding commands.

One might suggest that menu quick access be provided for group-specific and site-specific button files. Instead, link buttons to such things should be placed at the top of your personal button file. This provides a more flexible means of quick access.

3.5.7 Mailing Buttons

Hyperbole allows the embedding of buttons within electronic mail messages that are composed in Emacs via the standard `(mail)` command normally bound to `{C-x m}`. An enhanced mail reader can then be used which will allow activation of the buttons within messages just like any other buttons. Presently support is available only for the Emacs Rmail mail reader, see section "Rmail" in *the GNU Emacs Manual*. (The '`hmail.el`' file provides a generalized interface that can be used to hook in other mail readers when the necessary interface functions are written.)

All explicit buttons must be created within the *mail* outgoing mail buffer. There is no present support for including text from other buffers or files which contain explicit buttons. From a user's perspective, buttons are created in precisely the same way as in any other buffer. They also appear just like any other buttons to both the mail sender and the reader who uses the enhanced Rmail facilities automatically loaded by Hyperbole. Button operation may be tested any time before the message is sent. A person who does not use the Rmail reader can still send messages with embedded buttons since the process of composing mail is independent of any mail reader choice.

Hyperbole buttons embedded within received mail messages act just like any other buttons. The mail does not contain any of the action type definitions used by the buttons, so the receiver must have these or she will receive an error when she activates the buttons. Buttons which appear in message Subject lines are copied to Rmail summary buffers whenever such summaries are generated. Thus, they may be activated from either the Rmail or Rmail summary buffer.

Nothing bad will happen if a mail message with explicit buttons is sent to a non-Hyperbole or non-Rmail user. The user will simply see the text of the message followed by a series of lines of button data at its end. Hyperbole Rmail users never see this data in its raw form.

A final mail-related facility provided by Hyperbole is the ability to save a pointer to a received mail message by creating an explicit button with a `link-to-mail` action type. When prompted for the mail message to link to, if you press the Smart Key on an Rmail message, the appropriate parameter will be copied to the argument prompt, as described in Section 3.4 [Entering Arguments], page 16.

Future versions of Hyperbole will include much more extensive mail support.

4 Developing with Hyperbole

This chapter is only for people who are familiar with Emacs Lisp and wish to extend Hyperbole, customize Hyperbole or develop other systems using Hyperbole as a base.

4.1 Hook Variables

Hyperbole provides a number of hook variables that allow you to adjust its basic operations to meet your own needs, without requiring you to change the code for those operations.

We find it best to always set the value of hook variables either to nil or to a list of function names of no arguments, each of which will be called in sequence when the hook is triggered.

Given the name of a function, a Hyperbole hook variable triggered within that function has the same name as the function with a `-hook` appended. Hyperbole provides the following hook variables:

hyperb:init-hook
: For customization at Hyperbole initialization time. Use this to load any personal Hyperbole type definitions you might have.

ebut:create-hook
: To add to the Hyperbole explicit button creation process.

ebut:delete-hook
: To add to the Hyperbole explicit button deletion process.

ebut:modify-hook
: Executed when an explicit button's attributes are modified.

htype:create-hook
: Executed when a Hyperbole type (e.g. action type or implicit button type) is added to the environment.

htype:delete-hook
: Executed when a type is deleted from the environment.

Hyperbole also makes use of three hook variables that are a part of GNU Emacs.

find-file-hooks
: This is called whenever a file is read into a GNU Emacs buffer. Hyperbole uses it to highlight any buttons within files when run under Epoch, an X Window system based version of GNU Emacs.

write-file-hooks
: This is called whenever a GNU Emacs buffer is written to a file. Hyperbole uses it to save any modified button data associated with the file's directory.

rmail-show-message-hook
This is called whenever a message is displayed within the Rmail mail reader. Hyperbole uses it to ensure that the user never sees the button data that it stores within messages.

4.2 Creating Types

To define or redefine a single Hyperbole type, you may either:

- move your Emacs point to within the type definition and use `{C-M-x}` `(eval-defun)` (only works in Emacs Lisp mode);
- or move your point to the end of the last line of the type definition and use `{C-x C-e}` `(eval-last-sexp)` (works in most modes).

The functions from the `htype` class may be applied to any Hyperbole types, if needed.

The following subsections explain the specifics of Hyperbole type definitions which are beyond standard practice for Emacs Lisp programming. See the definitions of the standard types in '`hactypes.el`' and '`hibtypes.el`' for examples.

4.2.1 Action Types

An action type is created, i.e. loaded into the Hyperbole environment, with the `(defact)` function (which is an alias for `(actype:create)`). The calling signature for this function is given in its documentation; it is the same as that of `(defun)` except that a documentation string is required. (An interactive calling form is also required if the action type has formal parameters and is to be used in explicit button definitions. Implicit buttons never use an action type's interactive form. It is good practice to include an interactive form since the type creator cannot know how users may choose to apply the type.)

An action type's parameters are used differently than those of `(defun)`. Its interactive calling form is used when an explicit button is created to prompt for type-specific button attributes. The rest of its body is used when a button with that action type is activated. Then the type-specific button attributes together with the action type body are used to form an action that is executed in response to the button activation. The return value from the action is ignored.

An action type body may perform any computation using Emacs Lisp and Hyperbole functions.

The interactive calling form for an action type is of the same form as that of a regular Emacs Lisp function definition (see the documentation for the Emacs Lisp `(interactive)` form). It may additionally use Hyperbole command character extensions when the form is given as a string. The present extension characters are:

I Prompts for an existing Info node name and file.

M Prompts for a mail message date and the file name it resides in. The mail parameters prompted for by this character code are likely to change in the future.

V Prompts for a Hyperbole view specification. Not yet available for use.

Arguments are read by the functions in Hyperbole's `hargs` class, rather than the standard Lisp `read` functions, in order to allow direct selection of arguments via the Smart Keys.

If an action type create is successful, the symbol that Hyperbole uses internally to reference the type is returned. `Nil` is returned on failure so that you may test whether or not the operation succeeds.

Once you have defined an action type within your present Hyperbole environment, you can create new explicit buttons which use it. There is no explicit button type beyond its action type, so no further work is necessary.

Call `(actype:delete)` to remove an action type from a Hyperbole environment. It takes a single parameter which should be the same type symbol used in the type definition call (not the Hyperbole symbol returned by the call).

4.2.2 Implicit Button Types

[Be forewarned that the form of implicit button definitions is likely to change incompatibly in future major releases of Hyperbole to provide more extensive functionality, but any necessary changes to existing implicit button types should be minimal.]

An implicit button type is created or loaded via the `(defib)` function (which is an alias for `(ibtype:create)`). The calling signature for this function is given in its documentation; it is structurally similar to that of `(defun)` with a number of limitations. The parameter list should always be empty since no parameters will be used. A documentation string is required. The type's body follows this. An optional action type symbol may be given as a final argument after its body. The given action type is associated with the implicit button type or when not given, an action type with the same name as the implicit button type is used. The action type used may be created before or after the implicit button type definition but obviously should be defined before any implicit buttons of the given type may be activated.

The body of an implicit button type is a predicate which determines whether or not point is within an implicit button of the type. If not, the predicate returns `nil`. If so a one or three element list is returned (the decision of which kind of list to return is up to the type designer). A one element return list consists solely of the string to be used as the button label. A three element return list consists of the button label, the label's start position as an integer, and the label's end position as an integer. If the two position arguments are returned and Hyperbole flashes activated buttons on the current display, then it will flash buttons of this implicit type when they are activated. See Section 3.1 [Initializing], page 13.

The returned button label is fed to the button's action as its sole argument. Thus an implicit button action type must rely solely on the button's type (if it is used only with buttons of a single type), the button's label and the current Emacs environment to determine how to act.

If an implicit button type create is successful, the symbol that Hyperbole uses internally to reference the type is returned. `Nil` is returned on failure so that you may test whether or not the operation succeeds.

Once you have defined an implicit button type together with its action type within your current Hyperbole environment, you can create new explicit buttons which use it. Implicit button type names and action type names may be the same without any conflict. In fact, such naming is encouraged when an implicit button type uses an action type.

Call `(ibtype:delete)` to remove an implicit button type from a Hyperbole environment. It takes a single parameter which should be the same type symbol used in the type definition call (not the Hyperbole symbol returned by the call). This will not delete the action type used by the implicit button.

By default, a request for help on an implicit button will display the button's attributes in the same manner as is done for explicit buttons. For some implicit button types, other forms of help will be more appropriate. If an Emacs Lisp function whose name is formed from the concatenation of the type name followed by `:help`, e.g. `my-ibtype:help`, is defined, it is used to respond to requests for help on buttons of that type. Any such function should take a single argument of an implicit button construct. (This is what `(ibut:at-p)` returns when point is within an implicit button context.) The button may be queried for its attributes using functions from the `hbut` and `hattr` classes. See the '`hib-kbd.el`' file for an example of a custom help function.

4.3 Explicit Button Technicalities

4.3.1 Operational and Storage Formats

Hyperbole uses a terse format to store explicit buttons and a more meaningful one to show users and manipulate during editing. The terse format consists solely of button attribute values whereas the edit format includes an attribute name with each attribute value. A button in edit format consists of a Lisp symbol together with its property list which holds the attribute names and values. In this way, buttons may be passed along from function to function simply by passing the symbol to which the button is attached. Most functions utilize the pre-defined *hbut:current* symbol by default to store and retrieve the last encountered button in edit format.

The `hbdata` class handles the terse, stored format. The `hbut`, `ebut`, and `ibut` classes work with the name/value format. This separation permits the wholesale replacement of the storage manager with another, with any interface changes hidden from any Hyperbole client programming.

4.3.2 Programmatic Button Creation

A common need when developing with Hyperbole is the ability to create or modify explicit buttons without user interaction. For example, an application might require the addition of an explicit summary button to a file for each new mail message a user reads that contains a set of keywords. The user could then check the summary file and jump to desired messages quickly.

The Hyperbole class `ebut` supports programmatic access to explicit buttons. See it within the '`hbut.el`' file for full details. The documentation for `(ebut:create)` explains the set of attributes settings necessary to create an explicit button. For operations over the whole set of buttons within the visible (non-narrowed) portion of a buffer, use the `(ebut:map)` function.

4.4 Encapsulating Systems

A powerful use of implicit button types is to provide a Hyperbole-based interface to external systems. The basic idea is to interpret patterns within the application's displays as implicit buttons. See the '`hsys-*`' files for examples of how to do this.

4.5 Embedding Hyperbole

[NOTE: We have never done this ourselves, though we have done similar things which leads us to infer that the task should not be difficult.]

The standard Emacs-based Hyperbole user interface has purposely been separated from the Hyperbole backend to support the development of alternative interfaces and the embedding of Hyperbole functionality within other system prototypes. The Hyperbole backend functionality that system developers can make use of is called its Application Programming Interface (API). The API may be used to make server-based calls to Hyperbole when Emacs is run as a non-interactive (batch) process, with its input/output streams attached to another process.

The public functions and variables from the following files may be considered the present Hyperbole API:

'`hact.el`', '`hargs.el`', '`hbmap.el`', '`hbut.el`', '`hhist.el`', '`hmail.el`', '`hpath.el`', '`htz.el`', '`hypb.el`', and '`set.el`'.

(Note that within a class definition, functions and variables that follow a line of dashes are private.)

Appendix A Glossary

Concepts pertinent to operational usage of Hyperbole are defined here. If some GNU Emacs terms are unfamiliar to you, see section "Glossary" in *the GNU Emacs Manual*.

`action` An executable behavior associated with a Hyperbole button. A specific class of actions which display entities are called *links*, such as a link to a file.

`action type`
A behavioral specification for use within Hyperbole buttons. Action types usually contain a set of parameters which must be given values for each button with which they are associated. An action type together with a set of values, called arguments, may be considered an *action*. *Actype* is a synonym for action type.

`activation`
Request for a Hyperbole button to perform its action. Ordinarily the user presses a key which selects and activates a button.

`argument` A button-specific value fed to a Hyperbole type specification when the button is activated.

`attributes`
Slot names associated with Hyperbole buttons. An *attribute value* is associated with each button attribute.

`button` A selectable Hyperbole construct which performs an action. A button consists of a set of attributes that includes: a textual label, a category, a type and zero or more arguments. *Explicit buttons* also have creator, create time, last modifier, and last modifier time attributes.

Buttons provide the user's gateway to information. The user sees and interacts with button labels, the rest of the button data is managed invisibly by Hyperbole and displayed only in response to user queries.

`button activation`
See *activation*.

`button attributes`
See *attributes*.

`button data`
Lists of button attribute values explicitly saved and managed by Hyperbole. One list for each button created by Hyperbole.

`button file, global`
A per-user file named 'HYPB' that may be used to store any desired buttons and may then be displayed via a menu selection whenever a user is within that directory.

`button file, local`
A per-directory file named 'HYPB' that may be used to store any desired buttons and may then be displayed via a menu selection whenever a user is within that directory.

`button key`
A normalized form of a *button label* used internally by Hyperbole.

`button label`
: A text string that visually indicates a Hyperbole button location and provides it with a name and unique identifier. Within a buffer, buttons with the same label are considered separate views of the same button and so behave exactly alike. Since button labels are simply text strings, they may be embedded within any text to provide non-linear information or operational access points.

  A button is selected by pressing a *Smart Key* within its label. The maximum length of a button label is limited by the variable *ebut:max-len*.

`button selection`
: The act of designating a Hyperbole button upon which to operate.

`category`
: A high-level, conceptual grouping of Hyperbole buttons into classes. *Implicit* and *explicit* groupings represent categories.

`class`
: A group of functions and variables with the same prefix in their names, used to provide an interface to an internal or external Hyperbole abstraction.

`context`
: A programmatic or positional state recognized by Hyperbole. We speak of Smart Key and implicit button contexts. Both are typically defined in terms of surrounding patterns within a buffer, but may be defined by arbitrary Emacs Lisp predicates. (Context may come to have a broader meaning within future versions of Hyperbole.)

  See *Hyperbole environment*.

`explicit button`
: A button created and managed by Hyperbole. By default, explicit buttons are delimited like this `<(fake button)>`.

`global button file`
: See *button file, global*.

`hook variable`
: A variable that permits customization of an existing function's operation without the need to edit the function's code. See also the documentation for the function `(run-hooks)`.

`Hyperbole`
: A flexible, programmable information management and viewing system built on top of GNU Emacs. It utilizes a button-action model and supports hypertextual linkages. Hyperbole is all things to all people.

`Hyperbole environment`
: A programmatic context within which Hyperbole operates. This includes the set of Hyperbole types defined and the set of Hyperbole code modules loaded. It does not include the set of accessible buttons. Although the entire Emacs environment is available to Hyperbole, we do not speak of this as part of the Hyperbole environment.

`hypertext`
: A text or group of texts which may be explored in a non-linear fashion through associative linkages embedded throughout the text. Instead of simply referring to other pieces of work, hypertext references when followed actually take you to the works themselves.

`implicit button`
: A button recognized contextually by Hyperbole. Such buttons contain no button data. See also *implicit button type*.

`implicit button type`
: A specification of how to recognize and activate implicit buttons of a specific type. Implicit button types often utilize structure internal to documents created and managed without Hyperbole assistance, for example, programming documentation. *Ibtype* is a synonym for implicit button type. See also *system encapsulation*.

`instance number`
: A colon prefaced number appended to the label of a newly created button when the button's label duplicates the label of an existing button in the current buffer. This number makes the label unique and so allows any number of buttons with the same base label within a single buffer.

`link`
: A reference from a Hyperbole button to an entity. The referenced entity is sometimes called a *node* or *referent*. A specific class of actions which display entities are called *links*, such as a link to a file.

`local button file`
: See *button file, local*.

`mouse button`

`mouse key`
: See *Smart Key*.

`node`
: See *link*.

`predicate`
: A boolean (nil = false, non-nil = true) Lisp expression typically evaluated as part of a condition expression.

`referent`
: See *link*.

`Smart Key`
: A context-sensitive key used within Hyperbole and beyond. Actually, there are two Smart Keys, a primary and a secondary Smart Key, sometimes referred to as the Smart Key and Smart Meta Key, respectively. The primary Smart Key, typically bound to the middle mouse key, activates Hyperbole buttons and scrolls the buffer forward a screenful when pressed at the end of a line. The secondary Smart Key, typically bound to the right mouse key, explains what a Hyperbole button does or scrolls the buffer backward a screenful when pressed at the end of a line.

  To see what a Smart Key will do within a particular context, depress and hold the key at the point desired and depress the other Smart Key. A buffer containing a description of its contextual function will then be displayed. You can release the two keys in any order after you have them both depressed. A press of the secondary Smart Key in an unsupported context displays a summary of Smart Key functions in each context.

`source buffer / file`
: The buffer or file within which a Hyperbole button is embedded.

`system encapsulation`
: Use of Hyperbole to provide an improved or simply consistent user interface to another system. Typically, implicit button types are defined to recognize and activate button-type constructs managed by the other system.

`view`
: A perspective on some information. A view can affect the extent of the information displayed, its format, modes used to operate on it, its display location and so forth.

`view spec`
: A terse (and to the uninitiated, cryptic) string that specifies a particular view of a link referent or some other information.

Appendix B Suggestion or Bug Reporting

See the 'README' file in the Hyperbole distribution directory for complete details on Hyperbole mailing lists and how to subscribe.

If you find any errors in Hyperbole's operation or documentation, feel free to report them. Follow the appropriate details given for bug reporting in section "Bugs" in *the GNU Emacs Manual*, but also include the Hyperbole version number that appears in your top-level Hyperbole menu and send the bug report to the Hyperbole discussion list: hyperbole@cs.brown.edu.

Please use your subject line to state the position that your message takes on the topic that it addresses, e.g. send "Subject: Basic bug in top-level Hyperbole menu." rather than "Subject: Hyperbole bug". This simple rule makes all e-mail communication much easier.

If you have suggestions on how to improve Hyperbole, send them to the same address. Here are some issues you might address:

What did you like and dislike about the system?
What kinds of tasks, if any, does it seem to help you with?
What did you think of the Emacs-based user interface?
How was the Hyperbole Manual and other documentation?
Was the setup trivial, average or hard?
What areas of Hyperbole would you like to see expanded/added?
How does it compare to other hypertext tools you have used?
Was it easy or difficult to create your own types? Why?
Did you get any use out of the external system encapsulations?

Appendix C Questions and Answers

Missing Action Types

What if someone sends a mail message with a button for which I do not have the action type? Or a button whose link referent I can't access?

You receive an error that an action type is not defined or a link referent is not accessible/readable. This is hardly different than trying to get through a locked door without a key; you try the doorknob, find that it is locked, and then realize that you need to take a different approach or else give up.

Like all communication, people need to coordinate, which usually requires an iterative process. If you get a mail message with a button for which you don't have the action type, you mail the sender and request it.

Why is all the button data scattered across directories?

When you think of a hyper-space that you depend on every day, you don't want to have a single point of failure make you incapable of doing work. With Hyperbole, if some directories become unavailable for a particular time (e.g. the filesystems on which they reside are dismounted) you can still work elsewhere with minimal effect. We believe this to be a compelling factor to leave the design with localized button data storage.

Why are action types defined separately from their implicit button types?

Any category of button can make use of an action type. Some action types are useful as behavior definitions for a variety of button categories, so all action types are defined separately to give them independence from those types which apply them.

For implicit button types that require a lot of code, it is useful to add a module that includes the implicit button type definition, its action type definition and supporting code.

Appendix D Ongoing Work

This appendix is included for a number of reasons:

- to better allow you to assess whether to work with Hyperbole by providing sketches of planned additions;
- to direct further development effort towards known needs;
- and to acknowledge known weaknesses in the current system.

Button Copying, Killing, and Yanking

There is as yet no means of transferring buttons among buffers. We realize this is a critical need. Users should be able to manipulate text with embedded buttons in ordinary ways. Some of the Emacs Lisp functions that must be augmented to support this behavior include:

```
(append-next-kill)
(copy-region-as-kill)
(delete-region) ;; a built-in function
(get-register)
(insert-buffer-substring)   ;; a built-in function
(insert-file-contents)
(kill-append)
(kill-region)
(rotate-yank-pointer)
(set-register)
(view-register)
(yank)
(yank-pop)
(undo)
```

Hyperbole Views

These will permit rapid perspective shifting when studying and manipulating bodies of text. They will also allow fine control over the presentation of information displayed by action types. A more general, multi-media view model may be presented within the larger framework of our research on Personalized Information Environments (PIEs).

Global Buttons

Global buttons will be a new category of buttons much like explicit buttons but not embedded within buffers or files. They will be accessible at any point during Hyperbole use without the need to display anything to activate them. They will be activated by entering their names. There may also be a means of hierarchically grouping such buttons so that they may be more easily located and used.

Trails

Trails are an extension to the basic history mechanism presently offered by Hyperbole. Trails will allow a user to capture, edit and store a specific sequence and set of views of information for later replay by other users. Conditional branching may also be supported.

Storage of button data within button source files.

The current design choice of storing buttons external to the source file was made under the assumption that people should be able to look at files that contain Hyperbole buttons with any standard editor or tool and not be bothered by the ugly button data (since they won't be able to utilize the buttons anyway, they don't need to see or have access to them).

In many contexts, embedding the button data within the source files may be a better choice, so a provision which would allow selection of either configuration may be added. Here are some of the PROs and CONs of both design choices:

```
                      POSITIVE                        NEGATIVE

     Bdata in src file
                      Documents can stand alone.      All edit operators have
                      Normal file operations apply.   to account for file
                                                      structure and hide
                      Simplifies creation and         internal components.
                      facility expansion for
                      structured and multi-media
                      files.

     Bdata external to src file
                      Files can be displayed and      Currently, bdata for
                      printed exactly as they look.   whole directory is
                      No special display formatting   locked when any bdata
                      is necessary.                   entry is locked.

                      Button-based searches and
                      database-type lookup operations
                      need only search one file
                      per directory.
```

Relational Database Support Under Hyperbole

We have developed a basic in-memory relational database system that provides an initial means of structure manipulation within a hypertext node (a Hyperbole link button referent). Relations are permanently stored in files but each is read into memory as a whole before operations are performed, hence the *in-memory* descriptor.

A node can hold a single relation (a table of rows with dynamically sized columns (column size is fixed when the relation is displayed for viewing)) or a number of relations. A menu is used to operate on the database, e.g. to add or remove tuples (rows) or relations. The Smart Key can be used for direct manipulation of relation views. Standard relational Selects, Projects and Natural Joins can all be performed with direct manipulation so no special query language is necessary. (Emacs Lisp can be used for any procedural manipulation). For example, here is a relation view representing a simple filtering of the UNIX 'apropos help' command:

```
HELP-APROPOS.rsw
  ==============================================================
         NAME  SECTION                               DESCRIPTION
  ==============================================================
       Extent      3u  - help class that manages graphics info
         help       5                         - help file format
         help    LOCAL                   - an easy help facility
     help_open       1          - use help_viewer to open a file
   help_viewer       1   - SunView application and desktop help
   help_viewer       5                  - help viewer file format

A click of the secondary Smart Key on the "SECTION" label
removes it from the relation view:

HELP-APROPOS.rsw
  =====================================================
         NAME                               DESCRIPTION
  =====================================================
       Extent  - help class that manages graphics info
         help                         - help file format
         help                   - an easy help facility
     help_open         - use help_viewer to open a file
   help_viewer   - SunView application and desktop help
   help_viewer                  - help viewer file format

A click with the primary Smart Key on a tuple's NAME field
with NAME = "help" removes all but those records.  The secondary
Smart Key removes only the matching records).

HELP-APROPOS.rsw
  =====================================================
         NAME                               DESCRIPTION
  =====================================================
         help                         - help file format
         help                   - an easy help facility
```

(Right justification of fields is used right now. View specifications will allow any normal justification settings.) More complex queries will be performed by filling in argument templates.

Given the fact that entire relations are read into memory and that no query optimization is presently done, this setup may only be useful for databases of a few thousand records. Still, this solves many of the data management needs that people come upon day to day, with a minimal level of necessary expertise.

The same interface could be used to view and filter information from production relational database systems that Hyperbole could be hooked to by connecting an intermediary process that mediates between Hyperbole and the database manager (doing command and data format conversions). We have done things like this before and it is fairly straightforward.

NLS/AUGMENT-type File Structuring Being Investigated

NLS/AUGMENT treated files as a hierarchical set of nodes, for example, each paragraph would be treated as a node, with its own address. This would aid greatly in idea structuring and cross-referencing. It would also help us leverage off the years of thinking and development that went into that system within our own Personalized Information Environment work. Don't expect anything soon on this front, but keep in mind that we are interested in hearing about any recent related work.

Hyperbole Mail Interfaces needed for other Mail Front Ends

Most notable are the VM and MH-e front-ends that we would like to see support for but will not have time to produce for a while.

Forms-based Interfaces

This will allow one to create buttons more flexibly. For example, button attributes could be given in any order. Entry of long code sequences, quick note taking and cross-referencing would also be made easier.

Collaboration Support

From the early stages of Hyperbole design, collaborative work environments have been considered. A simple facility has demonstrated broadcast of button activations to a number of workstations on a local area network, so that one user can lead others around an information space, as during an online design review. (This facility is not yet ready for release.)

We shall do some work in specific collaborative mechanisms, but also expect that others who concentrate in collaborative work will provide more extensive capabilities.

Appendix E References

[Stallman 87] Stallman, Richard. *GNU Emacs Manual.* Free Software Foundation, Cambridge: MA, March 1987.

Key Binding Index

Code and File Index

Concept Index

A

B

C

D

E

G

H

Short Contents

Table of Contents