Inky: Internet Keywords with User Feedback

by

Victoria H. Chou

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author .....
Department of Electrical Engineering and Computer Science
February 1, 2008

Certified by .....
Robert C. Miller
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Inky: Internet Keywords with User Feedback

by

Victoria H. Chou

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The web today is accessed primarily through graphical user interfaces although some of its functionality can be more efficiently invoked through a command line interface. This thesis presents Inky, a sloppy command line for the web. This interface accepts commands that may be out of order, have missing arguments, contain synonyms or is otherwise sloppy and determines possible interpretations. It then presents these interpretations to the user, who can then pick one to execute. The interpretations are presented to the user as she types, providing responsive feedback on the state of the system. Inky uses specification files that indicate how to recognize a type or function in a command, how to display it to the user, and how to translate it into code. It also contains an interaction window and a keyword engine that determines the completions. This thesis argues that this interaction style is better for some functionality than the standard website interface. It also explores whether the sloppiness and feedback are useful in counteracting the negative effects of traditional command line interfaces.

Thesis Supervisor: Robert C. Miller
Title: Associate Professor

Acknowledgments

I would like to thank Rob Miller for being an incredible research advisor. Thank you for sharing your time and attention and for pulling me out of directional and technical doldrums with your high inspiration and deep technical expertise.

I would also like to thank Greg Little, Max Goldman, Darris Hupp, Kevin Su, and the other members of the User Interface Design group (and anyone else who sat through one of my design discussions) for your invaluable help, your scintillating conversations, and your willingness to subject yourselves to bad early prototypes. Thank you to my summer buddies, Lydia Chilton, Mariko Medlock, Prannay Budhraja, and Brandon Pung for making the summer productive and enjoyable.

Thank you, Maria Rebelo, for providing delicious snacks to us poor hungry grad students.

Finally, I'd like to thank my parents, Thomas Chou and Kuei-lin Tai, and my brother, Victor. Thank you for always being supportive of my academic endeavors and for providing constant love and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While the web is primarily a graphical user interface, some of its functionality can be more efficiently achieved through a command line interface. Users often go to a website to do a simple action they know how to do. For example, visitors to a travel website would know what kind of reservation they want to make, what times they want to travel, and where they want to go. Having been to such sites before, they also already know what information the site needs in order to provide the service the users want.

Under these conditions, the fastest way a user could convey this information would be to simply type it out for the website. Unfortunately, most websites only offer a menu or form based interface to access those services. As a result, even expert users are forced to parse the website's UI and painstakingly fill in each field (Figure 1-1).

To alleviate this problem, this thesis presents Inky, a command line interface for the web. It consists of a front-end window that can be invoked from a user's web browser. Users type keywords into the window that express what they want to do. For example, to search for flights from San Francisco to New York, some keywords might be `flight`, `san francisco`, `new york`, `search`, etc. These keywords, which form a keyword command, are processed by a back-end engine that creates possible interpretations of the command along with executable code for the interpretations. From the front-end, users can then execute the code corresponding to their desired interpretation. The window also provides feedback on the interpreter's state by showing

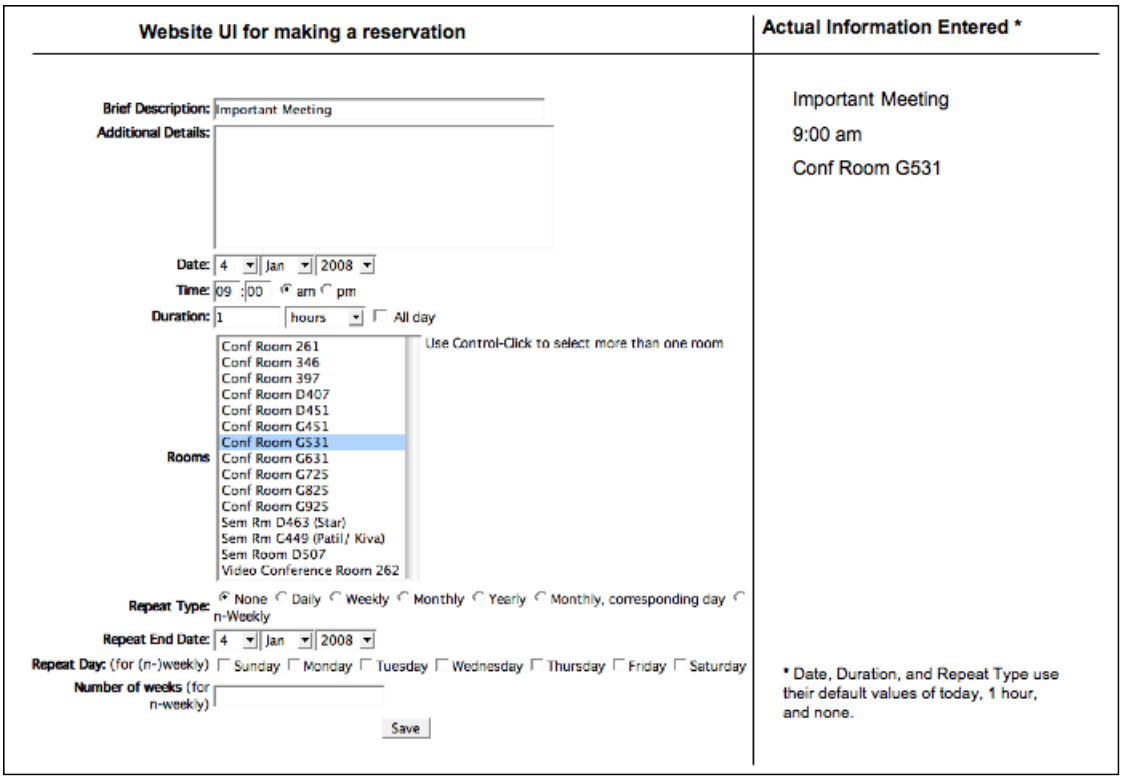

Figure 1-1: A comparison of a website's UI versus the information actually needed for the functionality.

users the top interpretations for the command as they enter it.

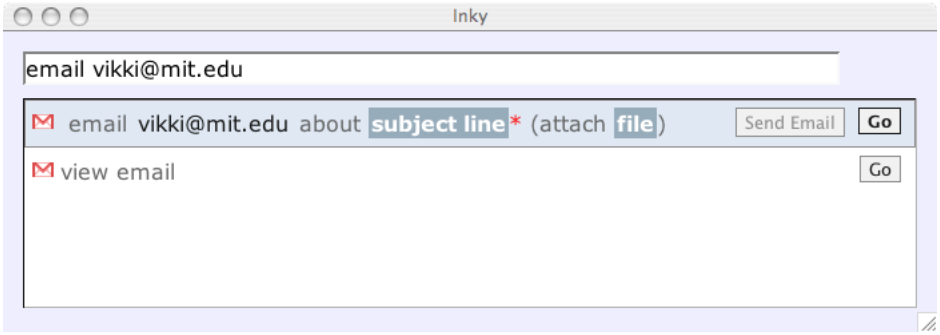

Figure 1-2: The Inky web command line interface.

By providing a command language for web functionality, Inky increases a user's efficiency by letting him bypass a website's interface. Instead of having to enter information into multiple fields or even having to go through multiple pages of fields, a user can just type all the information he has into Inky's command line. To get to the form shown in Figure 1-1, a user would only have to type `reserve g925 9am important meeting`. Even if a user just wants to go to a website and enter something into a single text box, the user has to remember the URL of the website or search for it with a search engine or through a list of bookmarks. With Inky, he just has to type in a word that relates to the website and what he wants to enter into that site's form, skipping the intermediate step. For example, if the user needed to define some term, instead of having to enter `http://www.dictionary.reference.com` into an address bar, waiting for the page to load, locating the site's search box, and submitting `fastidious` to it, a user could just type `define fastidious` into Inky and wait for the resulting page to load.

Using a command language also means users can communicate commands to others more easily as they will only have to send the others a single string instead of a list of things to do through a website's GUI.

Inky lets users create custom functions that could be an improvement over existing website functionality. Inky's custom functions are written in Chickenscratch, an extension to JavaScript that includes functions useful for manipulating webpages (such as entering text into a text field, clicking buttons, inserting content) and web

browsing (such as going to a website, opening a new tab, fetching a website in the background, etc) [1]. Through a custom function, Inky can access functionality that would require many steps in the original website. For example, if a website's directory search required the user to search for either first name, last name, or office number, a user could write a custom function that did all three searches and combined the results on a new page. He can then call that function from Inky with a single command. Since Inky's functions are related to websites, supporting Chickenscratch instead of just JavaScript makes it easier for users to write new functions.

Inky mitigates some of the classic pitfalls of command line interfaces. From a usability standpoint, command line interfaces are frowned upon because they force users to remember what they need to type. In contrast, graphical user interfaces display what functionality they have so users only need to recall what they want to do and how to navigate to it from the presented options [2]. Inky gets around this by being a sloppy command line and by displaying rich feedback to the user. For example, to get to the form shown in Figure 1-1, a user could type `reserve g925 9am important meeting`. He could also type `book g925 9:00 am important meeting` or `9am g925 important meeting reservation` or even just `9am important meeting g925`. This is because Inky accepts commands that use synonyms, have different keyword orderings, or are missing keywords. Synonyms for making a reservation include `reserve`, `book`, and `reservation` while synonyms for the time 9 am include `9am` and `9:00 am`. Ordering does not matter so `reserve g925 9am important meeting` and `9am g925 important meeting reservation` will both make a reservation at 9 am for room g925. Finally, even though `9am important meeting g925` does not contain a keyword for making a reservation, it works because Inky knows that the only functionality that requires a time, a room, and a description is making a reservation. Thus, Inky can still correctly interpret the command. Because Inky accepts these imprecise, or sloppy, commands, Inky is a sloppy command line. Accepting these sloppy commands greatly decreases a user's mental burden since the user doesn't have to remember exact names or orderings for commands.

Inky provides feedback that displays possible command completions as the user

types. It also displays descriptions for any arguments the user still needs to provide for each interpretation. Thus, users only need to recognize an interpretation from the feedback and read off the missing arguments. This again reduces a user's mental burden as she doesn't have to remember all of a function's arguments or a function's name. The feedback also helps prevent errors as it describes what will happen when a command is executed. This kind of visibility of the system's state is missing in most other command line interfaces. Inky's sloppiness and rich visual feedback differentiate it from previous web-related command lines.

Inky has been implemented as a Firefox extension built on top of Chickenfoot [1]. It contains a keyword interpreter that takes sloppy commands and returns possible command interpretations. A command interpretation includes a way to visually present the interpretation to the user and executable Chickenscratch code for the interpretation. Inky also contains XML files that describe the functions a user can call and the types that those functions use.

The rest of this thesis details the design and implementation of Inky. Chapter 2 describes related work in the area of command line and keyword interfaces. The design of Inky's user interface is discussed in Chapter 3. Chapter 4 details the implementation of Inky's keyword engine, how the system defines types and functions, and how types and functions are specified as XML. Design considerations for the XML files are also discussed in that chapter. Chapter 5 presents data from evaluations of Inky and discusses some of the ramifications of the results. Chapter 6 concludes with future work that can be done on this project as well as a review of its contributions.

Chapter 2

Related Work

This chapter presents various existing command line and keyword interfaces. How these interfaces relate to Inky is also discussed.

2.1 Unix Shell

The Unix shell is an example of a standard command line interface that reads in text and converts it into calls to functions or programs [3]. In its command language, a one-word name of the function always appears first, followed by any number of arguments to that function. The shell then finds the correct function and executes it, passing the arguments it received to the function. These functions usually require the arguments to have been typed in a specific order. The Unix shell also let users chain commands together by passing the output of one command as inputs to the next command. However, these command chains are linear so a later command can only use the output from a single previous command. The Unix command line is a very efficient way to access a lot of powerful functionality. However, it is difficult for novices to use since one needs to know the exact name of a function to call it, as well as what arguments it needs and in what order. When it is given an incorrect command, the shell simply tells the reader that the command could not be found, making it hard for users to recover from that error. If users call a command with the wrong arguments, the shell could give other error messages or even run a command

that was different from what the user intended. Inky tries to alleviate these problems by showing users the state of the interpreter and by giving the users suggestions for alternate commands.

2.2 YubNub

YubNub is a social command line for the web [4]. Its main site has a single text box where users can enter in a command as well as some example commands. YubNub also has a Firefox plugin that puts an equivalent text box at the bottom of users' Firefox windows. YubNub's functionality is similar to that of many traditional command lines. Like a Unix command, a YubNub command consists of a function's name followed by arguments to that function. While the arguments do not have to be typed in order, if a function has multiple arguments, the user has to specify the name and value of each argument. For example, `xe` is a function that calculates an exchange rate. It takes in a number, a starting currency, and an exchange currency. A command that calls that function is `xe -amount 1 -from USD -to EUR`. Arguments can also have default values and the results of function calls can be passed as arguments into other function calls.

To help novices learn how to use YubNub, there are extensive lists of example commands on its main site. It also supports the functions `ls`, which brings users to a webpage that lists all the YubNub commands, and `man` which takes in a function name and displays what that function does and some example calls for that function. Inky is more flexible than YubNub since it can take in sloppy commands. However, to get this added flexibility, users have to define synonyms and types in Inky. Inky also provides access to more powerful functionality since the backend uses Chickenscratch instead of HTTP requests. Finally, like the traditional command lines it is based on, YubNub does not provide users with any feedback on what their command will do when it is executed.

2.3 Search Fields

Text search fields provide functionality based on text it is given. In many cases, the search field used represents some function and the text typed in it represents the argument. For example, if one goes to Dictionary.com and types `fastidious` in their search box, one is effectively calling a `get definition` function on the argument `fastidious`.

Firefox's Smart Keywords [5] builds off this by letting users associate keywords with certain text fields. Afterwards, users can type that keyword into the Firefox address bar followed by the text to put into that search box. This gives users a quick way to access the search box's functionality. Keywords can also be associated with bookmarks or bookmarklets [6]. Smart Keywords still requires users to remember what keyword they chose for each function. It also cannot deal with functions that take in multiple arguments, such as forms with multiple text fields.

Specific functionality from keywords is also becoming available from Google search boxes [7]. For example, typing `weather` and then a zipcode or a city and a state will bring up a small box at the top of the search results that display the location's current weather and the forecast for the next three days. Google also provides information about stocks from a stock symbol, maps from an address, and travel information given a flight or an airport code followed by the word `airport`. Google searches are very flexible in that it can correct for some unordered commands and commands with synonyms or misspellings. If a user wants specific information that is not provided through Google, the list of search results it returns may include a link to another website that has the information the user wants.

2.4 Sloppy Commands

One of Inky's primary features is its ability to take in sloppy commands. An implementation of this was done as keyword commands in Chickenfoot and in Microsoft Word [8]. Keyword commands have no syntax so the ordering of its keywords does not

matter. Commands can also have synonyms and nested functions. This work showed that keyword commands are intuitive and could be useful in the web domain. It also exposed that misinterpreted commands were a major issue. Inky attempts to solve this by making the state of the interpreter more visible to users. In Chickenfoot's keyword commands, context played a role in how commands were interpreted. As a result, some misinterpretations happened because the website context changed. Inky generally will not have that problem as its commands are independent of context.

Another implementation of sloppy commands translated keywords into Java code [9]. This work gave algorithms for translating Keyword Commands into Java code and evaluated the algorithms. The new algorithms worked for APIs of over 2000 functions whereas previous implementations were limited to 20 or so functions or to a constrained context. Inky's engine for translating commands into code uses a slight variation the bottom-up algorithm presented in this work. The few changes make the algorithm a bit slower but help it provide more accurate interpretations.

2.5 Koala

Sloppy commands for the web appear in Koala [10]. Koala lets users create scripts of sloppy commands that can be played back or shared with others. These commands are all tied in to actions on webpages such as scrolling, clicking buttons and links, and entering text.

Koala gives the user feedback on the interpreter by displaying a green box around the website element that an action would be performed on and bolding the part of the command that was used by the interpreter. It also provided the user with a list of alternate interpretations in text form. If none of those interpretations were correct, the user could edit the command or specify that a certain command needed to be done by a human.

As Inky is made to execute only one command at a time, users cannot easily write scripts for Inky as they can for Koala. However, since Inky commands are only typed once and not read in from some source, it can give visual feedback to

the user dynamically as the user types. This dynamic feedback is primarily textual and is similar to the feedback provided for the alternate interpretations in Koala. Inky also provides this feedback for the primary interpretation. It has been shown that providing a user with sentences describing the state of a configuration makes users more confident about what they are doing and perhaps lets users work more quickly and accurately [11]. As arguments in an Inky command are effectively a way to configure a function, it makes sense to use a similar approach in Inky as the users type the original command.

Inky has to translate commands into Chickenscratch and store the Chickenscratch code in its interpretations. In contrast, the human-readable text is the script in Koala and no other information is stored. Inky provides access to a wider range of functionality than Koala as it isn't limited to standard website actions.

2.6 Quicksilver

Application launchers and shortcut applications are other examples of text-driven user interfaces. These programs take keywords for files or applications and open them. Several also have rich user feedback.

One such application is Quicksilver, a shortcut application [12]. Users start by giving it some subject, which can be a folder, an application, or a file. Then users provide an action to do on that subject. Finally, if the action requires an object, the user will provide that as well. For example, if the chosen subject were an HTML file, acceptable actions might be to view the webpage in a web browser or edit it in an editing program. If one were renaming a file, the object would be the new filename.

Quicksilver does subsequence matching for all its arguments and is case-insensitive. For example, one could type `inky`, `iky`, or even `ny` to indicate the file `Inky.doc`. This allows users to type in shorter names for subjects, actions, or objects and thus be more efficient. It also makes the keywords robust against some spelling errors.

Quicksilver also provides rich, dynamic feedback to the user. As the user types in a subject, action, or object, Quicksilver shows the current best match along with

the possible other matches for what has been typed so far. As a result, as soon as the user has typed enough for the top match to be what he wants, he knows he can move on to the next argument, saving time and typing. Quicksilver also limits and biases the possible matches based on what arguments it already has. Once the user has chosen a subject, the action field has a default action for that subject along with a list of possible actions that can be done on that subject (Figure 2-1). This way, if the user simply wanted to perform the default action, he wouldn't have to type anything extra. Furthermore, if he cannot remember the name of his desired action, he can just select it from the list of possibilities.

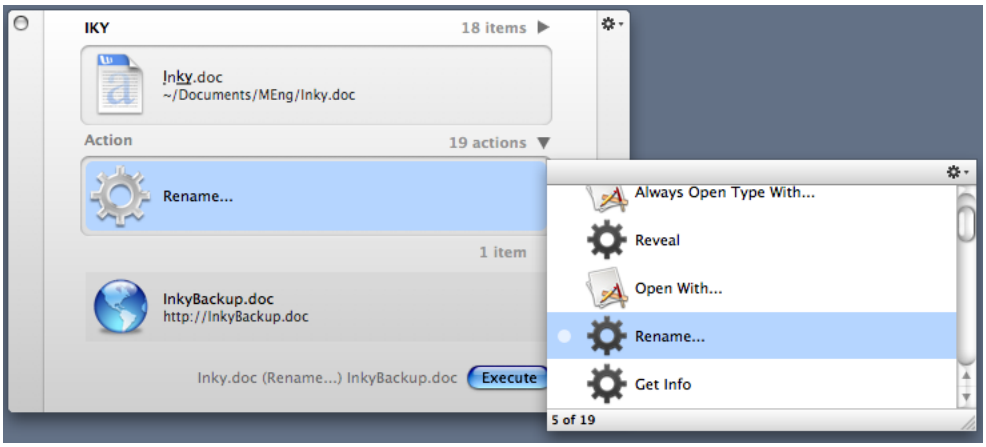

Figure 2-1: The Quicksilver interface.

By providing a list of possible subjects and actions, Quicksilver allows users to recognize what they want to do instead of having to recall it. While it still requires some small amount of syntax (subject, action, object), there are labels in each box specifying what goes where. Its subsequence matching further reduces a user's mental load by limiting what he has to remember in his head. However, subsequence matching does not help users who remember a different name for a subject that is not a derivative of the correct name. For example, if there were a program that looked up people's addresses called "Address Finder", `location lookup` would not match that application in Quicksilver.

Quicksilver supports plug-ins that add new subjects and actions.

Chapter 3

User Interface Design

The goal of Inky is to provide access to website-related functionality with the efficiency of a command line interface while mitigating the traditional pitfalls of that interaction style. Inky models website-related functionality as functions and Inky users should be able to specify a function to call as well as any arguments that function needs to run. In order to deliver this efficiency, Inky can be used entirely from the keyboard, although it also accepts mouse input. Inky displays visual cues that teach new users how to execute commands. It also finds and displays possible command interpretations to the user so the state of the system and the consequences of a command are visible to the user. The flexible nature of the keyword language also makes it easier for novices and infrequent users to use the interface.

3.1 Overview of the Layout

Inky appears in a browser as a pop up window. It is designed to be compact so that users can view websites behind it and reposition it if it is blocking anything they need to see.

Inky has two main areas: a text field for a user to input commands and a feedback area that displays the state of the command interpreter to the user (Figure 3-1).

Clicking on the browser dismisses Inky, as does hitting the Esc key, since Inky assumes that the user wanted to cancel the command. Since Inky is easy to start up,

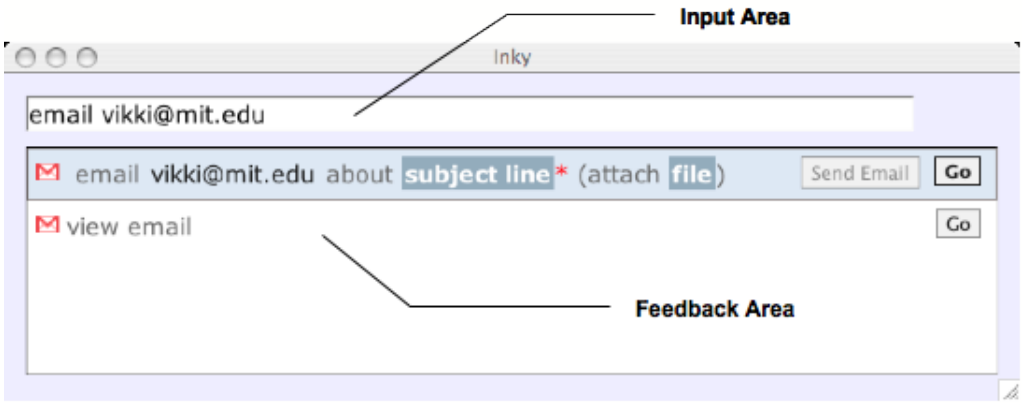

Figure 3-1: Inky's input and feedback areas.

if such a dismissal was an error, it would be easy to undo.

3.2 Startup

Users type in a keyboard shortcut in order to start up Inky. Although it is customizable, the default shortcut is Ctrl+Space, which can be typed with one hand and is consistent with other keyword interfaces such as Spotlight in Mac OS X and Quicksilver. New users can also start up Inky through the Tools menu.

The startup screen asks the user to type in a command and gives an example of such a command (Figure 3-2). This example command changes each time the window pops up. It also changes whenever the text field is blank after the user has typed something. As a result, this example command teaches new users how to get started and also slowly teaches them what commands are available to them.

Figure 3-2: Inky's display upon start up.

3.3 User Inputs

Users input text commands into a text field in order to access website functionality. On startup, the focus is on the text field so that users can start typing immediately. As they type, their command is sent to the keyword engine.

Commands contain function keywords that indicate the website functionality the users want to access as well as argument keywords that indicate what arguments to give to that function. For example, in order to reserve the room D463 at 3pm through the CSAIL web form, a user might type `reserve D463 3pm`. The `reserve` keyword indicates that the user wants to make a reservation through the CSAIL web form and `D463` and `3pm` are arguments to that function.

To make commands easier for novice and infrequent users to call, the order of the keywords entered does not matter. For example, `reserve D463 3pm` and `D463 3pm reserve` will result in the same interpretations. Keywords that represent arguments to a function can be switched around and interspersed with keywords specifying the function to be called. The order of the entered command only matters when the user is calling a function where two or more of its arguments could use the same set of keywords. For example, in the command `reserve D463 3pm 1 2 2007` it is unclear if `1` is the month and `2` is the date or vice versa. Thus the function has two arguments that could use the same set of keywords. This might also happen if the function requires two arguments of the same type such as `find flights SFO LAX`. Here, the user wants to search for flights between SFO and LAX. The function that looks for flights requires two airports, but the user did not indicate which airport was the starting airport and which was the destination. Under these circumstances, the system will give a higher rank to the interpretation that uses the keywords in left to right order for its arguments. However, the other orderings are still considered valid interpretations. How users choose alternate interpretations to run is discussed in the "Executing Commands" section of this chapter. Making the commands order independent makes the system easier for novice and infrequent users as they won't need to memorize argument orderings in order to access the desired functionality.

Inky commands can also use synonyms for both function keywords and argument keywords. To reserve D463 at 3pm, the user could also have typed `make reservation 32-D463 3:00pm` or `star room 15:00 csail reserve`. Here, "Star Room" is an alternate name for room D463, which is also referred to as 32-D463. Keywords for the time of the reservation and for the reservation function also have synonyms. These synonyms make Inky easier for novice and infrequent users to call functions since they do not need to remember exact names for functions or formats for function inputs.

Finally, given certain arguments, the system can guess what function the user is trying to call. For example, if the csail reservation function is the only one that uses both a time and a room, just typing `32-D463 15:00` would be enough to call the csail reservation function with those arguments. This improves the efficiency of Inky users.

3.4 User Feedback

Inky displays a list of command interpretations to improve system visibility and allow users to recognize commands once they have recalled enough for the system to guess what they are trying to do. An interpretation displays a function, the arguments a user has already filled in, any arguments a user can still fill in, and whether the function has a persistent side effect. These interpretations are updated as the user types in order to give continuous feedback (Figure 3-3). In order to provide these updated interpretations, the whole command is reinterpreted each time the user adds or removes a character. Words that do not fit into any interpretation, such as partially completed words at the end of a command, are ignored.

Each interpretation has visual cues that indicate its functionality and the arguments that have been found for it. A small icon indicating which website the function comes from appears next to the interpretation for quick scanning. The icon is followed by some text describing the function and what the function does with each of its arguments.

To the right of the description are some buttons that run the interpretation's cor-

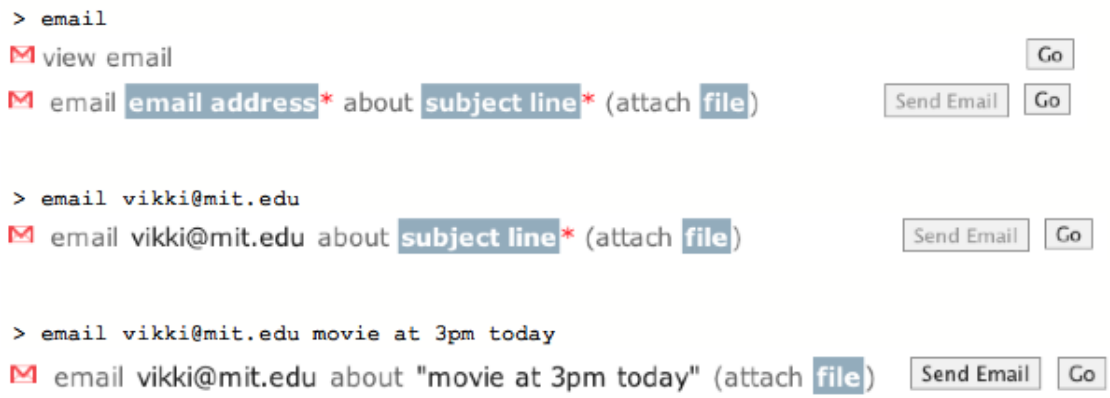

Figure 3-3: Inky's feedback as input progresses.

responding code. Most interpretations only have a "Go" button after the description. However, functions with persistent, difficult-to-undo side effects have an extra button with the name of the side effect on it. For example, in Figure 3-3, since emailing has the persistent side effect of sending an email, a button with that label appears by that interpretation. Since viewing emails does not have a side effect, no such button appears by that interpretation. The extra side effect button stands out to the users, drawing their attention so they think more before calling such a function. It is enabled only after a users has entered in all the required arguments to execute the function's side effect. In Figure 3-3, the "Send Email" button is enabled only after the user entered an email address and a subject line. We discuss what the "Go" button and side effect buttons do in the "Executing Commands" section of this chapter.

Inky displays feedback for each argument in a function indicating whether it has been filled in, whether it has a default, and whether it is required or optional. If a user has typed in a keyword for an argument, a version of the entered argument appears in the user feedback. This is usually exactly what the user typed, but can also be a standardized version of what the user typed. For example, if the command was `reserve star room 15:00` the user feedback might be "reserve D463 at 15:00". Inky changed `star room` to "D463" to indicate that it parsed `star room` as a room and knows that it corresponds to room D463.

If a user has not yet typed in a keyword for an argument, the name of the argument appears as white text in a dark box. The name is usually a word or phrase that

describes what argument is missing. This way, users can see if they are missing an argument and what argument they still need. If a missing argument has a default value, the name of the default value appears normally and there is only a faint outline against it. The less noticeable visual cue indicates to users that while they do not need to add in an argument, they could change the default with an appropriate keyword if they chose to. In Figure 3-4, `name, email, office, etc` is a missing argument while `month` is an argument with the default value of `this month`.

For functions with persistent side effects, Inky makes a distinction between arguments that are required for the side effect to take place and those that are not. Required arguments that have not already been filled in by the user appear with a red star next to them. In Figure 3-4, `room` and `description` are required arguments. Arguments with defaults are not required arguments since the function should simply use the default value.

Figure 3-4: Examples of argument feedback.

Functions without persistent side effects do not have required arguments in Inky. The reasoning behind this is discussed in the "Executing Commands" section of this chapter.

For both types of functions, Inky separates out optional arguments that users generally won't want to fill in. These are grouped, usually at the end of the line, with parentheses around them. This way, users can more easily ignore these rarely used functions but can still access them see any defaults they have if they want to change them.

In summary, arguments in Inky can be either required (but only for functions with persistent side effects) or optional. Optional arguments can then either have a default or not. Independently, optional arguments could also be rarely used or not. Examples of different kinds of arguments are shown in Table 3.1.

By providing visual feedback on how a user's command is interpreted, Inky makes

	Required?	Default	Rarely Used?
Make Reservations			
room number	yes		
description	yes		
start time	no	now	no
date	no	today	no
repeats	no	never	yes
Search flickr			
search term	no	No default	no
Send email			
email address	yes		
subject	yes		
file to attach	no	No default	yes

Table 3.1: Examples of different kinds of arguments.

the system state more visible to the users. Knowledge of the system state also reduces the possibility that a user would enter and execute an incorrect command. Finally, by listing several possible interpretations, the system allows the user to recognize a correct interpretation and possibly discover new functions that partially match the function they wanted to call.

3.5 Executing Commands

Users run a command by hitting Enter or an appropriate button on the Inky interface. By default, Inky assumes that the top suggestion is the correct interpretation. However, the user can use the arrow keys or the mouse to select a different interpretation from the suggestions (Figure 3-5).

The system differentiates functions that have persistent side effects from those that do not. A persistent side effect is loosely defined as a side effect that cannot be easily undone (e.g. sending an email, buying a book) or that changes the state of

some system with persistent memory (e.g. making a room reservation, uploading a photo to an online gallery).

For functions without side effects, hitting Enter will run the function with the given arguments. Users can also run the command by hitting the Go button next to the correct interpretation. Since there are no side effects and users running these commands just want to view information, these commands can run with as many or as few arguments as the user chooses to give it. If the function is missing some desired arguments, the website behind the function will fill in defaults for the missing arguments or prompt users for what it needs. By delegating these tasks to the website, Inky is able to access defaults that are stored by websites in difficult-to-access formats. AccuWeather.com stores a user's default city for weather forecasts using HTTP cookies. By letting the website deal with this default, a user can look up the weather in their default area just by executing the command `weather`. Letting the website handle certain missing arguments also allows them to use private information to determine what arguments are actually necessary. For example, if a user executes the command `travelocity flights june 30 july 7`, they will be taken to the Travelocity website's flight search form. The website will then prompt the user for a departure airport and an arrival airport. However, if Travelocity stored a user's most common departure and arrival airports, it would fill in those fields in the form as defaults and not prompt the user. A website's personal prompt may also include useful UI elements that are not available through Inky. If a user wanted to look at Travelocity flights between SFO and LAX but wasn't sure what dates he wanted to fly on, he could execute the command `travelocity SFO LAX`. Travelocity would prompt him for departure and return dates with a calendar widget that will give him more information on when certain dates are and when there are flights that can be booked.

Functions with side effects have two modes of execution: a view mode and a submit mode. Running a function with side effects in the view mode means that the user wants to set up the form that will cause the side effect, but not actually submit the request. This is done by hitting Enter or pushing the Go button next

to the correct interpretation. For example, when a user runs `email vikki@mit.edu remember to buy milk today`, he will be taken to an e-mail form with the To field and the Subject field filled in. He can then press the Send button to send the e-mail. In this execution mode, Inky deals with missing arguments the same way it deals with them for commands without side effects.

Running a command in submit mode means that the user actually wants to make the command's side effect happen. This is done by typing Ctrl+Enter or pushing the button with the side effect's name next to the correct interpretation. Users can only run a command in this mode if keywords for all the required arguments for the function have been entered. Running `email vikki@mit.edu remember to buy milk today` in submit mode will actually send an e-mail to vikki@mit.edu with the subject "Remember to buy milk today". Users will still see the site after the side effect so any confirmation pages would still be visible.

Separating command executions that commit side effects from those that do not discourages Inky users from making errors that would be difficult to reverse. Since the default execution methods always execute the command in a view mode, it is more difficult for a user to unknowingly cause a persistent side effect. However, by making it possible for users to execute in a submit mode, Inky increases the efficiency of users who know the system and want to commit a given side effect.

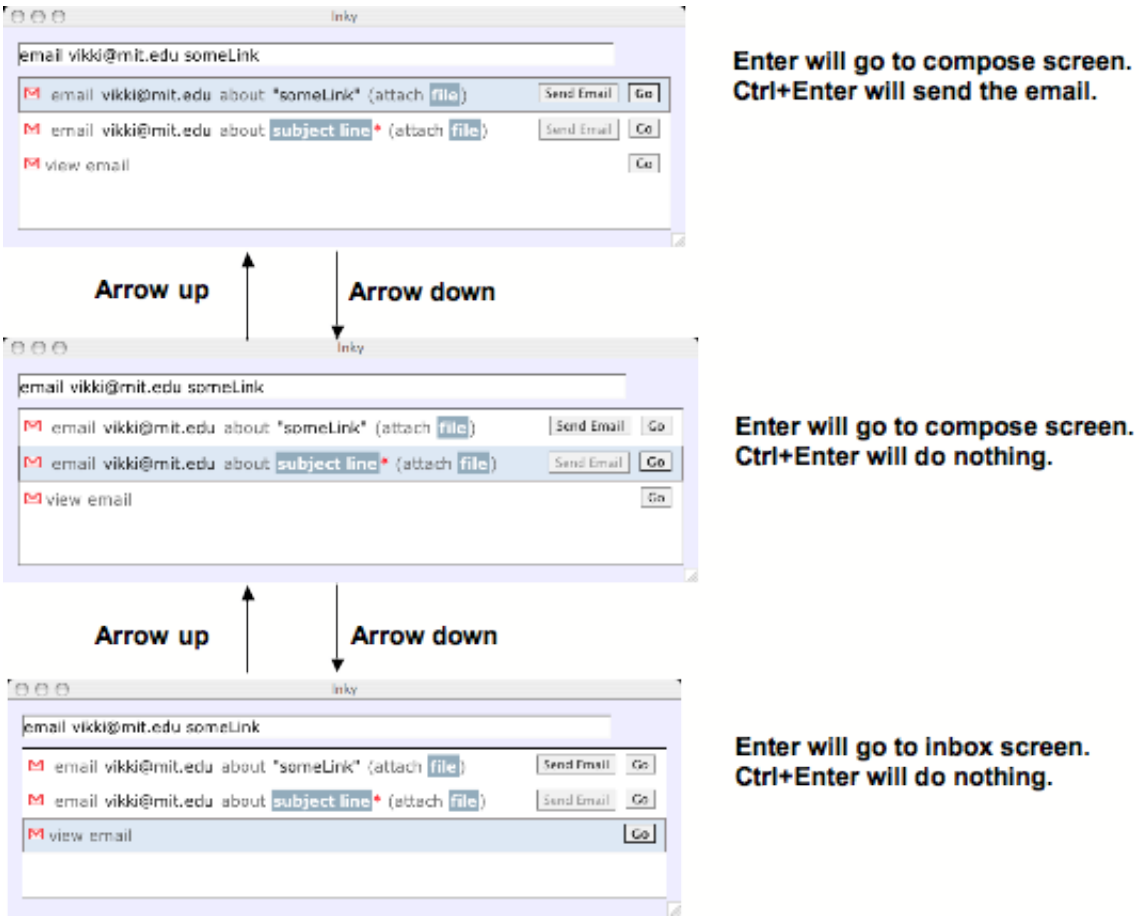

Figure 3-5: How Inky commands are selected and executed in run and submit modes.

Chapter 4

Implementation

Inky is a Firefox extension built on top of Chickenfoot and created with Chickenfoot's extension packager. The application includes the Inky window discussed above, a back end keyword interpreter, and XML files specifying the functions and types in the system.

4.1 Overview

This section provides an overview of Inky's components, which are discussed in more detail later in the chapter.

The three main parts of Inky are the Inky window, the keyword interpreter, and the files that specify the system's types and functions.

The Inky extension adds a Chickenfoot trigger that opens an Inky window when the user types the correct keyboard shortcut (Ctrl+Spacebar). The window is implemented with JavaScript and CSS. This window passes a user's commands down to the keyword interpreter and displays the resulting interpretations to the user. Once a user has chosen to execute a particular interpretation, the Inky window runs that interpretation's corresponding Chickenscratch function in the Chickenfoot environment, with respect to the currently visible tab. This part of the system also takes care of running other functions used for keyword interpretation that need to be run in the Chickenfoot environment.

The keyword interpreter takes in a command from the user and returns an ordered list of possible interpretations of that command. To do this, it evaluates whether each type and function it knows about is likely to exist in a given command. It goes through multiple cycles of evaluation in case the command has nested function calls. Finally, if no good solution is found, it tries some variations of the given command in an attempt to find a suitable interpretation. The interpreter is implemented in Java.

The functions and types that the keyword interpreter understands are specified in human-editable XML files.

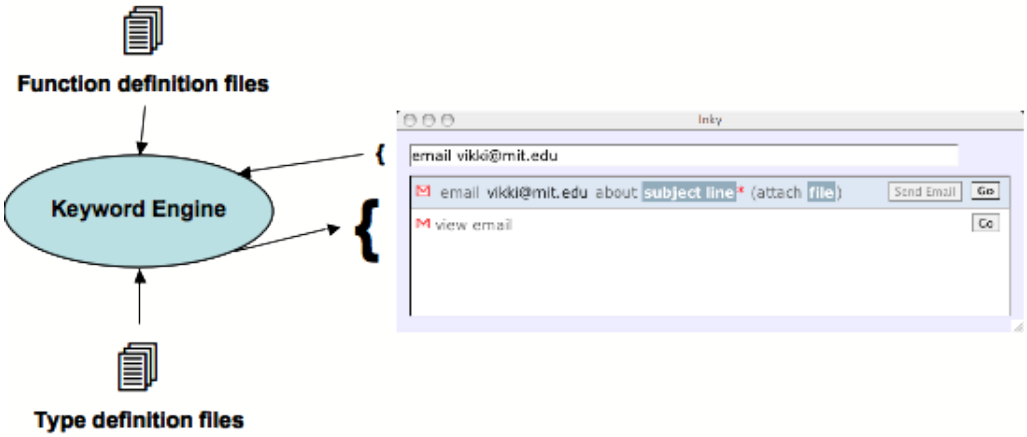

Figure 4-1: A block diagram of Inky's major components.

4.2 Representation of Types

The system uses the notion of types to help it interpret commands. A type represents a class of thing that can be used or created by a function. For example, a day of the week, an airport, or an age are all examples of types. A type has a name so that functions can specify that they use it. The system also needs to be able to recognize whether part of a command matches a type. For example, the system should know that `may`, `april`, and `march` match the month type. Given a command fragment that matches a particular type, the system also needs to be able to translate the string both into a version that is readable to humans and into the version that will ultimately be passed as an argument to the Chickenscratch function that uses the

type. Given `may`, the system might want to change it to "May" for the user so that it is correctly capitalized and `5` for the code so that the function reads it correctly.

Inky comes with some predefined utilty types. These include email address, string, state, time, zip code, month, and day of the week.

VOID is a special type in the system that represents the absence of a type. It is used as the output type of functions that do not have a return value.

4.2.1 Internal Representation

In the keyword interpreter, all the functionality related to a particular type is stored in a Type object. Each Type object stores the type's name, a ScoreDeterminer, a Generator for code, and a Generator for human-readable feedback.

A ScoreDeterminer takes in an input string and outputs a collection of ScoreVectors. Each ScoreVector represents a possible occurrence of the Type in the input string. It also indicates which words in the input string this occurrence uses.

For example, say a user wanted to recognize different soft drinks and some options were coca cola, pepsi, sprite, and 7up. She would create a Type with the name "soft drink". If she passed the string `I like coca cola more than pepsi` to its ScoreDeterminer, she would get a ScoreVector of 0,0,1,1,0,0,0 for the recognized occurrence of `coca cola` and another of 0,0,0,0,0,0,1 for the recognized occurrence of `pepsi`.

It is necessary for the ScoreDeterminer to return a list of ScoreVectors or else it would be difficult to determine whether a string contained a single instance of a type that happened to span multiple words or whether each word was a separate instance of that type.

A Generator for a Type takes in a string that is assumed to be an occurrence of that Type and translates it into a different string representation of that Type. To get the input string for a Generator, the system takes the original command and a ScoreVector given by the Type's ScoreDeterminer and concatenates all the tokens that got a positive score.

VOID is a special static Type with no ScoreDeterminer or Generators.

4.2.2 XML Specification for Types

Types are specified in XML files. A type XML file starts and ends with the `webCommandMappingTypes` tag. Each file contains zero or more `type` elements. A `type` element has two attributes: a `name` and a `category`.

Standard Types

Most types have a `category` of `standard`. Types of this category specify a `members` element that describes the type's ScoreDeterminer, a `codeRep` element that describes the code Generator, and a `userLang` element that describes the type's human-readable feedback Generator.

The `members` element specifies the type's ScoreDeterminer and contains a `category` attribute whose value is `enum`, `contains`, `regex`, or `js`.

- `enum` – If the category is `enum` the `members` element will have a `body` attribute that contains a comma-separated list of all the possible members of the type. Each of these members corresponds to exactly one possible instance of the type. For example, the ScoreDeterminer created by `<members category="enum" body="am,pm">`, given the command `foo am pm`, should return ScoreVectors of 0,1,0 and 0,0,1.

- `regex` – If the category is `regex` the `members` element will have a `body` attribute that contains a regular expression that matches an instance of the type. For example, the ScoreDeterminer for an email address is given by `<members category="regex" body="\w*@\w*\.\w*"/>`. If it is given the command `vikki@mit.edu rcm@mit.edu`, it should return ScoreVectors of 0,1 and 1,0.

- `js` – If the category is `js` it means the ScoreDeterminer is a custom one written in JavaScript. The text content of the `members` element will contain a CDATA block with the JavaScript function in it. This function takes in an input string and outputs an appropriate list of ScoreVectors. The function can also be

written in ChickenScratch and is run in the Chickenfoot environment. Thus, if a type needed to access some web service to recognize its members, it would be able to do so.

The `codeRep` and `userLang` elements specify the type's code Generator and user feedback Generator, respectively. They each have a `category` attribute which could be `self`, `map`, or `js`.

- `self` – A `self` generator would concatenate each of the recognized tokens in the given ScoreVector into a string. For example, if a `self` generator got the command `reserve g925 3 pm` and the ScoreVector 0,0,1,1, it would concatenate the last two words into the string `3 pm`.

- `map` – A `map` generator has an `ins` attribute that is a comma-separated list of possible unique inputs and an `outs` attribute that is a corresponding comma-separated list of outputs. The generator, given the input string and a ScoreVector would then extract the "self" string and see if it matched any of the unique inputs. This matching is case-insensitive. If it did, it would return the corresponding output. For example, if the generator corresponding to `<userLang category="map" ins="coca cola, pepsi, sprite" outs="sodaA, sodaB, sodaC">` got the command `buy Coca cola` and the ScoreVector 0,1,1 it would return `sodaA`.

- `js` – For a `js` generator, the element's text content would be a CDATA block with a JavaScript or Chickenscratch function that defined the generator. The function would take in an input string and a ScoreVector and output an appropriate string. For example, in the city type, there is a `js` generator that removes the comma that people sometimes put after cities. The `codeRep` element for that type was defined as:

```
<codeRep category="js" >
<![CDATA[
function(inputWords) {
    // remove commas and quotation marks
```

```
        inputWords = inputWords.replace(/"/,"");
        return inputWords.replace(/,/ig ,"");
    }
    ]]>
    </codeRep>
```

While all types need to have a unique ScoreDeterminer, most types use a `self` code generator and a `self` user feedback generator. As a result, these are considered the default code and feedback generators if the `codeRep` or `userLang` elements are missing. Here are some examples of `standard` type elements:

```
<type name="time" category="standard">
  <members category="regex"
           body="(([1-9]|10|11|12) ?(am|pm))|(([1-9]|10|11|12):[0-5][0-9] ?
                 (am|pm))|((1?[1-9]|2[0-4]):[0-5][0-9])"/>
  <codeRep category="js">
     <!--puts the input into 24hour time format-->
     <![CDATA[
     function(inputWords){
         // get time from sides of the colon
         var colonInd = inputWords.indexOf(":");
         var hours = 0;
         var minutes = 0;

         if (colonInd > 0) {
             hours = parseInt(inputWords.substring(Math.max(0, colonInd-2),
                                                   colonInd), 10);
             minutes = parseInt(inputWords.substring(colonInd+1,
                          Math.min(inputWords.length, colonInd+3)), 10);
         } else {
             // parseInt automatically ignores letters
             hours = parseInt(inputWords, 10);
         }

         // if pm exists, increment the hour by 12.
         if (inputWords.indexOf('pm') != -1 && hours < 12) {
            hours = hours + 12;
         }

         // format the string correctly
         if (hours < 10) {
             hours = "0"+hours;
         }
```

```
        if (minutes < 10) {
            minutes = "0"+minutes;
        }
        return (hours + ":" + minutes);
    }
    ]]>
  </codeRep>
  <userLang category="self"/>
</type>

<type name="zipcode" category="standard">
<members category="regex" body="[0-9]{5}"/>
</type>
```

Simple Map Types

Another possible value for a type's `category` attribute is `simpleMap`. A `simpleMap` type is a type that has a set number of abstract members that can be recognized in many ways. One could define a `simpleMap` type as a `standard` type with `enum` members and `map` code and user feedback generators. However, the format for them here makes it easier for editors to edit, add, and remove members of the type.

A `simpleMap` type contains zero or more `item` elements. Each element contains three attributes: `codeRep`, `members`, and `userLang`. Each `item` element represents an abstract member of this type. The `codeRep` is how that member should be expressed in code and the `userLang` is how that member should be shown to users. There is exactly one way for an abstract member to be translated into code and exactly one way for it to be translated into a user-readable format. However, there should be multiple ways to recognize this member. As a result, the `members` attribute is a comma-separated list of ways to recognize this item and acts as an enumeration of ways to get this item.

Here is an example of a `simpleMap` specification:

```
<type name="room" category="simpleMap">
<item codeRep="1" members="261" userLang="261"/>
<item codeRep="2" members="346" userLang="346"/>
<item codeRep="12" members="G449,patil,kiva,patil/kiva" userLang="G449"/>
```

```
<item codeRep="13" members="D463,star" userLang="D463"/>
<item codeRep="14" members="262,video conference" userLang="262"/>
</type>
```

Intermediary Types

The final possible value for a type's `category` attribute is `intermediary`. These types are types that are never recognized directly from a token but are instead output types of functions that are then used in other functions. For example, the system includes an `intermediary` type named "full date". It is defined with `<type name="full date" category="intermediary"/>`. This type corresponds to the JavaScript Date object. The system contains functions with an output type of "full date" that create these Date objects. These Date objects are then used by functions in the system that have an input with type "full date".

4.3 Representation of Functions

The system uses the notion of functions to represent a piece of website related functionality. The system needs to be able to recognize when a user's command corresponds to a particular function call. To that end, the system needs to know keywords that cue a function as well as the argument types of the function. Inky also needs to know how to display the function to the user and how to execute the function.

Functions the users execute must have a return type of VOID. This is because the web-based functionality provided by Inky should result in some change in the browser or website. As a result, executable Inky functions cause some action to happen and do not return a value.

While functions the users execute must have a return type of VOID, Inky also accepts functions with other return types. These functions are used to create intermediate arguments to a final executable function. This makes it easier for users to define argument types that are variations of existing types. For example, say a function has a date argument. There is already a date type, but the user wants the function to accept the token `today` as well. To support this, the user could create a new function

with a return type of date that had the keyword `today` and that returned an appropriate representation of the corresponding date. If there was an executable function that needed the date to be in a special form, the user could create a new function with no keywords that took in a date and returned the date in the special form. Since the code for these functions is evaluated in the Chickenfoot environment, allowing intermediate functions also lets users define intermediate arguments that access the browsing environment.

4.3.1 Internal Representation

In the keyword interpreter, a function is stored as a Function object. Each Function object stores the function's output Type, an ordered list of input Arguments, a ScoreDeterminer, the Chickenscratch body, whether it has a side effect, and data related to displaying the function to users.

Each Argument object stores the Type of the argument; whether it is required, optional without a default, or optional with a default; and how it should be displayed to the user when it is not filled in. By default, this display is the name of the type of the argument. However, it could also be a more descriptive name for the argument or the name of the argument's default. For example, if a function had an argument whose type was Date, the argument's displayed name might be "starting date". If that argument's default was always the current date, the displayed name might be "today".

The Function's ScoreDeterminer is used to mark keywords that indicate the Function might be called. It works the same way a Type's ScoreDeterminer works: by taking in an input string and returning a collection of ScoreVectors. This time, each ScoreVector represents a possible call to the function.

Each Function is also tied to a Chickenscratch function. The Chickenscratch function is what gets executed in the Chickenfoot environment, with respect to the current visible tab, when the user chooses to run an interpretation. In order to avoid namespace collisions when the JavaScript is run, the JavaScript functions are anonymous.

A Function stores variables that are related to how it is displayed to users. Executable Functions (those that have a return type of VOID) might include the URL of an icon that goes with the function. If the function has a side effect, it also stores the name of the side effect. Finally, all Functions store a FunctionUserLangGenerator that, given a list of found arguments, outputs HTML for a user-readable version of the function. The FunctionUserLangGenerator is created from a template that specifies the format of the output string and where each argument should go in it. If the FunctionUserLangGenerator for a function that makes reservations got the arguments `G925` and `tomorrow`, it would output HTML that rendered as follows:

4.3.2 XML Specification for Functions

Functions are specified in XML files that start and end with the `webCommandMappingFunctions` tag. Each file contains zero or more `function` elements. Every `function` element has a `returnType` attribute, which is the name of the type it returns and a `keywords` attribute, which specify keywords for the function. It can also have a `sideEffect` attribute that says whether the function has a side effect (the default is false), a `sideEffectName` attribute that names the side effect, an `example` attribute with a sample command using the function, and/or an `icon` attribute with the URL of an icon for the function.

A `function` element's `keywords` attribute will contain a comma-separated list of possible keywords for the function. These keywords are used to create a ScoreDeterminer that will output ScoreVectors that mark subsets of the possible keywords. For example, a ScoreDeterminer made for a function with the attribute `keywords="read,view,mail,email"`, given the command `read mail mail` should return the ScoreVectors 1,1,0 (one function call consumes `read` and the first `mail`); 1,0,1 (one function call consumes `read` and the second `mail`); 1,0,0 (only `read` is consumed); 0,1,0 (only the first `mail` is consumed) and 0,0,1 (only the second `mail` is consumed). Note that the ScoreVector 0,1,1 isn't produced. Even though both

the second and third tokens are in the list of keywords, because they are the same keyword, they are not part of the same function call.

Creating a function's ScoreDeterminer in this manner makes sense since there are usually several keywords that match a function call and users are likely to use the same keywords twice if they want to call a function twice. Furthermore, since all functions that return VOID are executable functions that can only be called once in a single command, it makes sense for a single function call to consume all keywords for it. This also makes it easy to define synonyms for function keywords since they just get listed in the `body` attribute of the `members` element. For example, a function that looked up a name in the CSAIL directory might have the keywords `csail`, `dir`, `lookup`, and `find`. Given `find rcm csail`, the ScoreDeterminer would return 1,0,1, correctly consuming both keywords for the call. It would also return 1,0,0 and 0,0,1 in case it made sense for either `find` or `csail` to cue another function call instead. The command `csail dir lookup rcm` would also be recognized correctly.

A `function` element also contains a `userLang` element. This element's body contains a template for the text that will be shown to the user for this function. An `input` element appears in the text wherever feedback for that argument should appear. Each `input` element contains a `type` attribute that names the type of the argument. It also contains a `category` attribute that can be `required` if the argument is a required argument, `optional` if the argument is an optional argument without a default, or `default` if the argument is an optional argument that does have a default. It could also have a `defaultName` attribute, which is how the argument should be displayed if it is not filled in. If there is no `defaultName` attribute, the argument will be displayed as its type's name until it is filled in. Any arguments that are rarely used by a user should be grouped towards the end with parentheses around them.

Here is an example `userLang` element:

```
<userLang>
email
<input type="email address" category="required"/>
about
<input type="subject line" category="required"/>
(attach <input type="file name" category="optional" defaultName="file"/>)
```

```
</userLang>
```

Formatting the `userLang` element this way makes it easy to tell what the feedback will look like and where each argument goes. Looking at the element, one can tell that the feedback will have the word "email", followed by feedback for the email address argument, followed by "about", followed by the subject line argument, etc.

Finally, `function` elements have a `code` element that includes the relevant JavaScript or Chickenscratch code for the function in a CDATA block. This function's arguments should be in the same order the arguments were described in the `userLang` element. For functions that have persistent side effects, the last argument of the JavaScript function should be a run mode that can be either "view" or "submit". The JavaScript function should set up but not do the side effect when that last argument is "view" and set up and do the side effect if the last argument is "submit".

Here are some examples of `function` elements:

```
<function returnType="void" sideEffect="true"
          sideEffectName="Send Email"
          example="email joe@joeuser.com my subject line"
          icon="https://mail.google.com/mail/images/favicon.ico"
          keywords="send,attach,email,mail,e-mail">
<userLang>
email <input type="email address" category="required"/>
about <input type="subject line" category="required" />
(attach <input type="file name" category="optional" defaultName="file"/>)
</userLang>
<code>
<![CDATA[
function(email, subject, filename, runMode) {
     go('http://mail.google.com/mail/h/');
    // wait for login
    if (location.pathname.indexOf("mail") == -1) {
        alert("Please log in.");
    }
     while (location.pathname.indexOf("mail") == -1) {
         sleep(.2);
     }

     click("Compose Mail");
     if (email != null) {
```

```
          enter("To", email);
      }
      if (filename != null) {
          enter("Attachments:", filename);
      }
      if (subject != null) {
          enter("Subject:", subject)
      }

    if (runMode == "view") {
        return;
    }

    click('first send')
}
]]>
</code>
</function>


<function returnType="void" sideEffect="false"
          example="weather 02139"
          icon="http://www.accuweather.com/favicon.ico"
          keywords="weather,forecast,temperature">
<userLang>
get weather
(for
<input type="zipcode" category="default"
                      defaultName="last zipcode you looked up"/>
)
</userLang>
<code>
<![CDATA[
function(zip){
    if (zip != null) {
        var href = 'http://www.accuweather.com/'+
                    'us-city-list.asp?zipcode=\%s&u=1&partner=accuweather';
        href = href.replace(/\%s/,zip);
        go(href);
        return;
    } else { // use default if it is there
        go('http://www.accuweather.com/');
                click('first go');
    }
}
```

```
]]>
</code>
</function>
```

4.4 Keyword Engine

The keyword engine does the bulk of the work determining what interpretations can be made from a command. On setup, it takes in a set of defined Types and Functions. It can then take in a command, determine which Types the command contains keywords for, and finally determine a list of callable Functions (with corresponding arguments) for that command. This list is passed up to the front-end UI layer, which displays these possible interpretations to the user and executes them.

4.4.1 Parsing Specified Types and Functions

To define the Types and Functions in the Keyword Engine, one simply gives it the paths to XML files that define the types and functions. These XML files have been described above.

The keyword engine places all the types in a table that maps from the type's name to a Type object. Since there is only one namespace, if two different types with the same name are defined in the type XML files, the second type's definition will override the first type's definition.

The functions in the XML files will have access to any type that was defined in any of the type files.

4.4.2 Command Interpretation Algorithm

Inky uses a variant of the bottom-up algorithm described in "Keyword Programming in Java" to determine how a command should be interpreted [9].

The algorithm has three stages. In the first stage, it determines which basic types might exist in the command. In the second stage, it determines what new types can

be created using functions. In the final stage, it determines what executable functions can be called from all the types it has access to.

Inky uses a CommandMatrix to implement the algorithm (Figure 4-2). It is given a set of Types and Functions along with the command to interpret. Each row of the CommandMatrix will contain a set of Types and lists of interpretations that would evaluate to each Type. This is implemented as a map from Type to PossibilityHeap. A PossibilityHeap is a collection of Possibilitys ordered by their score. Each Possibility represents an interpretation of part of the command that, when evaluated, would result in the key Type.

A Possibility stores a ScoreVector indicating which tokens it uses, the Type it evaluates to, the JavaScript code that corresponds to it, and HTML that describes how it should appear to the end user.

The first row of the CommandMatrix contains all the basic Types that probably exist in the command and their corresponding Possibilitys. To populate this first row, the algorithm iterates through all the Types (except VOID) and, using the Type's ScoreDeterminer, gets all the ScoreVectors for that type with the given command. If there is at least one good ScoreVector, it creates a PossibilityHeap and puts that into the first row with the Type as the key. A good ScoreVector is one that explains at least one token. Then, for each good ScoreVector for that type, it creates a corresponding Possibility and puts that into the PossibilityHeap. As a result, the keys of the first row will be the basic Types that are in the command and the values will be PossibilityHeaps where each Possibility evaluates to the key Type.

The intermediate rows of the CommandMatrix contain all the basic Types that exist in the command along with Types that can be created by calling intermediate Functions. Intermediate Functions are Functions whose return type is not VOID. For each intermediate row of the CommandMatrix, the algorithm looks at the previous row to determine what Types it already has access to. Then, for each Function that does not return VOID, it determines if that Function can be called. To do that, it looks at each of the Function's Arguments, ignoring any optional arguments, and sees whether that Argument's Type is available from the previous row. If the Function

Defined Types: A, B, C

Defined Functions:
func1 : A -> B
func2 : B -> C
func3 : A, C -> VOID

Input: `someA1 someA2`

	Type A Column	Type B Column	Type C Column
Row 0	someA1 		
	someA2 		
Row 1	someA1 	func1 (someA1) 	
	someA2 	func1 (some A2) 	
Row 2	someA1 	func1 (someA1) 	func2 (func1 (someA1)) 
	someA2 	func1 (some A2) 	func2 (func 1 (someA2)) 

Final:

~~func3 (someA1, func2 (func1 (someA1))~~ 
func3 (someA1, func2 (func1 (someA2)) 
func3 (someA2, func2 (func1 (someA1)) 
~~func3 (someA2, func2 (func1 (someA2))~~ 

Figure 4-2: Inky's command matrix.

has multiple Arguments of the same Type, it checks that the previous row has at least that many ways to make that Type. If the Function has no Arguments, it is callable only if the command contained at least one of the Function's keywords. This prevents functions with no Arguments from always being callable and thus making certain Types available when they should not be.

If a Function can be called, Possibilitys for calling that Function are created. To do this, the CommandMatrix gets ScoreVectors from the Function's ScoreDeterminer that indicate where this Function's keywords appear in the command. Then it takes the top ScoreVectors and tries to find good Possibilitys for each of the Function's Arguments. If no good ScoreVector for the keywords is found, it uses a ScoreVector of all 0s. A good Possibility for an Argument is one that does not use a token already used by the Function or any other Argument. To find good Argument Possibilitys, the algorithm enumerates all the possible ways to obtain the Function's Arguments from Possibilitys in the previous row. Any enumeration that uses the same token twice, or that uses a token that is used as a Function keyword is eliminated. The algorithm determines which tokens the Possibilitys use by looking at their ScoreVectors. Finally, it makes a new Possibility for each of the leftover enumerations and Function ScoreVectors. The new Possibility's ScoreVector is obtained by summing up the ScoreVectors of the Possibilitys in the enumeration and then adding the ScoreVector for the Function's keywords. This process is shown in Figure 4-3. The Possibility's Type is simply the Function's return Type. The JavaScript code and HTML are obtained through the Function's code Generator and FunctionUserLangGenerator, respectively. These take in the list of Possibilitys for the Arguments, get the code/HTML from them, and place the code/HTML in the correct place with respect to the Function's code/HTML. Once the algorithm has the Possibilitys for a Function, it adds them to the PossibilityHeap in the current row for the Type that the Function returns. After it processes all the Functions, it adds all the Possibilitys from the previous row to the current row. This makes it possible for the next row to access those Possibilitys. The CommandMatrix currently creates two intermediate rows, but this number is easily adjustable.

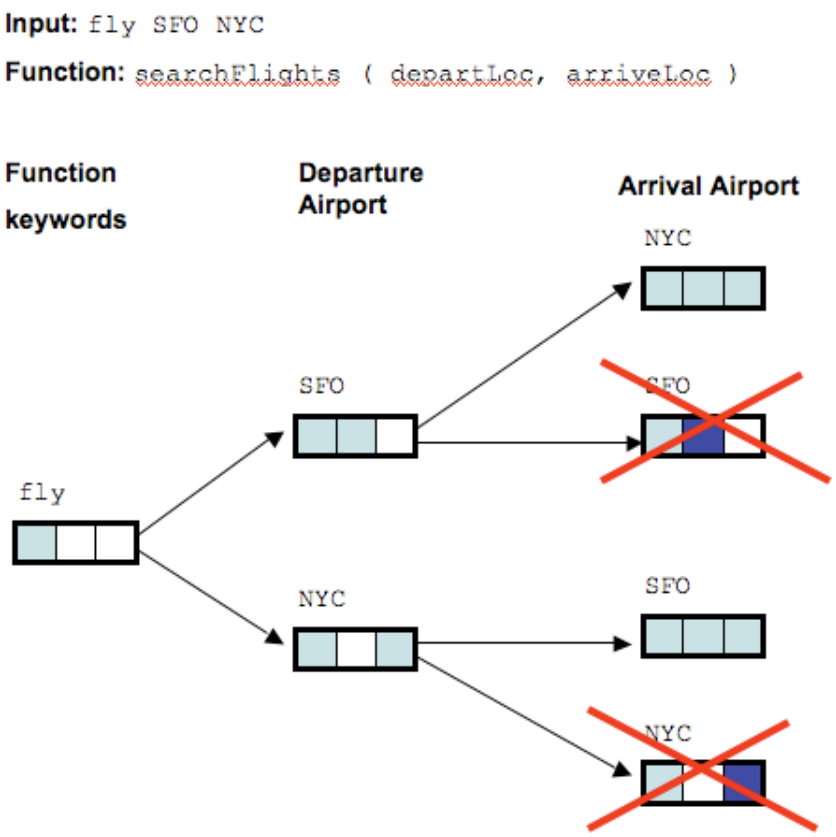

Figure 4-3: Inky enumerates the possible arguments and sums up the score vectors.

Once the algorithm is done with the intermediate rows, it determines what executable functions can be called. These are defined as functions that return VOID. The goal is to create RunPossibilitys for each way to call an executable function. A RunPossibility is like a Possibility except it contains code for different run modes and a side effect name. It also stores a pointer to the Function, which is used in post-processing. The method for obtaining and creating these RunPossibilitys is very similar to that for creating Possibilitys for Functions in the intermediate rows. The differences come from the fact that executable functions can be called in either a view mode (does not cause persistent side effects) or a submit mode (does cause side effects). A function can be called in view mode if the command contains any function keywords or if it has keywords for any of the arguments. A function can be called in submit mode only if the command contains all the required arguments. Once it is determined which (if any) modes a function can be called in, the algorithm creates ScoreVectors for that function in the manner described previously and uses those ScoreVectors to create RunPossibilitys. It then adds JavaScript code for each possible execution mode to the RunPossibility.

Finally, the CommandMatrix returns a list of the RunPossibilitys, ordered by score, for post-processing.

The algorithm differs from that discussed in [9] in the way it uses ScoreVectors (which they call Explanation Vectors). For example, it does not let two different functions calls or type instances try to explain the same token. When determining how well a function call would work with certain arguments, their algorithm would simply add all the ScoreVectors together. When this algorithm does this, it looks at each of the ScoreVectors and sees whether any two have a non-zero argument for the same token. If so, it rejects that possible function call with those arguments. This prevents it from getting interpretations that double-count tokens. For example, before implementing this change, `find flight SFO` returned an interpretation that searched for flights with SFO as both the departure and arrival airport.

The algorithm also creates separate ScoreVectors for each recognized instance of a type or function call instead of splitting the explanation value between all the tokens.

While this decreases the efficiency of the system by creating more ScoreVectors to process, it is only an issue when one type can be found multiple times or when one function is called multiple times. In return for the slight decrease in efficiency, the algorithm gains the ability to distinguish separate instances of a type or function call. This lets it enumerate possible arguments to a function more accurately, especially when the function takes in multiple arguments of the same type.

Code for a parsed result is generated as the command is parsed, along with the HTML feedback for the user. In contrast, this is done as a separate step at the end in Keyword Commands.

4.4.3 Post-processing

After a command has been processed, if the top resulting RunProbability has tokens that are unaccounted for, the algorithm does some post-processing to try to account for those tokens. Like Koala's algorithm [10], it tries to account for these tokens by extracting arguments from them.

It focuses its efforts on determining whether or not the tokens that are unaccounted for are supposed to be a string. Many functions take in a generic string. It makes sense for the generic string type to have quotation marks around it so that users can delineate where the string is and so that all commands wouldn't automatically match the string type. However, end users, especially novice ones, might not know that they should place quotation marks around such phrases. As a result, these phrases are not recognized as string arguments to the function. For example, the Google search function takes in a string. However, `google firefly tv show` would not be recognized as the function call `google ("firefly tv show")` since the last three tokens don't have quotation marks around them.

To deal with these occurrences, the algorithm looks at the top RunPossibility for the command. If the RunPossibility has tokens that are unaccounted for and the original command had no quotation marks in it, the algorithm sees if it can get a string argument from the unexplained tokens.

It goes through each RunPossibility and finds the longest unexplained substrings.

For example, if the original command was `send blake an email blake@email.com some subject`, the top RunPossibility's ScoreVector would be 1,0,0,1,1,0,0 since the email function has `send` and `email` as keywords and takes in an email address. The longest unexplained substrings are `blake an` and `some subject`. For each of these substrings found, the algorithm produces a version of the command where the substring is enclosed by quotation marks (`send "blake an" email blake@email.com some subject` and `send blake an email blake@email.com "some subject"`). This turns the substrings into something that would be recognized as a string object.

The algorithm then creates a CommandMatrix with all the defined Types. However, instead of having all the Functions, it is restricted to the executable Function tied to the current RunPossibility and the intermediate Functions. This means it does not have access to any other executable Function in the system. The new versions of the command are processed with this CommandMatrix and any resulting RunPossibility that has a score higher than or equal to the original top RunPossibility's score is added to the ordered list of RunPossibilitys returned to the user. In this case, both new commands would get a higher score than the original command since two more tokens would be explained as the subject line of the email to send. The user would then select the interpretation that had `some subject` as the subject line and not `blake an`.

It is necessary to remove all the executable Functions from the new CommandMatrix except for the current RunPossibility's executable function. Without this restriction, tokens that should have been arguments would disappear into a string and the list of suggestions provided to the user would be inundated with improbable interpretations. For example, consider the command `email rcm@mit.edu should we meet?`. This command would start out with two interpretations: sending an email to rcm@mit.edu, which gets a ScoreVector of 1,1,0,0,0 and reading one's email, which gets a ScoreVector of 1,0,0,0,0. The algorithm would then process `email rcm@mit.edu "should we meet"` through a CommandMatrix, which would produce the correct RunPossibility. However, it would also process `email "rcm@mit.edu something"` through the CommandMatrix. If the CommandMatrix is not limited,

this will produce interpretations that include sending an email with the subject "rcm@mit.edu should we meet" and no email address. Even though this interpretation is missing an email address, it still explains more tokens than merely checking email, so it would get a higher score. In fact, any interpretation that involved a function that just took in a string would be scored more highly than the viewing email interpretation since the string explains three tokens while the view email function would only explain one. Limiting the functions in the post-processing CommandMatrix solves this problem.

Chapter 5

Evaluation

Inky's interface was evaluated through a field study. The purpose of the field study was to determine how Inky might be used in day-to-day activities and whether it successfully met the design goals.

The performance of the Keyword Engine was also tested to determine if it was fast enough for the desired responsiveness and to see how it scaled for longer commands and more functions and types.

5.1 Field Study

A field study was conducted to see how users would utilize Inky. There were a total of 7 users, all members of CSAIL who regularly use Firefox. From the data gathered, Inky's learnability, the importance of synonym support, the importance of order-independence in commands, the importance of suggestions, and Inky's accuracy were evaluated.

For this field study, two different versions of Inky were released. The first, InkyA, provided all user feedback in the same standard order. This meant that, for both the example functions on startup and for the suggestions, the name of the function always appeared first and was followed by each of its arguments in the same order. The other version that was released, InkyB, provided mixed feedback. Sometimes the feedback had the function keywords at the front. Other times it was in the middle or

at the end. The order of the arguments in the feedback could also change between the startup example and the suggestions.

Each user went to a website that gave instructions on how to download either InkyA or InkyB and install it. The website also told each user the keyboard shortcut used to start Inky and the fact that the user should type into the Inky textbox.

After a week, users submitted the log Inky had been keeping of all their keystrokes in the Inky textbox. From this data, 131 commands that were typed into the textbox were submitted.

The results, which are discussed below in detail, were that Inky was fairly learnable. Its suggestions were important but the number of suggestions given should be reduced. Its support of synonyms is vital. Order independence is useful, but Inky could provide a more limited version of it without detracting from the user experience. Inky has reasonable accuracy.

5.1.1 Learnability

To test Inky's learnability, users received it with no instructions other than the keyboard shortcut to bring up the Inky screen and the fact that they should type something. The study aimed to determine if users could figure out what functions Inky had and how to use them solely from the startup screen and the user feedback.

Out of the 7 users, 4 noticed the startup suggestions right away and learned commands through them. 2 users never noticed the suggestions and 1 user noticed it after he tried a few invalid commands. All users successfully executed at least one command.

2 of the 7 users executed a command in submit mode.

3 of the 7 users realized they could execute a lower-ranked suggestion using the arrow keys.

From these findings it can be concluded that, while Inky is moderately learnable, new users could probably use a bit more instruction. It is unclear whether so few users utilized submit mode because they did not know what it was or because they honestly did not want to do a persistent action. A startup guide for Inky should

include some information about the user feedback panel.

5.1.2 Suggestions

This study also wanted to determine whether the suggestions were useful and how many suggestions Inky should display to the user.

4 of 7 users used the arrow keys to browse the suggestions. Of these, 3 looked at all five provided suggestions while 1 only looked at the first and second suggestions.

Out of the 39 commands executed by these four users, 35 used the first suggestion while 4 used the second suggestion.

The results imply that Inky should display fewer suggestions to the users. In a post-study interview, users mentioned that the suggestions were too similar and it was easier for them to change the command until the top suggestion was correct rather than scroll down for more suggestions. It would be incorrect to eliminate suggestions entirely as having at least two suggestions allowed users to execute their desired function even though their command was a better match for a different function. However, Inky should decrease the number of suggestions made and differentiate them more.

5.1.3 Synonyms

The study also demonstrated whether it was important to support synonyms in the functions and types. The startup screen and feedback provided used specific terms for specific functionality. If the users typed in commands that were valid but deviated from these terms, it is an indication that they wanted synonym support.

The study included 95 examples of user commands that were correctly interpreted by the system. Of these, 16 used a synonym for a function keyword. Of the 36 user commands that were not correctly interpreted by the system, 11 used a synonym that was not defined in the system. After adding those synonyms to the system, those commands were interpreted correctly.

These example commands give strong evidence that synonym support is crucial

to the system. With synonyms, the system could correctly interpret about 80% of the commands in the study. Without synonyms, this would drop to about 61%.

The user commands gathered also stress the importance of defining good synonyms for one's functions. For example, users expected the keyword `search` to be a synonym for the Google search function and `lookup` to be a synonym for the directory function. Since those were not defined in the function file, those commands were not correctly interpreted.

5.1.4 Order Independence

Since supporting order independence plays a large part in how the keyword engine was implemented, it was useful to see if it actually improved the user experience. Not supporting order independence would make the engine significantly faster. Since the feedback provided to the user contains the function keywords and arguments in a particular order, it is not completely unreasonable to expect the user to type her command in the same order.

The study collected a total of 89 commands that had some ordering information. Commands that were a single word long or were for functions that Inky does not support are not counted. 13 of these commands had an ordering that was inconsistent with the feedback Inky gave them.

From users of InkyA, which always had the function keywords at the start and the arguments in a consistent order in the feedback, 3 out of 60 commands did not match the order of the feedback. In all 3 of these cases, the user typed in the function keywords in a different order from the feedback. To illustrate, one of Inky's examples is `see csail reservations tomorrow`. In one case, the user typed `csail see tomorrow` instead. However, all the function keywords still appeared before all the argument keywords.

From users of InkyB, which had function keywords in different places and also reordered the arguments in different areas of feedback, 10 out of 29 commands did not match the order of the feedback. 7 of these commands had the function keywords at the start of the command whereas the feedback Inky gave them had them in the

middle or end of the command. 6 of the commands had the arguments in an order that was inconsistent with the feedback. (3 commands satisfied both qualities.)

The data suggests that users will naturally want to put function keywords before argument keywords in a command. However, it should be possible for different function keywords to be reordered amongst themselves. It is also important for Inky to support order independence among the arguments.

5.1.5 Accuracy

This field study also tried to judge the accuracy of Inky's keyword engine.

There were a total of 131 commands.

95 commands were interpreted correctly by Inky. If the correct interpretation of a command appears in the list of suggestions Inky provides, that command is considered to have been interpreted correctly. The users executed 55 of the correctly interpreted commands. However, in 5 of these cases, the user executed Inky's first suggestion instead of the correct interpretation.

11 commands were not interpreted correctly because Inky did not have the correct synonyms defined. These synonyms have since been added to Inky and the commands are interpreted correctly.

4 commands included misspelled function or argument keywords that prevented the interpreter from correctly guessing the function call.

21 of these commands were for functions that are not in the system. For example, Inky did not have any functions for buying food or storing notes so commands in this category include `find an apartment near` and `order food`.

The conclusion is that Inky's keyword engine is considerably accurate for its list of defined functions and within the list of suggestions it provides. However, since users often only look at the first interpretation, the keyword engine could be improved such that its first suggestion was more accurate. Changing the engine such that it could correct spelling errors would improve its accuracy. It might also be useful to change the engine to provide some default action for commands that ask for unsupported functionality. For most of the unsupported commands, Inky could have suggested

running them through a search engine to provide reasonable functionality. For example, searching for "order food" in a search engine will probably bring up a website for ordering food online.

5.2 Keyword Engine Performance

To test the performance of the keyword engine, tests were run using the commands acquired from the field study as well as randomly generated commands.

Each of the commands acquired in the field study were processed with the keyword engine. The average time it took to process a command was 15 milliseconds. The maximum processing time was 55 milliseconds and the minimum processing time was 4 milliseconds.

To determine how the keyword engine scaled, its performance was measured with commands of different lengths and with different numbers of types and functions. To generate commands of different lengths, a word bank was created using words from commands acquired in the field study.

Command lengths between 1 and 30 were tested. For each length, twenty commands of that length were generated by randomly choosing words from the word bank and stringing them together into a command. The processing times for those twenty commands were then averaged. Table 5.1 and Figure 5-1 show how the time to process a command changed with respect to the length of the command and the number of types and functions defined.

Inky aimed to provide instantaneous feedback to the user. According to [13], users do not notice delays of about 150 milliseconds for keyboard interactions. The times it took to process the commands from the field study were all well below this threshold. Furthermore, using all the functions and types currently defined in the system, processing commands of up to 30 words still did not produce a noticeable delay. Thus, Inky succeeds in providing instantaneous feedback to the user.

Adding types and functions increases the time it takes to process a command, but not by unreasonable amounts. The number of types were doubled and the number of

Command Length	gmail - 3 types, 2 functions	csail - 9 types, 14 functions	all - 16 types, 47 functions
1	16.31	4.28	24.08
2	6.65	12.82	47.67
3	7.10	11.71	46.28
4	5.20	10.60	38.89
5	5.37	12.56	42.06
6	4.11	13.56	38.49
7	5.60	12.59	42.53
8	5.19	15.49	39.69
9	6.21	14.30	44.08
10	4.92	10.97	39.47
11	6.33	14.86	41.72
12	6.11	18.01	40.97
13	5.63	14.35	44.45
14	6.60	15.12	47.04
15	6.41	18.48	46.11

Command Length	gmail - 3 types, 2 functions	csail - 9 types, 14 functions	all - 16 types, 47 functions
16	6.68	15.83	46.93
17	6.52	18.33	46.66
18	6.19	17.81	49.42
19	5.51	23.32	46.70
20	8.23	17.18	51.37
21	8.41	20.06	53.45
22	8.77	21.73	54.02
23	8.01	20.82	53.98
24	8.56	20.64	56.03
25	6.80	21.98	55.60
26	8.86	22.44	57.66
27	7.86	21.99	56.65
28	7.04	27.51	60.19
29	7.70	22.29	61.99
30	6.39	25.44	63.68

Table 5.1: Time in milliseconds it took to process commands of different lengths with different numbers of types and functions defined.

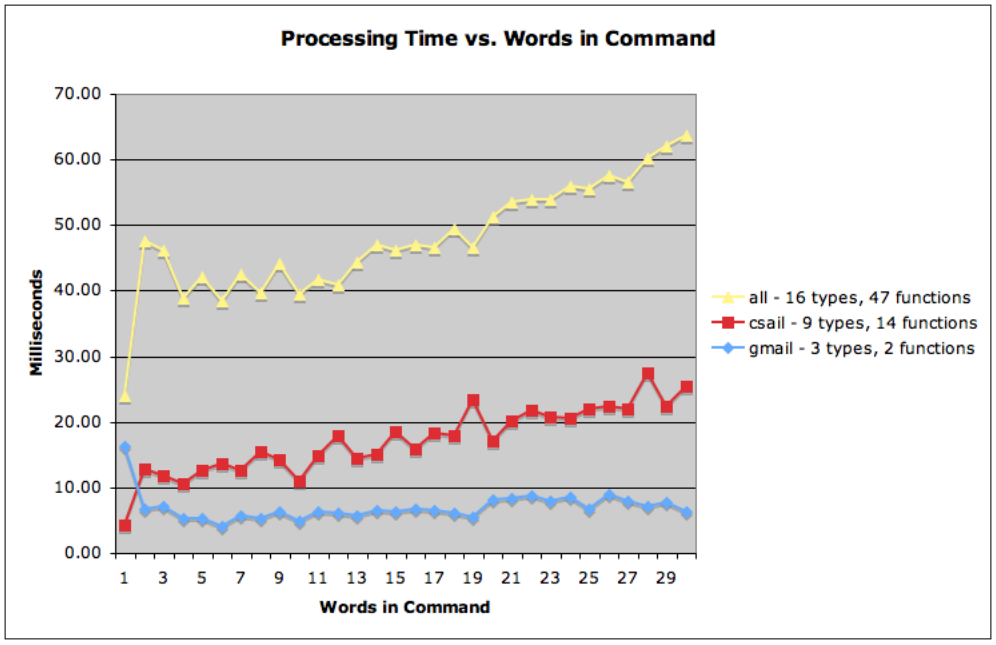

Figure 5-1: Processing time versus command length for different sets of types and functions.

functions were tripled between the csail set and the complete set. The time it took to process a command approximately tripled between the sets as well.

Chapter 6

Conclusion

Inky allows users to access website functionality in a command line-like interface while mitigating the traditional pitfalls of such interfaces. As a Firefox plugin, it is easily accessible to the user and has an area for users to type commands. It displays dynamic feedback to the user about the state of the interpreter and gives the user hints on how to complete her command.

The sloppiness of Inky commands and the feedback it provides on the state of the interpreter are Inky's distinguishing features. Since Inky's command are sloppy, it accepts synonyms of function and type names and allows a user to type them in any order. Inky's feedback indicates what command the user is trying to call, the arguments a user has already filled in, and the arguments a user still needs to fill in. It provides several suggestions when a command is ambiguous. Inky also defines a separation between commands with persistent side effects and those without.

By storing its supported functions and types in human-readable XML files, Inky also makes it easy for users to add to the system.

From informal user tests, there is strong evidence that Inky is a learnable interface that can accurately interpret a user's command. There is also evidence that sloppiness is important to the interface.

6.1 Future Work

From the user studies and interviews with users, it is clear that there are many ways that Inky can be extended to improve the system. Improvements can be made to the front-end to make it more memorable. There is also significant room for improvement in function and type definition. Finally, it would be good to do some more user tests and increase the accuracy of the keyword engine.

6.1.1 Improved Reminder Feature

Users in the field study often forgot that Inky existed. As a result, they would fall back on their old, less efficient ways of accessing web functionality. Inky should remind users that it exists in an unobtrusive manner.

To do this, one might add in a reminder feature like that in Smart Bookmarks [14] that would display a drop down message when a user visits a page that corresponds to a command. The message will include a sample command that they would have been able to run to get to the same page, in the manner of other self-disclosing tools [11]. In order to make the reminder less annoying to the user, Inky should keep track of how many times the user has executed each type of command. Then, when the user goes to a page corresponding to that command, Inky would only pop up the reminder probabilistically. This probability would be inversely proportional to how often the user has called that command. This will prevent the reminders from coming up when a user who knows about the Inky command just wants to manually access a webpage.

6.1.2 Front-end for XML Editing

While XML is designed to be a human-readable format, Inky should include a better interface for editing its function and type files. This would make it easier for end users to customize Inky and for website owners to create downloadable Inky files for their users.

To do this, a sidebar that Inky can bring up for editing these files could be created.

Since Inky functions do not have names, users would specify a function to edit by typing a command. This would bring up feedback similar to that in Inky, but with fewer visual cues for the arguments. Users can then click on the function they want to edit to enter a function-editing window. For feedback that includes multiple function calls, users would click on the function keywords of the function they wanted to edit.

The function-editing window will have fields that let users edit the function's keywords, arguments, and JavaScript body. An advanced editing button will open fields that edit the function's icon, example, side effect name, how to translate it to a user-readable format, and its arguments. The argument-editing area of the window will include a button that will pop up a type-editing window for that type. To edit a type, users can also specify a type name and the file it comes from. Inky should also provide a testing area so that users can see how Inky would interpret commands with their changes before they save the changes.

It would also be good for users who are not familiar with JavaScript or Chicken-scratch to be able to create their own functions more easily. One idea is to incorporate work done in Smart Bookmarks [14]. Users could then essentially save parameterized bookmarks, create arguments for each parameter, add in some synonyms for calling that bookmark, and call it through Inky. This would prevent users from having to stop and fill in the parameters each time the bookmark ran.

The interface for editing functions could be somehow integrated with Inky so that users could quickly add something when Inky gave incorrect interpretations.

6.1.3 Hierarchy of Functions

To make function files easier to create, Inky could impose a function hierarchy on the system. This way, functions can be grouped by domain name and subdomain name. This kind of hierarchy will allow Inky to automatically detect things like the favicon URL and basic keywords that should go with the function. With a hierarchy, users won't have to specify that every CSAIL function has "csail" as a keyword or that every Google Calendar function has both "google" and "calendar" as keywords. This function hierarchy could also make the interpreter's algorithm more efficient.

For example, when a keyword specific to part of the hierarchy appears, it could limit or at least heavily bias its search towards functions in that subtree. This hierarchy would only be imposed on callable functions.

6.1.4 Directed User Study

While the field study evaluated Inky on its own merits, it would be useful to conduct a directed user study in order to compare Inky to traditional ways of accessing web functionality. In such a study, users should be asked to access functionality both through traditional methods and through Inky. The experimenter should then compare how much time each method took and the overall user experience for each method. User tasks should include at least one task that corresponds to a single argument function, one task that corresponds to a multiple argument function, and one task that corresponds to a custom Chickenscratch function.

6.1.5 Algorithm Improvements

The algorithm could be improved such that it was faster and so that the correct interpretation appeared in the top two or three suggestions. There are several ways to do this.

One improvement would be to have the algorithm weight functions based on how likely they are to be called. This weighting would be a combination of some default likelihood of being called and data gathered from the user. The default likelihood would be specified in the XML files and would be the XML author's best guess as to how often their function should be called. For example, one would expect someone to look at room reservations more often than they would actually reserve rooms. One would also expect people to use a generic search engine more often than they would use an image or synonym search engine. To make the weightings more personalized, Inky will also gather data on how often a user calls each function. Functions that were used more often would be given more weight, as it is likely that the user wanted to call that same function again.

Another improvement would be to limit or bias the algorithm's search. As the user studies showed, there is a high probability that any function keywords would appear at the start of the command. If the function hierarchy were implemented, that would further limit the search space. This would allow the engine to run faster and perhaps display more relevant suggestions with less information. The command matrix could also store the results of past commands that had been processed and use that information if a previous command were a substring of a new command. That way, the whole command wouldn't have to be reprocessed every time the user added a new character, allowing the user to get more instantaneous feedback.

Bibliography

[1] M. Bolin. End-user programming for the web. Meng thesis, Massachusetts Institute of Technology, 2005.

[2] R.C. Miller. Lecture 6: Models & metaphors. http://groups.csail.mit.edu/graphics/classes/6.831/lectures/L6.pdf, September 2004.

[3] D.M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6), July-August 1978.

[4] YubNub - a (social) command line for the web. http://www.yubnub.org.

[5] Mozilla firefox - smart keywords. http://www.mozilla.org/products/firefox/smart-keywords.html.

[6] A. Pash. Hack attack: Firefox and the art of keyword bookmarking. http://lifehacker.com/software/bookmarks/hack–attack–firefox–and–the–art–of–keyword–bookmarking–196779.php.

[7] Google help : Search features. http://www.google.com/help/features.html.

[8] G. Little and R.C. Miller. Translating keyword commands into executable code. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 135–144, Montreux, Switzerland, 2006. UIST.

[9] G. Little and R.C. Miller. Keyword programming in java. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 84–93, Atlanta, Georgia, 2007. ASE.

[10] G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, and E. Kandogan. Koala: Capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, San Jose, CA, April 2007. CHI.

[11] J.O. Wobbrock. In your own words: Using full sentences as feedback. *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI '02)*, pages 886–867, 20-25 April 2002.

[12] quicksilver:what_is_quicksilver. http://docs.blacktree.com/quicksilver/what_is_quicksilver.

[13] J.R. Darowski and E.V. Munson. Is 100 milliseconds too fast? *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems (CHI '01)*, pages 317 – 318, 2001.

[14] D. Hupp. Smart bookmarks: Automatic retroactive macro recording on the web. Meng thesis, Massachusetts Institute of Technology, 2007.

[15] Java 2 platform standard edition 5.0 api specification. http://java.sun.com/j2se/1.5.0/docs/api/.

[16] Chickenfoot. http://groups.csail.mit.edu/uid/chickenfoot.

[17] Javascript – mozilla developer center. http://developer.mozilla.org/en/docs/JavaScript.