2003

IIC: Information in Context

Jennifer Mankoff
Oberlin College

IIC: Information in Context

Jen Mankoff (jmankoff@cs.oberlin.edu)
Oberlin College, Computer Science

January 29, 2003

Abstract

IIC (Information in Context) is an interactive programming environment meant to be used for profiling and tracing functions, as well as for visualizing information about functions. IIC runs in the Scheme interpreter STk and has both a graphical and a textual user interface. By integrating a profiling environment with a visualization system, IIC gives the programmer the power to dissect the internals of her program. In addition, IIC gives the programmer the power to control what data is gathered through the use of *scope* IIC makes it possible to evaluate the efficiency of functions and to easily identify bottlenecks. IIC can also be useful for debugging.

Contents

1 Introduction **2**
- 1.1 Goals and Motivations . . . 2
- 1.2 Information in Context . . . 2

2 Background and Related Work **2**
- 2.1 A sampling of other information gathering systems for programming and debugging . . . 2
- 2.2 Cooperative Software . . . 3
- 2.3 Tools used . . . 4

3 Information In Context **4**
- 3.1 Context when gathering data . . . 4
- 3.2 Context when viewing data . . . 6

4 Implementation **6**
- 4.1 Collecting Information . . . 6
- 4.2 Visualizing Information . . . 6
- 4.3 The Interface . . . 7

5 Testing IIC **8**
- 5.1 User evaluation of the interface . . . 8
- 5.2 User evaluation of the utility of the program as a whole . . . 12

6 Applications of IIC **12**
- 6.1 Using IIC to estimate the algorithmic complexity of functions . . . 12
- 6.2 Using IIC to generate traces of functions . . . 12

7 Limitations of IIC **13**
- 7.1 The Heisenberg bug . . . 13
- 7.2 Limitations of STk . . . 13

8 Further Investigations and Questions **13**

9 Conclusions **14**

A The user evaluations of IIC **16**

1 Introduction

1.1 Goals and Motivations

While simple debuggers ([Ber93], [HO85], [NF93], [Dyb87], [Cho89], [Duc93], [GH92], [Kam90], [nML92], [Nai92], [NF92], [OCH91], [Sha83], [Sny90], [Tol92], [WN88]), execution profilers ([San94], [Kis92]) and trace facilities like those available in Chez Scheme ([Dyb87], [FF89]) allow you to view events in a functions execution and to traverse the execution stack are widely available, they only allow the programmer access to a small subset of the information which could be useful to her. For example, data about how often functions are called in different situations, or how much CPU time is spent on a specific task can help the programmer to identify bottlenecks or to measure the efficiency of her program. In order for the programmer to gather data in a specific situation, IIC allows the programmer to specify the scope in which each function should be profiled. In order to make that data immediately useful to the programmer, IIC integrates data gathering with a graphical and textual data visualization system.

My broad goal in this project is to provide programmers with a complete interactive programming environment for writing interpreted programs. What I have done begins to fill in some of the gaps that are generally present in functional programming environments by making information about parts of a program which are not readily accessible available to the programmer. In addition, it integrates information gathering with information visualization and a graphical user interface meant to increase ease of use ([TS84], [DFAB93]).

1.2 Information in Context

Context is defined as "The circumstances in which a particular event occurs" (from the `Webster` program, written by David A. Curry). In the case of IIC, an event is a function call. The user can specify the context in which to watch for certain function calls by using scope. Such events are recorded as data items. While raw data is not very intelligible, it can become useful depending on how it is displayed. This is another example of context – the context in which the data is viewed.

IIC lets the user determine when information should be recorded based on the scope from which a function is called. This scope can be as broad or as specific as the user chooses. If data were recorded every time every function were called, the volume of information recorded would be huge. In fact, if the user were to run a really large program, the volume of information might be too large to cope with. By allowing the user to specify a context for information gathering, most of the unnecessary data which would normally be recorded can be weeded out.

Once the raw data has been gathered, IIC provides the programmer with helpful ways to view it in order to make it understandable. Because the data is gathered based on a context (scope), data recorded in similar contexts can be grouped together to improve understanding. This means that the context in which information is gathered also helps in the understanding of that information. A graphical plot of such a group of data gives the information a context from which information can be extracted. A list of data sorted by some criterion might also help the user to extract some meaning from the raw data.

2 Background and Related Work

In this section I discuss some other types of information gathering systems and a complete cooperative program environment which ideally should integrate them. In addition, I give some background information about the tools which were used in the implementation of IIC.

2.1 A sampling of other information gathering systems for programming and debugging

There are many different types of information available about programs both in their textual form and as they execute. This section introduces some of those types of information, and systems which have been used to gather and analyze it.

Space/time/call-count information: Data can be recorded every time any function is called, and then analyzed to provide call count data. In addition, at runtime, a specific set of functions can be watched and their space usage/ CPU usage can be recorded. This is generally referred to as "execution profiling" ([San94], [Kis92])

Execution trace: At runtime, a trace of functions called and arguments can be printed out. The `trace` function in Chez Scheme ([Dyb87]) provides an example of this

Dataflow information: Dataflow information ([MJ81], [Bar93] [AC76], [Ryd83], [KSF92], [OCH91], [Wei84], [AH90], [Bal93], [ASU86], [Dat82]) can be used to analyze possible execution paths which may be followed in a program:

- Dynamic dataflow analyses

  At runtime, information can be gathered about every function that is called, and which function it was called from ([Cho89]). When an error occurs, or an inefficiency is discovered, this information can be used to track down its cause.

- Static dataflow analyses

  A program can be parsed into a graph showing every function that may be called, and which function it may be called from. This is useful, for example, in determining which functions will be affected if a given function is changed (as part of a bug fix, for example). ([Bar77])

In addition, dataflow analyses can be used to understand the hierarchy of functions or classes in a program.

Stack information: At runtime, access to the stack of an interpreter can help the programmer with debugging problems. Breakpoints and errors can provide an entry into the debugger, which then allows the user to traverse the stack.

Environmental information: Access to the environment during runtime can also help the user with debugging tasks.

Type information: In dynamically typed languages such as Scheme, it can be helpful to know what type has been assigned to a variable. A type inference system can also help the programmer to determine what type the interpreter will assign to a variable. A comparison of the relative usefulness of type versus dataflow information can be found in [Bar93].

Text-based information: Program text can be scanned for cliches (i.e., the word sort, or standard algorithms, or loops) and this can be used to facilitate editing ([Wat94]) or provide suggestions (i.e., recognizing or giving warnings about potential endless loops). Two versions of a program can be compared or combined by identifying the semantic and textual differences between them ([Hor90]).

2.2 Cooperative Software

The ideal cooperative programming environment (something I have not had the time or the resources to implement) would help the programmer to do much more than just analyze her programs. In [Ret93], cooperative software is defined to have the following properties:

Cooperative Software [Ret93]:

Domain Orientation: A tool with this property can easily incorporate and represent aspects of the specific domain it is used in. This simplifies the problem of communicating with the user. Clearly, my tool has this property.

Critics: "Critics are feedback-generated rules that examine the work in progress and the domain knowledge to offer non-intrusive constructive critiques."[1] Another way to view this would be a context-oriented help system mixed with an expert system that can suggest meanings of error messages, or code which might cause endless loops. This alone would be a large programming project.

[1] ([Ret93], p. 26)

Integrated Problem Setting and Problem Solving: "Allow people to propose a partial solution (by choosing from a catalogue of parts or starting from scratch), consult the critics, then plan their next move."[2] (eg [ELJ94]) In the context of programming environments, such a catalogue might for example include the Scheme slib[ELJ94] library.

Integrated Action and Reflection: This ties in with the above item. When the programming process stalls, the programmer can consult a critic (in addition to reflecting or consulting the tutor or a friend).

Integrated Construction and Argumentation: The critic can be supported by argumentation tools, by providing "access to information about the nature of the problem, and the rationale behind it. The tool's domain knowledge should not only include facts and issues, but information about their dependencies and relationships."[3] In addition, I feel that the argumentation tool should be able to learn more about the program as it is being built, in the ways described under above in Section 2.1 (A Sampling of other information gathering systems for programming and debugging).

2.3 Tools used

IIC is implemented using the STk ([Gal]) interpreter. STk has the following properties:

Scheme: STk is, at heart, a Scheme interpreter. STk is R4RS[CJ91] compatible.

Tk: The most important of the extensions to Scheme in STk (and the basis for the name STk – *Scheme-Tk*) is the Tk[Ous94] widget set. The TK widget set combined with the CLOS-like object system (see below) make it possible to quickly prototype user interfaces.

A CLOS-like Object System: STk also comes with a CLOS-like object system. One of the most useful aspects of this system is the fact that you can encapsulate Tk widgets as objects. This makes them very easy to manipulate.

Access to interpreter: STk is freely distributed as source code. This makes it possible to directly manipulate the interpreter. While I did consider making use of this feature, IIC is implemented without any interpreter modifications.

3 Information In Context

In this section I go into more specifics of how IIC allows the programmer to make use of the concept of context introduced in Section 1.2 (Information in Context).

3.1 Context when gathering data

IIC allows users to filter some portion of the voluminous information available while a program is being run. IIC is based on the assumption that users want information about specific functions. It takes that idea further by allowing users to gather information about specific functions in specific contexts. The context which IIC uses to filter data is scope. At the functional level, scope is an ideal filter. For example, a programmer might wish to know about every occurrence of the function '+' within the function 'factorial'. The function '+' might be called often in other contexts, but because the programmer can specify the scope in which data about '+' is recorded or displayed, the programmer won't have to search through all of the useless data generated by calls to '+' that don't occur within 'factorial'.

In order to specify scope, the programmer sets up a directed graph of functions which are being watched. The nodes of the graph are functions. An edge in the graph points from a function, to another function which is being watched for by the first function. In other words, if data is being recorded about B whenever it is called from within the scope of F, there will be a directed edge pointing from node F to node B. This graph is stored in a data-object, along with the data and other information. The graph representing functions and

[2] (ibid, p. 26)
[3] ([Ret93], p. 26)

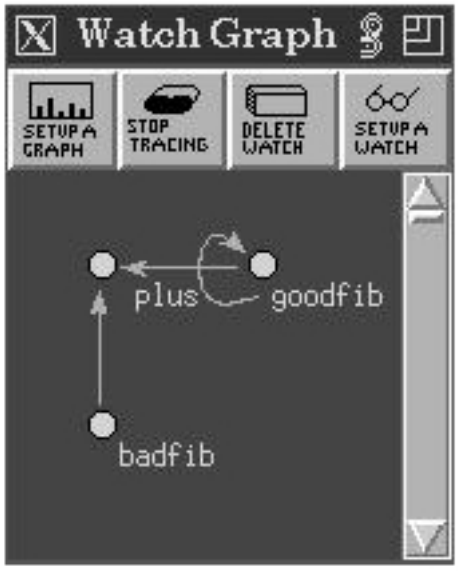

Figure 1: **A Sample Watch Graph**: In this graph, all occurrences of plus within `goodfib` and `badfib` are recorded. In addition, anytime `badfib` is called, the event is recorded. `goodfib` and `badfib` are two versions of a function which, given input n, generate the nth Fibonacci number. `badfib` is "bad" because it takes a long time to generate this number. It will however eventually return the correct Fibonacci number.

the scope in which data should be recorded about them is represented visually in IIC. Nodes are labeled with the appropriate function name, and edges are drawn between nodes (representing scope information). (See Figure 1)

For each initial call to B with different input, IIC records a new set of data. If B is called twice with the same initial input, the new data replace the old. If B is called recursively from within itself, data is *only* recorded for the initial input/call to B.

There are two categories of information available when a function executes – transient information, and permanent information. One could choose to record every piece of information permanently or never to record certain pieces of information. When deciding how to categorize this information, it is advantageous to consider how such information might be used. For example, when a function is called, its current depth is useful in a trace of the execution, and the fact that it was called is useful in a call count. A trace is often done as the function runs. Once it has finished running, and the trace has been printed out, the depth does not need to be kept around. On the other hand, the call count is not very useful *until* the function has finished running – until then you don't have a total. IIC categorizes the available information as follows:

- Permanent information gathered
  - Initial input to B
  - # calls to F from when B is called with that initial input until B returns
  - CPU time spent in F from when B is called with that initial input until B returns
  - `cons` cells allocated for F from when B is called with that initial input until B returns
- Transient information gathered
  - Current input to B
  - Current input to F
  - Current depth of B
  - Current depth of F

3.2 Context when viewing data

Even once the volume of data has been lowered as described above, there is usually still too much information recorded to easily understand. By visualizing parts of that information using graphs and charts, the programmer can extract meaning from it. The principles used by IIC for data visualization were derived from the work of William S. Cleveland ([Cle43], [JMCT83], [Cle93]).

4 Implementation

IIC was implemented using the STk ([Gal]) interpreter. IIC was implemented solely at the scheme level: no modifications were made to the interpreter itself. The reasons for this choice were

- Portability: IIC can be more easily distributed if users don't have to compile a special version of STk to use it.
- Ease of implementation: Modifying the interpreter directly was difficult because IIC needed to be integrated with the interpreter for ease of use. Since parts of IIC had to be implemented as STk functions, when one of those functions was noticed by IIC, it would set off a chain of events which would call that function which would set off a chain of events which would call that function which would ... and so on. Basically, it was very hard to avoid the problem of infinite recursion.

No great pains were taken to make IIC efficient. By ignoring this issue, I was able to implement a much more complete program, but as a result it is not as useful for large or for recursive functions and measurements of CPU time and `cons` cell usage are generally inaccurate.

4.1 Collecting Information

IIC uses one main object, `basic watcher` to keep track of all the functions which are being watched for. `basic watcher` stores a literal representation of the virtual graph discussed in Section 3.1 (Context when gathering data). A function is categorized as a `watched function` if there is a directed edge arriving at the node representing that function. A function is categorized as a `watcher function` if there is a directed edge leaving the node representing that function. IIC uses STk's object system to represent such functions as objects. An object is instantiated for each `watcher` and each `watched` function. If a function is both a watcher and watched then two objects are instantiated. A watcher object contains a list of all the watched objects which it is watcher for (i.e., all objects for which data should be recorded if they are called within its scope). Data for each watched function is stored in that function's watched object.

IIC uses a stack to keep track of when functions are called and when they return. When a watcher function is called, it adds itself to the stack. When a watched function is called, it checks to see if any of the functions in the scope of which it should record data are on the stack. If they are, it has been called from within the scope of that function, and it records any important permanent information and prints out any important transient information. Functions on the stack store their depth and input values so that appropriate trace information can be outputted.

When a function is watched, it is redefined as a `dynamic-wind` which pushes the function on the stack and does any other setup necessary, calls the original function, and then pops itself off the stack and does any cleanup necessary. A dynamic wind is a scheme construct which takes three lambda closures. They are executed in order. The value of the middle lambda closure is returned. All three are guaranteed to be executed.

4.2 Visualizing Information

There are two types of information in need of visualization: permanent and transient information. Because of the differences in the nature of these types of information, it is important to provide different routines for viewing them.

- Visualization of transient information (the trace)

  IIC provides a textual report system which is equivalent to the `trace` function in Chez Scheme ([Dyb87]). This uses information from the stack described above to print out function names with their arguments, indented based on their overall depth whenever an instrumented function is called or returns.

- Visualization of permanent information: One important thing to realize about the raw data gathered by IIC is the fact that the data consists of a set of discrete data points. IIC has no way of understanding the relation between two datapoints. Remember that when data is gathered for all occurrences to B within the scope of F, something is recorded for each initial input to F. The unique tag for a data value, then, is the initial input to F combined with the knowledge that the data value represents the number of calls, total CPU time, and total `cons` cells used by B for that input to F.

  In order for IIC to relate two data points, it must know what the relation between two inputs to F is. This may change dramatically depending upon what F does. For example, if F represents a mathematical function, it's inputs might be integers, and they might relate in a monotonically increasing way. On the other hand, if F is a function which will sort the elements of a list, F's inputs might be lists, and the length of each input to F might relate in a monotonically increasing way. Other functions might be even more complicated. Although the programmer *does* (usually) know what the relationship between these inputs is, IIC does not know.

  Given those constraints, IIC provides the following methods for visualizing permanent information:

  - Textual report of the information

    The facility for viewing data textually is currently very primitive. Data is sorted from largest to smallest by value and printed out in a long, minimally formatted list.

  - Plotting the information

    Since each of the datapoints is discrete, IIC is only left with one real option as to how to represent them visually: the canonical plot to use for discrete datapoints is the dotplot. What IIC can do is allow the user to provide a comparison function which will allow IIC to sort the datapoints appropriately. Thus, the dotplot can represent datapoints in an order which will help in making sense of the result.

    The user can make such a plot in two ways: By selecting a pair of functions from the "Watch Graph", the viewer can view a plot of all of the data gathered when one of those functions was called from within the scope of the other one of those functions. The viewer can also display a dotplot of data chosen based on a viewer-provided function which selects input values. This allows the viewer, for example, to compare the performance of five different implementations of `sort` on lists of length 100.

4.3 The Interface

The interface for IIC ([TS84], [DFAB93]) was not designed using a specific set of user interface guidelines. It does, however, attempt to fulfill the nine heuristics listed in Table 1 (from [NM90]). In general, all of the important functions of IIC's interface are implemented both as part of the graphical user interface and as part of the textual user interface.

IIC's interface consists of two windows: The "Watch Graph" and the "Watcher" windows. In the "Watch Graph" window are buttons and menus for setting up and removing watches, for changing characteristics of watched functions (i.e. whether they should print out a trace of transient information), and for making graphs of data. In addition, the "Watch Graph" window has a graphical display of the directed graph of watched functions as described in Section 4.1 (Collecting Information). The function of the "Watcher" window is to display dotplots of the data which has been gathered. It provides buttons for deleting plots, for quitting IIC, and for getting help with IIC.

Watches can be set up both via STk commands typed in to the interpreter and by using the menus and buttons described above. Dotplots can also be created using STk commands or by using graphical dialog

Table 1: A list of nine heuristics for user interfaces

1) Simple and natural dialogue	6) Provide clearly marked exits
2) Speak the user's language	7) Provide shortcuts
3) Minimize user memory load	8) Good error messages
4) Be consistent	9) Prevent errors
5) Provide feedback	

boxes. This makes it possible for the user to create a batch script which will execute a series of watches and bring graphs of them up, or to manipulate IIC by hand.

A datapoint in one of IIC's dotplots represents some data value for a function B which was watched for in the scope of some other function F. Data points are labeled by the initial input to F. When such labels are too long (eg, a list of 100 elements), they are represented as a single dash. When there are so many datapoints that the labels cannot all fit, they simply overlap. Users can select and view a specific label by clicking on the data point represented by that label with the mouse. That label will then be brought to the foreground (above other labels which might overlap it) and highlighted. If that label is represented by a dash, it will be replaced by the actual input to F. An example of overlapping labels can be found in the plus/goodfib-helper graph in Figure 2.

One might think that the logical solution for dealing with the problem of too many labels would be to only show every 5th or 10th or 100th label. However, because IIC cannot know the relationship between two datapoints in the plot, it has no way of knowing when it is appropriate to do this and when it is not. So instead, IIC displays every label, and allows the user to select multiple labels as described above which should be highlighted.

Note that the ticks along the bottom of the plus/goodfib-helper graph do not necessarily represent the exact number on which a data value will fall. The ticks are generated based on the suggestions of William Cleveland in [Cle43]. In general IIC aims for between 5 and 10 tick marks. It attempts to find integer values for these tick marks, but the algorithm has not been perfected, and occasionally it misses as in the plus/goodfib-helper graph in Figure 2.

The help facilities in IIC are relatively simple. Each dialog box has a `help` button which will provide a brief description of the functionality of that dialog box. In addition, the main window (`Watcher`) has a help button which, when selected, will bring up a window with a textual description of the whole interface. This window contains information about watching and unwatching functions, the watch graph, tracing, and graphing data. Users can navigate through the text using a scrollbar.

5 Testing IIC

Ideally, IIC should be usable, applicable and relatively easy to learn. In order to test these properties of IIC, people not involved in the development process of the program were asked to help with its evaluation.

5.1 User evaluation of the interface

The standard approach to user evaluation of an interface involves developing a questionnaire about the interface. In [NM90], it is shown that five to ten subjects without training in Human Computer Interaction, working independently, discovered more than two thirds of the known (to the researchers) problems with a user interface. In addition, those subjects found problems that were not previously known to the researchers.

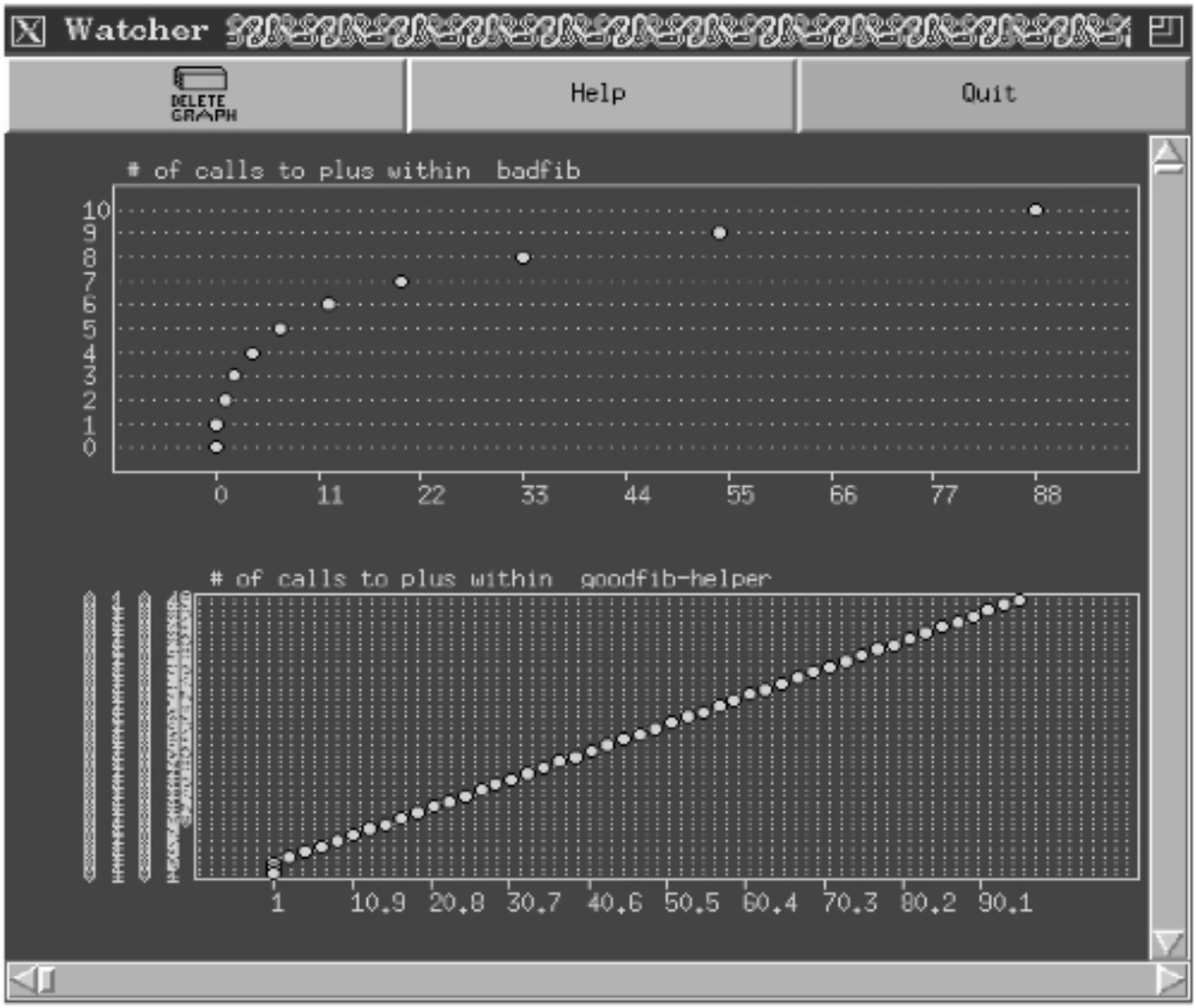

Figure 2: **Plots of data about calls to `plus` within `badfib` and `goodfib`**: This picture is an example of two dotplots, one depicting the performance of the function `badfib` on a variety of inputs, and the other depicting the performance of `goodfib` on a variety of inputs. To measure the performance, we count the number of occurrences to `plus` within `badfib` and within `goodfib`. `badfib` and `goodfib` are two versions of a function which, when given a number n, generates the n'th Fibonacci number. `badfib` is "bad" because it takes a long time to generate this number. It will however eventually return the correct Fibonacci number.

Seven volunteers[4] evaluated IIC, and between them they found most of the known problems with the interface, and more than one new problem. The volunteers were given a list of nine heuristics (See Table 1) to consider when judging the interface.

The volunteers seemed to feel that IIC successfully followed the nine heuristics. Some volunteers felt that IIC did not effectively minimize the user memory load: The sequence of steps involved in analyzing a function using IIC can be long. This seems, however, to be more of a learning curve problem than anything else. Another problem that more than one person spoke of involved error messages: STk handles errors by printing out the top four things on the stack. Since that usually includes code used specifically for instrumentation, it makes it hard for users to debug instrumented functions. In addition, there were some problems with consistency: some very important functionality is only available via the textual interface.

The full text of the volunteer responses as well as a more detailed summary of their ideas can be found in Appendix A. Below is a summary of the main problems and goodies which the volunteers identified, taken from the volunteers' evaluations. Items marked with stars below are things which I had not thought about or thought of before the volunteers brought them to my attention. The number preceding each item is the approximate number of people who remarked on it (since each volunteer worded things differently, sometimes it's hard to determine if they are saying exactly the same thing or not).

General Stuff: These are points about the interface in general.

2* The grey background in the entries makes the cursor hard to see.

1* The ability to use arrow keys to move inside the entries would be nice.

1 It would be nice to be able to trace any lambda closure (e.g., named lets)

1 It would be helpful to have a sort of typical pipeline for analyzing functions.

1 With a small amount of time, I became used to IIC, and was able to be quite productive with it. It only takes a little bit to play with all the different features before you feel comfortable enough with it to use it quite quickly.

1 In general, the goal of not taxing the user's memory load isn't met. There's a lot to do, and nothing is around to remind me of each of the steps (except Help, which in my opinion shouldn't have to be consulted terribly often).

1 Processes are simple enough that users do not have to memorize very much.

1* Before opening new instances of X windows, check if they're already there.

1 Wow, those STk error messages suck.

1* Can't play with textual stats with a button.

1 It might be nice if you added something that automatically ran something many times with various inputs the user chooses so you can get lots of data points.

1* Return value of watch is `#f`. Why?

1 It ends up being a little confusing having both the graphical interface and the stk interface.

1 It would be nice to be able to do everything in the graphics windows, somehow. Like loading files, etc.

1 The dialog boxes are slick. The controls are easy to use. The user's choices make sense. The user is presented with several choices but not so many that the dialog becomes muddled. Like any well-designed interface, the inexperienced user can learn a lot about the program's use just from a few minutes of experimenting.

1 The user is presented with both text and GUI options for many functions. This is helpful. The user is offered easily discernible exits.

1 You said that one of your goals was to fulfill the nine heuristics. In terms of these nine I'd say your software succeeds.

[4] the wonderful, amazing, cool volunteers: Deneb Meketa, Lukas Karlsson, Willie Seitz, Matt Heitz Amy Morneweck, Josh Davis, Ben Smith-Manschott

1 You should be able to delete a watch pair with the delete function, not just a function.

1 When I tried to graph calls to plus within goodfib, I got the error message "Nothing to graph about call to plus within goodfib." This is a good error message - short, to the point, and no inexplicable characters.

1 The interface is usually very simple and well-organized. This is very important.

In the "Watch Graph" window: These are points specific to the "Watch Graph" window.

1 The arrows sometimes overlapped the text (readable but ugly)

1 Long function names like `goodfib-helper` get truncated in odd places

1 Sometimes it's hard to click on the arrows

1 It should check if the functions are defined (on set up a watch) before barfing on errors or misspellings

2* How about providing feedback on the graph for the status of textual tracing.

1* A button for turning on text output would be good (right now there *is* one to turn off text output)

1 The graphic representation of which functions trace others is clear and concise. It makes sense that the user would select an arrow between functions to graph that relationship. So this feature scores high marks for intuitiveness.

In the "Watcher" window: These are points specific to the "Watcher" window.

1 Why is the window for the the watch graphs so large? It makes it difficult to move around in the window.

2 Data points on the vertical are not evenly spaced.

1* Clicking in the graph is awkward. Why not let users click anywhere on the horizontal line or label for a given data point

1 Being able to click on the dash, or anywhere on the line in the dotplot, (when the input is too long to be displayed) in order to view the input would be nice

1 When a dot is selected, it becomes the same color as the text of the input, which makes it a little hard to read. You could make them different colors, or even leave the dot white (it'll still look selected if there's red text next to it)

1 The graphs look nice and are clear and easy to understand.

1 It might be nice to be able to drag graphs around the Watcher window. That way two graphs could be positioned so that the user could compare them side-by-side.

1 The overlapping labels in the graphs are a problem.

In the "Help" window: These are points specific to the "Help" window.

1 Hypertext!

1* The OK button is 15 miles away from the scrollbar (the only other place in the window you might be clicking). Why not put them together?

1 Maybe you could include a section on the form that the plots take. For example, saying why the graphs are dot plots and something about the labelings would be helpful. You should also explain how to highlight a particular value in the plot to see it's coordinate on the y-axis.

Notice that in general, everyone seemed to find non-overlapping points to criticize or compliment. On the other hand, there was only one instance where two people said things which directly contradicted each-other.

5.2 User evaluation of the utility of the program as a whole

The volunteers evaluated the utility of the program based on some experimentation with the tracing aspect of the program and what they could gather from watching a demo of the aspects dealing with permanent information. They did not have the opportunity or the time to experiment with its uses on code which they had written, or even on something simple like a group of `sort` functions. Also, the demo was biased in that it was structured to run smoothly and a problem which shows IIC off was chosen. Given all of this, not much can be concluded from the evaluation of IIC applicability. The volunteers seemed to agree with the ideas behind IIC, but they had no way of testing how well it implements those ideas.

6 Applications of IIC

In this section I introduce two examples of applications of IIC to real programs. First I demonstrate its usefulness in estimating the algorithmic complexity of functions using permanent information, and then I show how a trace of some transient information can be used to identify an error.

6.1 Using IIC to estimate the algorithmic complexity of functions

In order to estimate the algorithmic complexity of a function F, we must be able to count the number of "steps" performed by F for a given input. In general, a "step" will be a call to some base function B (eg a comparison function, or a simple arithmetic function). IIC can be used to record the number of calls to B that occur for a call to F on a given input. IIC can then be used to view a dotplot displaying a data point for each call to F representing the corresponding number of calls to B. The user also has the power to tell IIC what order these datapoints should be displayed in. By ordering the data points according to increasing number of calls to B and viewing the corresponding dotplot, the user can visually estimate whether the corresponding curve is linear, exponential, or logarithmic. For an example of this, see Figure 2.

6.2 Using IIC to generate traces of functions

IIC's trace facilities for viewing transient data about functions are most useful for debugging purposes. Such a trace can help one to determine the start of or the cause of an infinite loop, or to identify the cause of an inefficiency in a function.

For example, a programmer might have implemented a factorial function which never returns when called on most inputs.

```
(define fact
  (lambda (n)
    (if (zero? n) 1
       (* (- n 1) (fact n)))))
```

What follows is a transcript of what might happen if the programmer were to trace the function.

```
STk> (define fact
  (lambda (n)
    (if (zero? n) 1
       (* (- n 1) (fact n)))))
STk> (watch basic-watcher fact)
STk> (fact 5)
(fact 5)
| (fact 5)
| | (fact 5)
| | | (fact 5)
| | | | (fact 5)
| | | | | (fact 5)
| | | | | | (fact 5)
...
```

From this transcript, it becomes quickly clear that the `fact` function is repeatedly being called recursively on the same input. Clearly, this is the reason why the factorial function is never returning a value. The trace facility allows the programmer to easily and quickly identify the problem

7 Limitations of IIC

This section addresses some of the limitations of IIC. In the future, many of these limitations could be decreased. The first group of limitations is caused by a physical principle which cannot be defeated, though its effects can be reduced. The second group of limitations is based in STk and many of these will probably be improved with new versions of the interpreter.

7.1 The Heisenberg bug

The Heisenberg principle states that it is impossible to watch some thing without having some effect on that thing just by the act of watching it. In terms of IIC, this has a number of implications:

Error Messages and the runtime stack: Because IIC sets up a function for watching by putting a wrapper around it, this wrapper will show up on the runtime stack and in the error messages when a watched function breaks for some reason. Often errors that should happen in the original code for the function end up happening in the IIC wrapper instead. The fact that IIC is watching the function changes what's on the stack and the effect of the user's interactions with STK.

CPU time and `cons` cell usage: The act of watching a function f adds to the CPU time and `cons` cell usage it takes for f to run. While it is possible to measure only the time spent in the closure representing the original, unwatched f, if that closure happens to call some *other* function g which is also being watched then the CPU time and `cons` cell usage spent in watching g will be added to the running time and `cons` cell usage of f. Since IIC cannot look inside f's original closure, it has no way of telling that g was called or that g's watch time and watch `cons` cell usage was added to f's runtime and `cons` cell usage. This problem is especially bad when dealing with recursive functions. Each recursive call to a function which is being watched will be added to the total runtime and `cons` cell usage of the original function. In IIC, this extra CPU time and `cons` cell usage is significantly large. Recursive functions are seriously slowed down by the presence of the watch code.

Function call counts: In addition to keeping track of CPU time and `cons` cell usage, IIC keeps track of the number of times which a function g is called from another function f when the user has set a watch for all occurrences of g within the scope of f. These call counts are always completely accurate. The presence of IIC has no effect on them.

7.2 Limitations of STk

Although in many ways STk has provided an ideal and unique environment in which to implement IIC, it also comes with some limitations.

First of all, in STk, primitive functions and macros cannot be redefined. Since IIC depends upon the ability to redefine a function using `set!` in order to put a wrapper around the original function, IIC cannot watch primitives or macros. This means that, for example, to watch for occurrences of a function such as +, the programmer must write their own version of that primitive to trace instead.

In addition, Error messages in STk are often not very helpful. When an error occurs, STk prints out the top four things on the stack. In the case of watched functions, this often means that most of what it prints out is IIC code. This makes the job of a programmer debugging with errors in watched functions extremely difficult and confusing.

8 Further Investigations and Questions

Although the current implementation of IIC fills in some important gaps in most interpreted programming environments, it lacks certain features that one would expect in a complete cooperative programming envi-

ronment. These are listed below. In future work, I would like to expand IIC in order to create a *complete* cooperative environment. I would also like to test IIC more formally, in order to measure its usefulness in solving the problems which it is intended for.

- IIC's underlying efficiency could be greatly improved. This would make it possible to get more accurate measurements of space and CPU data, and it would make it possible to watch more CPU-intensive functions without slowing them down too much.
- The data visualization aspects of IIC could be expanded upon. For example, IIC could allow users to compare dotplots by putting two sets of data side by side in the same plot. Also, the plots would be more readable if users could supply a function that given an input will generate a label. This would mean, for example, that functions taking long lists as input could, instead of being labeled by a hundred-element list, be labeled by the number 100.
- There may be better ways of dealing with the problem of overlapping labels. It would be an interesting data-visualization problem to explore further the implications of the fact that IIC cannot relate different data points, and to explore ways of letting the user tell IIC how to relate them using scheme functions.
- If IIC could be expanded to provide access to its stack rather than just displaying transient information, it would be much more effective as a debugger.
- It would be worthwhile to examine how IIC deals with `call/cc`. Currently, it uses a dynamic wind to guarantee that setup and cleanup functions are always executed when an error occurs. This might be problematic in terms of `call/cc`.
- In general, the theoretical foundations of IIC could be more deeply explored. For example, one could develop a set of semantics describing what it means to set up watches about one function in the scope of another function. This might then make it easier to explore the problem of `call/cc` mentioned above.
- IIC would be more useful as a teaching/algorithm-animation tool if graphs would dynamically update as a function is running.
- IIC's usefulness could be augmented by implementing more of the things described in Section 2.2 (Cooperative Software).

9 Conclusions

IIC (Information in Context) is an approach to gathering, visualizing and understanding information about your programs. It is based on the fact that context can help you to determine what information to gather, and that context can help you to understand information once you have it. IIC lets the programmer specify what information to gather using scope and provides the programmer with visualizations of both transient and permanent information.

Limitations of Contextual Information :

- The user has to specify *every* function which she wants to watch
- The user has to always check whether to record an event rather than, for example, recording *every* event

Advantages of Contextual Information :

- The user can gathered more detailed, more useful information. Many profilers gather a huge quantity of information, much of which is useless. As a result, users cannot gather as much detailed information since there is a limit to the amount of information that can be stored or parsed. With IIC, users can limit the amount of useless information which is gathered, allowing them to gather more detailed information.

Acknowledgments

Many thanks to my Advisor Rhys Price Jones, and to Tom Ball who guided me in my selection of the project. Thanks to the Volunteers who tested IIC. Thanks to Joe Bayes for supporting me and helping me and being a great bouncy board to bounce ideas off of. Thanks to Karen Ward for her advice and suggestions.

A The user evaluations of IIC

This appendix contains the original mail messages from each volunteer evaluating IIC.

```
From: SUBJECT A
To: jmankoff
Subject: testing
Date: Thu, 4 May 1995 16:34:31 -0400

Jen:

I'm SUBJECT A.  These are my notes on your interface.

the function random-list often allows zeroes into its output, which
makes the product of the list zero (boring).

The grey background in the dialogs makes the cursor hard to see.

It would be nice to be able to trace loop within many-mult.

There are problems with the stk interface.  A command-line interpreter
is a pain to learn.  I didn't enjoy having to poke around files to
find out what to do in order to use the program.  If you put buttons
or other graphical tools into the X windows that ran the data-display,
tracing, etc. utilities, that'd be nice.  You might come up with ideas
of typical pipelines for analyzing a function, then line up the
buttons in that order.

In general, the goal of not taxing the user's memory load isn't met.
There's a lot to do, and nothing is around to remind me of each of the
steps (except Help, which in my opinion shouldn't have to be consulted
terribly often).

The thing with the overlapping labels in the graphs is yucky.  Why not
exclude enough labels to make them not overlap?  It would be clear
enough what was going on if you left all the data points in but not
necessarily all the labels.  And clicking in the graph is awkward.

before opening new instances of X windows, check if they're already there.

wow, those error messages suck.  Another reason to lose the stk
interface.  Maybe you could pop up another X window that echoed most
of the Scheme proceedings but filtered out awful details like the
error messages (could be done by ignoring everything between "Error:"
and "STk>").

that's about all I had time for.  good luck!

SUBJECT A
```

```
************************************************
From: SUBJECT B
To: jmankoff
Subject: Honors
Date: Thu, 4 May 1995 17:09:18 -0400


If the arrows in the graph gave information about if textual tracing
is on or not it would be easier to understand the current "state" of
things.

Can't play with written stats with a button.

It might be nice if you added something that automatically ran
something many times with various inputs the user chooses so you can
get lots of data points.

Data points on the vertical are not evenly spaced. So I guess giving the
graph logical stuff to work with is up to the user.

--SUBJECT B (real one)

************************************************
From: SUBJECT C
To: jmankoff
Subject: things for you to read
Date: Thu, 4 May 1995 17:03:01 -0400

overall stuff:
   return value of watch is #f.  why?
   It's really hard to see a black cursor against a dark-gray backgound.
        Why not make it the same color as the text?
   the ability to use arrow keys to move inside the entries would be nice.
in the watch graph window:
   the arrows sometimes overlapped the text (readable but ugly)
   long function names like goodfib-helper get truncated in odd places
   sometimes it's hard to click on the arrows
   it should check if the functions are defined (on set up a watch)
before barfing on errors or misspellings
in the watcher:
   being able to click on the dash (when the input is too long to be
displayed) in order to view the input would be nice
   also, just clicking anywhere on the line in the graph instead of
just the dot would be very nice
   when a dot is selected, it becomes the same color as the text of the
input, which makes it a little hard to read.  You could make them
different colors, or even leave the dot white (it'll still look
selected if there's red text next to it)
help:
   hypertext!
   the OK button is 15 miles away from the scrollbar (the only other place
    in the window you might be clicking).  Why not put them together?
```

****
From: SUBJECT 1
To: jmankoff
Subject: my evaluation
Date: Thu, 4 May 1995 20:24:28 -0400

I found at first that i was not all that sure of what the thing was supposed to do, and when i got that far, it took a little while to figure out exactly how to do it. i dont see that as a function of the presentaion, but i think it is just in the nature of the program. something like this will always take a little getting used to. with a small amount of time, i did become used to it, and was able to be quite productive with it. it only takes a little bit to play with all the different features before you feel comfortable enough with it to use it quite quickly.

i found that the tutorial was quite easy to follow. the examples given made it easier to understand what was going on. it ends up being a little confusing having both the graphical interface and the stk interface. it seems like there would be a better way to implement that... possibly by having a graphical window where stk could output stuff, which had a similar look and feel to the other windows. it would also be nice to be able to do everything in the graphics windows, somehow. like loading files, etc. otherwise, i thought it was wonderful. very beautiful and friendly to work with. i enjoyed it.

- SUBJECT 1

****
From: SUBJECT 2
To: jmankoff
Subject: honors project feedback...
Date: Sat, 6 May 1995 16:13:24 -0400

1) 9 heuristics:
1) Simple and natural dialogue
2) Speak the user's language
3) Minimize user memory load
4) Be consistent
5) Provide feedback
6) Provide clearly marked exits
7) Provide shortcuts
8) Good error messages
9) Prevent errors
2) demo (goodfib and badfib): see ../demo.stk
5) trace: many-mult.stk

* "Stop Tracing":

How about the ability to stop a trace by clicking on the arrow that represents the trace in the graph. This would be quick and easy (not requiring the user to remember or re-type the names of a function) and it would be consistent with teh behavior of the delete graph button in

the watcher window.

* "Stop Tracing":

How about proving feedback on the graph for the status of textual
tracing. Perhaps the functions not being traced could be drawn as
cicles of a different color. (Perhaps those being traced could be
drawn as miniature document icons, with the little folded over corner.
This might require less explanation than simply drawing them in two
different colors.)

* "Setup Graph" dialog box:

The following function will be used in a call to "sort" to compare
pairs of
[values | inputs]
[<textfield]

Sort data about the functions as follows:
popup: [value | user function | length]

Perhaps something like this would come across more naturally:

Sort by: [output values | input values]

by comparing: [value | length | with a user-defined function:
;; implemented as check boxes.

;; this only appears if "user-defined function" is checked above.
Enter the user defined function below:
[<text field>]

* Turning on Text output

A button for this would be a great idea.

****
From: SUBJECT 3
To: Jen Mankoff <jmankoff@five.cs.oberlin.edu>
Subject: graphs etc.
Date: Sat, 6 May 1995 16:36:25 -0400 (EDT)

Comments:

For the first 45 minutes I was just trying to get the correct files
loaded. I realize that during testing things are not polished, but
it would help just to have a list of files and their contents.

The interface has several nice features. The graphic representation
of which functions trace others is clear and concise. It makes sense
that the user would select an arrow between functions to graph that
relationship. So this feature scores high marks for intuitiveness.
the graphics are not gorgeous but they are more than adequate. The
buttons are self-explanatory.

I give the Watcher window mixed reviews. The graphs look nice and are clear and easy to understand. It is helpful to be able to click on a datum and have it highlighted. Multiple-higlighting and the ability to drag the mouse over points and have them each highlight in turn are also nice touches. I was impressed by the delete button. Once the Compare Data and Resize buttons are fixed/removed the buttons will be clear. It might be nice to be able to drag graphs around the Watcher window. That way two graphs could be positioned so that the user could compare them side-by-side. Also, the numbers on the y-axis often appear crammed together (although you already mentioned your trouble with selecting out certain y-values. Another nice feature would be the ability to flip the axes on a graph. I mentioned this to you. This may not be crucial for mathematical analysis but it might make the user happier.

Another idea I have on the Watch Graph and Watcher windows is to integrate them. To my sense of organization the interface would be cleaner with only one window. I realize that they really serve different purposes. It was just an impression I had.

The dialog boxes are slick. The controls are easy to use. The user's choices make sense. The user is presented with several choices but not so many that the dialog becomes muddled. Like any well-designed interface, the inexperienced user can learn a lot about the program's use just from a few minutes of experimenting.

I think when I select a command in a program I think about what dialog box it will bring up and this helps me remember which features accomplish which tasks. I am not very familiar with your program, so maybe that is the problem, but when I do something which I know will bring up a dialog box I don't know ahead of time which box it will bring up. I think a couple of your dialog boxes are quite similar. Maybe if they looked different that would help me distinguish them in my mind. Sorry if this paragraph doesn't make sense.

The user is presented with both text and GUI options for many functions. This is helpful. The user is offered easily discernible exits. I was never in a panic because of being locked into a dialog and not knowing what was going to happen next. If this software is aimed at programmers and students then the users should have little problem getting accustomed to the interface. Processes are simple enough that users do not have to memorize very much.

Things I like best:

The interface is usually very simple and well-organized. This is very important.

Things I'd like to see improved:

Hmm. I wish I had more time to go into some of my labs and play

around with my own functions. There is not much I can criticize
about the software. You said that one of your goals was to fulfill
the nine heuristics. In terms of these nine I'd say your software
succeeds.

SUBJECT 3

****
From: SUBJECT 4
To: Jen Mankoff <jmankoff@six.cs.oberlin.edu>
Subject: honors testing
Date: Sat, 6 May 1995 16:38:28 -0400 (EDT)

These are my comments on the interface.

1. This is a silly point, but it was something that I noticed. In the help
text for the window Watcher, you explain how to do the functions of the
buttons by typing. In the examples of what a <watch> could be you have
<fun-name> and then <fun-pair> and then examples of each. These should
be preceded by e.g. not i.e. i.e. stands for "that is", while e.g. means
"for example." I'm not sure if i.e. is incorrect in this context, but I
think e.g. is closer to what you mean.

2. You should be able to delete a watch pair with the delete function,
not just a function. I think people will more often want to stop
watching a particular pair, not all of the watches involving one
function. It would be best if both functionalities were available.

3. In part 4 of the help text, "Stop Watching" should be changed to
"Stop Tracing". Also, you use "dialog" and "dialogue" inconsistently.
The "??? ???" button is now the "Setup a Graph" button.

4. Why is the window for the the watch graphs so large? It makes it
difficult to move around in the window. It would be good if the area of
the window in view would move automatically to show whatever graph was
produced most recently. This would save searching around the window for
the new plot. Another solution would be to have an explicit description
in the help of where the next graph will appear in relation to the
previous plots. It seems to make two on the same row and then move to
below the first plot. Why is this?

5. When I tried to graph calls to plus within goodfib, I got the error
message "Nothing to graph about call to plus within goodfib." This is a
good error message - short, to the point, and no inexplicable characters.

This program was easy to use with the data you had set up. I could figure
out what was supposed to happen as well as what was happening. Since I
don't know scheme, I couldn't test any new data, but it doesn't seem that
it would be difficult. The help text was good. Maybe you could include
a section on the form that the plots take. For example, saying why the
graphs are dot plots and something about the labelings would be helpful.
You should also explain how to highlight a particular value in the plot to

see it’s coordinate on the y-axis.

This was fun. And thanks for the banana bread.

- SUBJECT 4

References

[AC76] F. E. Allen and J. Cocje. A program dataflow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 246–256. ACM SIGPLAN, ACM Press, June 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Bal93] Thomas Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, March-December 1993.

[Bar77] Jeffery M. Barth. An interprocedural data flow analysis algorithm. In *The Fourth ACM Symposium on Principles of Programming Languages*, pages 119–131, Los Angelos, California, January 1977. ACM Press.

[Bar93] Andrew J. Barnard. *From Types to Dataflow: Code Analysis for an Object-Oriented Language*. PhD thesis, University of Manchester CS, 1993.

[Ber93] Karen L. Bernstein. Debugging strict functional programs. November 1993.

[Cho89] Jongdeok Choi. *Parallel Program Debugging with Flowback Analysis*. PhD thesis, University of Wisconsin – Madison, August 1989.

[CJ91] William Clinger and Jonathon Rees, editors. Revised^4 report on the algorithmic language scheme, 1991.

[Cle43] William S. Cleveland. *The Elements of Graphing Data*. Number ISBN 0-534-03730-5. Wadsworth Advanced Books and Software, CA, 1943.

[Cle93] William S. Cleveland. *Visualizing Data*. Number ISBN 0-9634884-0-6 Cloth. Hobart Press, NJ, 1993.

[Dat82] February 1982. Computer Magazine: Focus on Dataflow.

[DFAB93] Alan Dix, Janet Finlay, Gregory Abowd, and Russel Beale. *Human-Computer Interaction*. Number QA76.9.H85H85, ISBN 0-13-458266-7 (hbk.) – ISBN 0-13-437211-5 (pbk.). Prentice Hall International, UK, 1993.

[Duc93] Mireille Ducassé. A pragmatic survey of automated debugging. In *1st International Workshop on Automated and Algorithmic Debugging*, pages 1–15. Department of Computer and Information Science, Linköping University, May 1993.

[Dyb87] R. Kent Dybvig. *The Scheme Programming Language*. Number ISBN 0-13-791864-X 025. Prentice Hall, NJ, 1987.

[ELJ94] Todd R. Eigenschink, Dave Love, and Aubrey Jaffer. Slib, the portable scheme library. April 1994.

[FF89] Daniel P. Friedman and Matthias Felleisen. *The Little Lisper*. Number ISBN 0-574-24005-5. Science Research Associates, Chicago, 3rd edition, 1989.

[FL88] Charles N. Fischer and Richard J. LeBlanc, Jr. *Crafting A Compiler*. Number ISBN 0-8053-3201-4. Benjamin/Cummings, Menlo Park, California, 1988.

[Gal] Erick Gallesio. Stk reference manual.

[GH92] Michael Golan and David R. Hanson. Duel - a very high-level debugging language. Technical Report CS-TR-399-92, Princeton University, November 1992.

[HL] Per Hammarlundt and Björn Lisper. On the relation between functional and data parallel programming languages. Technical report, Royal Institute of Technology, ??

[HLR91] Halbwachs, Lagnier, and Ralel. Generating efficient code from dataflow programs. Technical Report SPECTRE L8, IMAGE, Grenoble, 1991.

[HO85] Cordelia V. Hall and John T. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 60–68. ACM Press, June 1985.

[Hor90] Susan Horowitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–244. ACM SIGPLAN, ACM Press, June 1990.

[JMCT83] Beat Kleiner John M. Chambers, William S. Cleveland and Paul A. Tukey. *Graphical Methods for Data Analysis*. Number ISBN 0-87150-413-8 in Wadsworth Statistics/Probability series. Wadsworth International Group, CA and Duxbury Press, MA, 1983.

[Kam90] Samuel Kamin. A debugging environment for functional programming in centaur. Technical Report 1265, INRIA Sophia-Intipolis, June 1990.

[Kis92] Amir Shai Kishon. *Theory and Art of Semantics-Directed Program Execution Monitoring*. PhD thesis, Yale University, May 1992.

[KSF92] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzon. Interprocedural dynamic slicing. In *Programming Language Implementation and Logic Programming*, pages 370–384. Springer-Verlag, August 1992.

[Lef81] Robert Lefferts. *Elements of Graphics – how to prepare charts and graphs for effective reports*. Number ISBN 0-06-012578-0. Harper & Row, NY, 1981.

[MJ81] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Number ISBN 0-13-729681-9 in Software Series. Prentice Hall, 1981.

[Nai92] Lee Naish. Declarative debugging. Technical Report 92/6, University of Melbourne, University of Melbourne, Victoria, Australia, 1992.

[NF92] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. In *Programming Language Implementation and Logic Programming*, pages 385–399. Springer-Verlag, August 1992.

[NF93] Henrik Nilsson and Peter Fritzson. Lazy algorithmic debugging: Ideas for practical implementation. In *1st International Workshop on Automated and Algorithmic Debugging*, pages 151–165. Department of Computer and Information Science, Linköping University, May 1993.

[NM90] Jakob Nielson and Rolf Molich. Heuristic evaluation of user interfaces. In *ACM Conference on Computer Human Interaction*, pages 249–256. SIGCHI, April 1990.

[nML92] Guy LaPalme nad Mario Latendresse. A debugging environment for lazy functional languages. *Lisp and Symbolic Computation*, 5:271–287, 1992.

[OCH91] Ronald A. Olsson, Richard H. Crawford, and W. Wilson Ho. A dataflow approach to event-based debugging. *Software – Practice and Experience*, 21(2):209–229, February 1991.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Number ISBN 0-201-63337-X in Professional Computing Series. Addison-Wesley, 1994.

[Ret93] Marc Rettig. Cooperative software. *Communications of the ACM*, 36(4):23–28, April 1993. Practical Programming.

[Ryd83] Barbara G. Ryder. Incremental data flow analysis. In *The Tenth ACM Symposium on Principals of Programming Languages*, pages 167–176, Austin, Texas, January 1983. ACM Press.

[San94] Patrick M. Sansom. *Execution Profiling for Non-strict Functional Languages*. PhD thesis, University of Glasgow, 1994.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program Debugging*. PhD thesis, ACM Distinguished Dissertations, 1983.

[Sny90] Robin M. Snyder. Lazy debugging of lazy functional programs. *New Generation Computing*, 3:139–161, 1990.

[Tol92] Andrew Tolmach. *Debugging Standard ML*. PhD thesis, Princeton University, October 1992.

[TS84] John C. Thomas and Michael L. Schneider, editors. *Human Factors in Computer Systems*. Number ISBN 0-89391-146-1 in Human/Computer Interaction. Ablex Publishing Co, New Jersey, 1984.

[Wat94] Richard C. Waters. Cliché-based program editors. *ACM Transactions on Programming Languages and Systems*, 16(1):102–150, January 1994.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WN88] Russel Winder and Joe Nicolson. Jdb: An adaptable interface for debugging. *Software – Practice and Experience*, 18(3):221–238, March 1988.