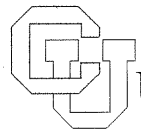
Simplifying the Construction of Interactive Programs In a Functional Programming Environment

Eric Blough

CU-CS-884-99



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Simplifying the Construction of Interactive Programs in a Functional Programming Environment

by

Eric Browder Blough

Baccalaureus Artiem et Scientiae, University of South Carolina, 1988

M.S., University of Texas, 1991

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1998

Blough, Eric Browder (Ph. D., Computer Science)

Simplifying the Construction of Interactive Programs in a Functional Programming Environment

Thesis directed by Professor Clayton Lewis

Functional programming offers numerous benefits to the programmer, including higher-order functions, absence of control flow, automatic storage management, and static type inference and type checking. The writing of interactive programs has been a long-standing challenge to the functional programming community, since the statefulness of input and output does not merge smoothly with the statelessness of a pure functional language. Historic and recent approaches to interactivity clearly demonstrate the utility of functional programming of interactive systems, but these approaches are cognitively problematic.

Spreadsheets offer numerous, but largely different, benefits for certain classes of programming problems. These benefits include visible, manipulable data representations, automatic consistency maintenance, absence of control flow, and immediate feedback. Although the spreadsheet environment provides significant interactive power, it is limited and minimally customizable, and has cognitive weaknesses of its own.

We present Esquisse, an alternative to both of these approaches that shares much with each. Esquisse combines the full power of a functional programming language with an extensible spreadsheet-inspired environment made of pure functional components. Esquisse demonstrates that powerful interactive programs can be produced quickly with relatively low cognitive cost, and suggests a promising direction for future research in functional programming environments.

Acknowledgement

This research was funded in part by the United States Department of Energy grant number DE-FG03-95ER14499 to the University of Colorado.

Contents

Chapter

Figures

Figure

Algorithms

Chapter 1

Introduction

Programming a computer is a complex skill that takes considerable time and effort to master. A number of cognitive barriers that impede the programming process have been identified for traditional languages [LO 1987]. We focus on two alternatives to traditional programming, each of which reshapes these cognitive barriers in its own way.

Functional programming languages provide important tools for modularization and program development that are not present in traditional languages [Hughes 1990]. Historically, functional language approaches to writing interactive (and thus stateful) programs have been complex and cumbersome. The advent of monadic I/O has brought an elegant, efficient, semantically sound approach to stateful interaction. Another recent development is a collection of interactive toolkits written in functional languages.

Spreadsheet applications fall even farther outside the realm of traditional programming, and yet they have extended some level of programming power to a much wider audience than traditional languages have. More recent spreadsheet programs offer extension languages for creating graphics and interfaces, increasing the power of spreadsheets in new directions.

In addition to our study of these two programming approaches, we explore the benefits of a synthesis of functional programming languages and the spreadsheet model of interaction.

1.1 Functional Programming Languages

1.1.1 Functional languages provide novel tools, challenges, and approaches to interactivity.

1.1.1.1 Functional languages provide powerful new programming tools.

Functional programming languages have long been a subject of active research by a substantial and passionate research community, yet they are best known for what they lack: side effects, control flow, loop constructs, and so on. Advantages such as automatic storage allocation are all very well and good, but are hardly unique to functional programming languages. What distinguishes functional programming languages from other language designs is the presence of several important modularization tools that support the smooth creation of abstractions and high-level aggregate operations.

For the purposes of this document, "functional programming languages" refers to side-effect free, strongly typed, lazily-evaluated languages such as Haskell [HPJW+ 1992, HPF 1997] and Miranda[1] [Turner 1986]. Examples are shown in Haskell, or in a few cases, Haskell-like pseudocode.

Higher-order functions build specialized functions from general-purpose functions.

One of the distinguishing features of pure functional programming is that functions are "first-class" objects: they can be bound to names, used as elements of data structures, passed as

[1] Miranda is a trademark of Research Software Ltd.

arguments to functions, and returned as function results. They are also a powerful abstraction tool.

For instance, higher-order functions support the creation of special-purpose functions from general-purpose functions. The `foldr` list-reduction function can be parameterized by a function and an identity value to create a family of specialized functions [Hughes 1990]:

```
foldr :: (a -> b) -> a -> [a] -> b
foldr f x [] = x
foldr f x y:ys = f y (foldr f x ys)

sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
append xs ys = foldr (:) ys xs
```

(These examples, with the exception of `append` from [Hughes 1990], are taken from the Haskell Standard Prelude [HPJW+ 1992].) Notice that for `sum`, `product`, `and`, and `or`, `foldr` has been given its first two arguments (the function and the identity value), but not the third (the list to be reduced). A function applied to some (but not all) of its arguments is said to be **curried**; currying is a simple, powerful, and quite useful way of defining one function in terms of another.

A general purpose numerical differentiation function is a more literal example of a function returning another function. The general function can be parameterized to create a family of functions which compute derivatives to different accuracies:

```
deriv :: (Fractional a) => a -> (a -> a) -> (a -> a)
deriv deltaX f = \ x -> (f (x + deltaX) - f x) / deltaX

deriv01, deriv0001 :: (Fractional a) => (a -> a) -> (a -> a)
deriv01 = deriv 0.01
deriv0001 = deriv 0.0001
```

which can then be used to calculate a function `f'` from a function `f`:

```
f' :: (Fractional a) => (a -> a)
f' = deriv0001 f
```

Lazy evaluation separates the issues of generating possible solutions and choosing the best solution.

Writing a function in many programming languages often involves specifying not only the result to be computed, but how much of it to compute. Many applications make use of a generate-and-test strategy: a number of possible solutions are generated, and the first solution which passes the test is returned. Game-playing programs, for instance, generate a tree of possible move sequences, and assign values to each node (i.e., each game state). Since a lazy functional language only evaluates an expression when the value of that expression is needed by another part of the program, these two parts of the program can be separate functions. The generator function will be evaluated only to the extent necessary to create exactly as much of the game tree as the test function needs to find a solution. In languages which do not provide lazy evaluation, these two functions must be integrated together in order to avoid generating more of the tree than is necessary [McPhee 1992].

Consider a program which uses the Newton-Raphson method to find the roots of a function (an example from [Hughes 1990]). The Newton-Raphson method involves computing successive approximations to a root:

```
nextApproximation ::
    (Fractional a) => (a -> a) -> (a -> a) -> (a -> a)
nextApproximation f f' = \x -> x - (f x / f' x)
```

An iterate function (provided in the Haskell Standard Prelude) can compute a list of such approximations:

```
iterate :: (Fractional a) => (a -> a) -> a -> [a]
iterate f x = x:(iterate f (f x))
```

Another function can search the list for two approximations whose ratio is less than some epsilon:

```
within :: (Fractional a) => a -> [a] -> a
within epsilon x1:x2:xs | abs(x1-x2) <= epsilon = x2
                        | abs(x1-x2) > epsilon = within epsilon x2:xs
```

These functions can be combined to produce the root-finding function

```
findRoot :: (Fractional a) => (a -> a) -> a -> a -> a
findRoot f x0 epsilon
  = within epsilon
          (iterate (nextApproximation f (deriv0001 f)) x0)
```

Lazy evaluation allows the generator (`iterate (nextApproximation f (deriv0001 f)) x0`) to be separated from the test (`within epsilon`). Each of these components is available for use in other computations. The generator can generate an infinite list of approximations, because only those approximations which are needed by the test function will be computed.

Some non-functional languages include constructs that present the elements of a data structure one at a time. CLU iterators, for instance [Liskov 1996], provide a variant of a loop construct that iterates a body of code over the elements of a data structure. While this does provide some generality, lazy evaluation supports infinite data structures and partial, directed exploration of those structures; CLU iterators provide the elements of a data structure in an unordered sequence, which is considerably less flexible.

Referential transparency makes implicit parallelism possible.

Lazy evaluation not only relieves the programmer from concerns about evaluation order, it requires that the programmer give up direct control over evaluation. It is lazy evaluation that forces functional programming languages to be stateless [Hughes 1990].

The result is referential transparency: any expression in a functional program can be replaced with its value without affecting the program's result. Since evaluation order is constrained only by the dependence of each expression on its subexpressions, expressions which are independent can be evaluated in parallel. Functional language implementations can thus provide parallelism implicitly, without requiring attention from the programmer.

Type inference supports powerful static type-checking with or without explicit type signatures.

In the examples in the previous sections, we have provided explicit type signatures. Functional programming languages are strongly typed and statically type-checked; furthermore, the types of all but a small subset of expressions in the language can be inferred, without explicit type signatures. In fact, even when explicit signatures are provided, type checking is performed by inferring the type of the function and comparing the inferred type with the explicit signature.

Type inference is important because it safely supports overloaded functions. For example, the `(:)` infix operator (pronounced "cons," like its namesakes in Scheme and Lisp, or sometimes "followed by"), has the type:

```
(:) :: a -> [a] -> [a]
```

That is, given a value and a list of values, it evaluates to a new list consisting of the first value followed by the list of values:

```
1:[2,3] ---> [1,2,3]
```

The value must have the same type as the list (which is why the type variable `a` appears throughout the type signature), but the list can be any type at all: integer, fractional, Boolean, even functions from integers to Booleans (for example, `[even, odd, prime, composite, perfect]`). The type system of Haskell and of other functional languages implements these constraints, supporting the creation of a single (parametrically) polymorphic function, rather than a raft of type-specific ones. (Haskell's type classes support ad-hoc polymorphism as well, which is less relevant to our discussion here.)

1.1.1.2 Functional languages change the cognitive barriers to programming.

Some cognitive barriers are lower.

The absence of a control model in functional languages frees the user from one of the more difficult concepts in traditional programming. Functional languages also allow the user to specify the dependencies among the data, instead of having to maintain them explicitly. Function definitions tend to be shorter than their imperative counterparts, encouraging low viscosity (greater ease in changing one part of the program without needing to change other parts), and the presence of higher-level aggregate operations reduces somewhat the need for the user to play "programming games" to construct them from primitives [LO 1987].

Some cognitive barriers remain.

Functional programming shares features with nonfunctional languages which are known to pose cognitive obstacles to novices, such as the use of variables [Lewis 1989] and the need to "pump" data from the human world to the computational world and back [Draper 1986, LO 1987, Lewis 1989]. Higher-order functions, which are not present in most nonfunctional languages, also appear conceptually difficult for novices [ERT 1987].

1.1.1.3 Since pure functional languages must be stateless, the statefulness of I/O makes programming interactivity awkward from a functional perspective.

Computational environments provide access to the world outside the programming language: operating system devices such as files, network connections, instruments, displays, and interaction devices, and through these, the world of the human users. Both of these worlds are inherently stateful. Computer systems and human memory store information and are history-sensitive. Most objects in the human world are persistent and mutable, and for humans to have a similar model of the contents of a file system or the visual appearance of the screen is arguably quite natural.

The introduction of an imperative syntax for I/O in Haskell reinforces this point. The imperative syntax is syntactic sugar for pure functional syntax, but its purpose is to make functions which involve I/O more readable and writeable — an implicit acknowledgement of the naturalness of a stateful model of I/O.

Statefulness prevents referential transparency and lazy evaluation.

Unfortunately, the introduction of state destroys referential transparency, and with it, lazy evaluation. The challenge in the eyes of the functional programming community is to provide access to the stateful world of interactive input and output in a manner which does not directly introduce state into the functional language. Responses to this challenge are detailed in section 1.1.2.

Statefulness presents cognitive obstacles.

The presence of state is at the root of several cognitive obstacles to programming. State transitions force a need to understand evaluation order, and thus control flow. The use of state variables tends to raise viscosity, since the same variable can be used in two widely separated

program locations, and a change in one location can affect the other. Maintaining consistency in the presence of state is more difficult also, since state variables introduce dependencies between potentially distant parts of the program. The cognitive issues suggest restricting these effects of state.

1.1.2 Functional programming approaches to interactivity must deal with the statefulness of interactive systems.

1.1.2.1 Historically, functional programming research on interactivity has focused on the problem of state in terminal-based environments.

Pure functional programs are completely declarative; they specify computations via expressions, not sequences of statements. This absence of state is essential to referential transparency, lazy evaluation and implicit parallelism.

In sharp contrast, the world of input and output is highly stateful. References to I/O devices are not transparent, and evaluation order is critical. Approaches to input and output in functional languages are thus tightly bound to the problem of state. In fact, the approaches to I/O described in this section are also used for other applications involving state, and they can be and are combined in a single program. For example, the Fudgets system uses streams for message passing between components and continuations for storing the internal state of a component. Somewhat similarly, the (formerly standard) Dialogue I/O model of Haskell 1.2 [HPJW+ 1992] used both streams and continuations.

The game of hangman demonstrates the state problem.

Games are interactive and highly stateful. As a simple example, consider the game of Hangman. One player thinks of a word and the other attempts to discover it by guessing letters. As the guessing player guesses each letter, the player who chose the word reveals whether and where that letter occurs in the word (example from [BW 1988]):

```
       -------
a      -a---a-
n      -an--an
h      han--an
g      hang-an
m      hangman
```

A function can implement this part of the process:

```
reveal :: [Char] -> [Char] -> [Char]
reveal target guesses = map dash target
  where dash w | w `elem` guesses = w
               | w `notElem` guesses = '-'
```

Given this definition of reveal, the computation necessary to obtain this "progress" string is simply:

```
progress = reveal target guesses
```

Note that reveal requires the entire list of guesses, not just the most recent guess. The list of letters which have been guessed is, in essence, the state of the game, and some part of the surrounding program must maintain the list of guesses. This is the state problem: the value of the guesses list stores the history of the letters guessed.

Functional programming approaches impose a total order and unique bindings on state-related operations.

Since an expression in a pure functional language must always evaluate to the same value, the state (e.g., the current state of the input channel) cannot have a single binding. Functional

programming research solves this problem by treating the changing state as a sequence of state values. The input to a program can be viewed as a sequence of values; alternatively, the state of the I/O system can be viewed as a sequence of state values, each computed from its predecessor by some state transformation function. In either case, each item in the totally ordered sequence has a unique binding. (This total ordering is only necessary for I/O operations; imposing a total order on all evaluations would eliminate lazy evaluation.)

Five major functional programming approaches address the problem of state at the language level. Each provides a means for ensuring a total order on I/O operations (sequencing them) and a unique binding for each I/O state access. The approaches vary in the amount of support provided for the programmer, and the amount of care required on her or his part, to guarantee these conditions. (The following discussion is based in part upon pages 5 through 14 of [NR 1994].)

Streams use lazy evaluation to interleave input and output.

Streams [Thompson 1990] are lazy lists: lists in which each successive element is computed only when it becomes needed by another computation. The entire input to the program can be treated as a single list of characters, as can the output:

```
type Input = [Char]
type Output = [Char]
```

thus binding successive state values to successive elements of the input or output streams. Enforcing a total order on I/O operations (sequencing) is partly the programmer's responsibility, since the entire input stream is potentially available to any part of the program.

To pass state from one part of a program to another, we define the type:

```
type Inter a b = (Input, a) -> (Input, b, Output)
```

where `a` and `b` are the types of state (other than the I/O state) accepted and returned by the interaction function. (If the interaction function needs to pass around more than one state item, those items must be bundled into a tuple.) Interactions are composed using higher-order functions called combinators, such as:

```
sequence :: Inter a b -> Inter b c -> Inter a c
choose   :: (a -> Bool) -> Inter a b -> Inter a b -> Inter a b
while    :: (a -> Bool) -> Inter a a -> Inter a a
```

These combinators help the programmer enforce proper sequencing. The hangman example in the streams method would look like this (ignoring the issues of entering the target string and echoing the input):

```
guess :: Inter [Char] [Char]
guess (input, guesses) = (input', newguess:guesses, output)
  where (newguess, input') = readChar input
  output = printString (reveal target (newguess:guesses))
```

The use of `input` and `input'` force the input stream to be read in the proper fashion, but it is up to the programmer to ensure that they do so, and the extra bindings produce extra program clutter, since the programmer must provide a name for the state of the input stream after each operation.

This method does not specify how input and output are interleaved. The hangman example forces input and output to alternate: for `output` to be computed, `newguess` must be computed, which requires that `input'` be computed by reading a character. In other cases, the computation will not automatically enforce this ordering. Thompson [Thompson 1990] mentions two cases in which this flexibility can produce undesired results. Output (such as a prompt) can

be delayed by pattern matching on the following input, and output which does not depend on preceding input can be printed prematurely (and repeatedly), without waiting for the input to be read.

Several solutions to these problems have been proposed: explicit echoing of the input, a wait function which forces evaluation, or making input and output streams of requests and responses rather than streams of characters [NR 1994, page 9].

Systems store the entire I/O state in a single data structure.

Another alternative is to pass the entire I/O state around via a data structure called a system. [NR 1994, page 10]. Every function performing I/O takes the system state as an additional parameter, and returns the new system state as an additional result. In the hangman example, the system consists of the input and output (both lists of characters, as in the streams approach).

```
type System = ([Char], [Char])
guess :: [Char] -> System -> ([Char], System)
guess guesses system = (newguess:guesses, system'')
  where
    (newguess, system') = readChar system
    system'' = printString (reveal target (newguess:guesses))
                           system'
readChar :: System -> (Char, System)
printString :: [Char] -> System -> System
```

The `system`, `system'`, and `system''` parameters enforce sequencing, but clutter the program even more than in the streams example (as in the streams model, a distinct name must be provided for the system after each I/O operation). The problems arising from interleaving input and output described on page 6 are applicable here, as are the approaches addressing them.

In this case, the `System` type is relatively simple, since only one input channel and only one output channel are required, but in graphical applications, the `System` type includes widgets, mouse and keyboard input, and so on, which can make dispatching input and output to the appropriate functions quite involved.

History models use systems implicitly.

The history model [NR 1994, page 13] is similar to the systems model, but the system parameter is implicit rather than explicit. In the hangman example, the progress string might be specified by:

```
progress = reveal target guesses
```

but this specification would behave as if it were written:

```
(progress, history') = reveal target guesses history
```

This reduces program clutter, since all bindings to the I/O state are hidden. The I/O operations are totally ordered, but at the cost of imposing a total order on the entire program. This completely eliminates the benefits of lazy evaluation and the potential for implicit parallelism in the non-I/O parts of the program.

Continuation passing bundles state into function applications.

Continuation passing style (CPS) provides another solution. In CPS, instead of a function returning a value (example from [NR 1994, page 10]):

```
add2 :: Int -> Int
add2 num = num + 2
```

the function takes a continuation, to which it passes its result:

```
add2 :: Int -> (Int -> a) -> a
add2 num cont = cont (num + 2)
```

The continuation function represents the computation in the "rest of the program". Thus, `add2` takes the rest-of-the-program function as an argument and applies it to `add2`'s result. This results in the program state being passed from function application to function application. The function calls enforce the sequence of I/O operations, but it is up to the programmer to ensure that the function calls are correct.

The streams, systems, and continuations approaches have been proven to be equivalent in expressive power [HS 1988]. One advantage to CPS over streams and systems is that the state can be passed as multiple arguments, rather than being bundled into a tuple. Another is that the result passed to the continuation need not be named explicitly (as in the `add2` example above).

In CPS, the `guess` function becomes:

```
guess :: [Char] -> ([Char] -> Input -> Output) -> Input -> Output
guess guesses cont input
  = output ++ cont (newguess:guesses) input'
  where output = reveal target (newguess:guesses)
        (newguess, input') = readChar input
```

The arguments `newguess:guesses` and `input'` are passed to the continuation directly, without needing to be bundled into a tuple or given.

Unfortunately, the input and output streams must be dealt with explicitly, and the interleaving problems described on page 6 are still possible.

Monads encapsulate the I/O world and force strict evaluation of I/O operations.

Monadic I/O [PJW 1993], the newest and most popular model, hides the input and output streams and the I/O state, providing only an interface via I/O functions which represent actions. This saves the programmer from the pitfalls of managing the input and output explicitly, and the interleaving problems do not arise. In addition, I/O operations are performed when the corresponding I/O functions are evaluated, not when the resulting values are needed by some other part of the program; that is, I/O evaluation is strict, not lazy. Finally, the type signatures of I/O operations are much simpler.

Sequencing of I/O operations is explicit, via combinators which link I/O operations together:

```
return :: a -> IO a
(>>)   :: IO a -> IO b -> IO b
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

(The `(>>)` and `(>>=)` are infix operators, sometimes read "sequence" and "bind.") The `return` action performs no I/O but returns the value passed to it; `(>>)` performs the first action given it, followed by the second action, and returns the value returned by the second action; `(>>=)` performs two actions, passing the result of the first to the second, and returning the value of the second.

The I/O operations needed for the hangman example are `getChar` and `putStr`, which read a character and display a string, respectively.

```
guess :: [Char] -> IO [Char]
guess guesses =
  getChar                                    >>= \ newguess ->
  putStr (reveal target (newguess:guesses))  >>
  return newguess:guesses
```

Monadic I/O is described in greater detail in Chapter 2.

These five approaches present various cognitive obstacles.

Although four of these five interaction mechanisms implement input and output without disrupting the formal properties of functional languages, they do so at the cost of additional cognitive obstacles (the exception is the history approach, which adds minimal cognitive difficulty, but completely destroys lazy evaluation and referential transparency). All of the four have special type signatures for functions which include I/O: the types for continuations appear the most problematic, while the types for monads appear the least so. All four require that each I/O operation be handled explicitly by the user.

Streams, systems, and continuations have some unfortunate drawbacks not shared by monads. All three require that the user generate distinct names for different versions of the I/O state (the continuations approach avoids this for the function results passed to the continuation, but other names are sometimes necessary). Because of this, the user must be careful to return the correct version of the I/O state for use by other functions. Finally, all three are subject to the pitfalls described on page 6; the echoing and wait function workarounds to those pitfalls create more work for the user and present more opportunities for error.

1.1.2.2 Recent functional programming research has demonstrated the viability of functional languages for graphics and interaction.

Functional programming research since 1990 has produced a number of elegant approaches to programming graphics and interaction, with the goals of demonstrating that functional languages are not only the equals of imperative languages for constructing graphical user interfaces, but can surpass them in conciseness and power. As with the functional approaches to terminal-oriented I/O, these systems do not necessarily address the cognitive barriers to programming interactivity; the mechanisms for accessing this power have some of the same drawbacks, and in some cases introduce new obstacles as well.

(Note: the ensuing discussion is drawn from papers written about these systems and from examples provided by the system designers. In particular, the implementations of hangman given in this discussion should be viewed as pseudocode, as they may deviate slightly, or even significantly, from the actual system capabilities.)

Fudgets are interface components that communicate via streams.

Fudgets (functional widgets) [CH 1993] are best visualized as components with four pins, two above and two below.

The upper two pins are the application input and output message streams. The lower two pins are the implementation input and output event streams. Information flows through the fudget from right to left, for consistency with the syntax of function composition.

A fudget is of type `F a b` where `a` and `b` are the types of input and output messages, respectively. Fudgets are connected via combinators, which determine the connection of message streams and the screen layout of the fudgets. For example, the fudget side-by-side composition combinator

```
>==< :: F b c -> F a b -> F a c
```

can be used to place two text fudgets next to each other on the screen:

```
textF >==< textF
```

The output of the right fudget goes to the input of the left fudget; text typed in the right fudget is echoed in the left one.

Input from and output to the message streams is handled by continuation passing, using two functions

```
getSP :: (a -> SP a b) -> SP a b
```

```
putSP :: [b] -> SP a b -> SP a b
```

(the SP stands for "stream processor"). The function `getSP (\a -> sp)` is a stream processor which reads `a` from the input stream, then becomes the stream processor `sp` (i.e., it calls the continuation `sp`). Similarly, `putSP [b] sp` is a stream processor which places all the messages in the list `[b]` on the output stream, then becomes the stream processor `sp`. The `absF` operator is used to convert an ordinary (stream-processing) function into a fudget:

```
absF :: SP a b -> F a b
```

Such a fudget is referred to as an abstract fudget, because it does not have a visible representation.

In implementing hangman in Fudgets, using three fudgets seems to be the most natural approach, one each for the target, guesses, and progress strings. In the simplest implementation, the target string is unchangeable, and the target fudget is omitted.

```
progress_f :: textF
guesses_f :: textF
textF = F String String
```

The stream-processing function for the hangman example is:

```
reveal_SP :: SP String String
reveal_SP = reveal_SP' ""
  where reveal_SP' guesses = getSP computeProgress
        computeProgress newGuesses =
            putSP [reveal newGuesses target]
                  (reveal_SP newGuesses target)
```

and the hangman program is thus:

```
progress_f >==< (absF reveal_SP) >==< guesses_f
```

If we wish `target` to be manipulable as well as `guesses`, it is also necessary to create a composite fudget containing `target_f` (also of type `textF`) and `guesses_f`. The composite fudget consists of `target_f` and `guesses_f` "in parallel", with their input and output streams multiplexed together. The disjoint union fudget combinator `>+<` accomplishes this:

```
>+< :: F a1 b1 -> F a2 b2 -> F
(Either a1 a2) (Either b1 b2) Either a b = Left a | Right b
```

In addition, the stream-processing function must be more sophisticated.

```
reveal_SP:: SP (Either String String) String
reveal_SP = reveal_SP' ("","")
  where reveal_SP' (guesses,target) = getSP computeProgress
        computeProgress (Left newGuesses) =
          putSP [reveal newGuesses target]
                (reveal_SP newGuesses target)
        computeProgress (Right newTarget) =
          putSP [reveal guesses newTarget]
                (reveal_SP guesses newTarget)
```

and the hangman program is thus

```
progress_f >==< (absF reveal_SP) >==< (guesses_f >+< target_f)
```

Fudgets provide an easily-understood interface model. Using the combinators to join existing fudgets creates an interface relatively easily, since the fudget comes ready to "plug in." However, writing stream processors is still convoluted, and the input and output multiplexing can become quite a chore for large applications, although the multiplexing does guarantee deterministic concurrency. The need to create abstract fudgets separates computational and interface elements and creates extra work.

Concurrent Clean uses a hierarchical systems model.

Concurrent Clean [AP 1994] is a lazy functional language based on term graph rewriting. Clean is distinguished by several features. Graph rewriting is explicit – specifying the program also specifies the graph structure. Strictness annotations (the `!` operator, also available in Haskell) allow the programmer to make the program more efficient by specifying that a computation be eager rather than lazy. Various sorts of process annotations are available for concurrent applications. Finally, and most importantly for interaction, Concurrent Clean has an unusual system of environment passing and type and variable annotations which it uses to perform input and output.

In a lazy functional program, a single expression may recur at several different points in a program expression tree. Referential transparency states that the value of that expression must be the same at each evaluation. If this is not the case, the program result will depend upon the evaluation order, which violates the conditions of a declarative language.

Expressions referring to input and output, however, cannot have the same value each time they are evaluated. For a functional program to interact properly with input and output devices, an evaluation order must be enforced on the operations involving I/O. Function composition can enforce an evaluation order, but the problem remains of ensuring that each expression is evaluated exactly once.

Concurrent Clean addresses this problem via uniqueness (linear) types. Uniqueness annotations to a type (via `*`) or variable (via `.`) specify that when the expression is evaluated in the context of the annotation, the evaluation will occur exactly once. Since the term graph rewriting tree is an explicit part of Clean semantics, this is equivalent to saying that the annotated expression occurs uniquely in the term graph tree.

Concurrent Clean I/O occurs via environment passing, much like the system model discussed in the previous section. The Clean system provides a hierarchy of environments for files, interaction devices, and so forth, allowing functions to access exactly as much of the I/O state as they need, without having to unpack the I/O world parameter and repack the I/O world return value.

In this approximation to the hangman example, `GuessState` is a process with guesses and progress as its local state. The `ps:{pLocal=local, pIOState=io}` matches `local` and `io` to the `pLocal` and `PIOState` fields of `ps`, respectively, and the `{ps & pLocal = x}` syntax specifies a value consisting of the value of `ps` with the `pLocal` field set to `x`.

```
:: GuessState share :== PState Local share
:: Local = { guesses :: Text,
             progress :: Text }
guesses :: KeyState *(GuessState s) -> *(GuessState s)
guesses (c,_,_) ps:{pLocal=local, pIOState=io} =
  {ps & pLocal = {local & guesses = guesses1,
                          progress = progress1},
        pIOState = DrawInActiveWindow drawingFunctions io}
where drawingFunctions = DrawString progress1,
      progress1 = reveal target guesses1,
      guesses1 = addCharacter c local.guesses
```

The Clean system differs significantly not only from other interactive functional programming

toolkits, but (as noted in [NR 1994]) from other functional programming languages. Clean was originally designed to be an intermediate language between other functional languages, and the strictness and uniqueness annotations and the lack of syntactic sugar are consistent with this design. Unfortunately, the interface tools are built-in and inflexible, and there are no combinators for composing objects. Finally, uniqueness types are somewhat tricky to use, occasionally requiring more code than seems strictly necessary, and requiring some of the same name-generation tricks common to the streams, systems, and continuations approaches.

Gadget Gofer components pass continuations along multiple input and output channels.

Gadget Gofer [NR 1995] provides components similar to fudgets, but each component can have an arbitrary number of input and output channels (wires). Each gadget is a process (of type `Process s`, where `s` is the state of the process), which communicates asynchronously through wires (of type `Wire a`, where `a` is the message type). Gadgets send messages to other gadgets with the tx primitive:

```
tx :: Out m -> m -> Process s -> Process s
```

which takes an output pin, a message, and a process, and returns a process (a form of continuation passing). Gadgets receive messages from their inputs via a combination of the `rx` and `from` primitives:

```
rx :: [Guarded s] -> Process s
from :: In m -> (m -> Process s) -> Guarded s
```

The `from` primitive takes an input pin and a continuation for that pin, and returns a guarded continuation. The `rx` primitive takes a list of these guarded continuations and waits for a message to arrive on one of the input pins, then chooses the guarded continuation which matches that pin.

Gadgets also communicate with the display and input devices (referred to as the screen manager, or `SM`) via wires, but these wires are encapsulated within functions and are not directly available. For instance, a Gadget would use the `txSM` function to send a message to the screen manager instead of to another Gadget:

```
txSM :: m -> Process s -> Process s
```

Implicit, hidden wires also provide access to the operating system and to the layout managers. Gadget layout is specified by the programmer using layout combinators.

To program hangman, we need three other primitive functions:

```
wire :: (Wire a -> Process s) -> Process s
ip :: Wire a -> In a
op :: Wire a -> Out a
```

The `wire` primitive creates a wire, which it passes to its continuation argument. The functions `ip` and `op` simply select the input and output pins of a wire, respectively.

We can program the hangman example in Gadget Gofer with three gadgets. Two (`guesses` and `target`) are what we assume a stock text Gadget would be; the third (`progress`) is a special text Gadget with two input wires. Two wires, one each from `target` and `guesses` to `progress`, complete the picture.

```
hangman :: Gadget
hangman =
  wire $ \ gp ->
  wire $ \ tp ->
```

```
let t = text "" (op tp)
    g = text "" (op gp)
    p = rx [
             from tp $ \t -> txSM (reveal t g) $ p
             from gp $ \g -> txSM (reveal t g) $ p
           ]
```

Gadget Gofer is similar to Fudgets, and addresses two of the latter's shortcomings: the need for abstract fudgets (which separate computation from interaction) and the need to multiplex messages through a single channel between fudgets, although determinism is lost. The dynamic creation of wires and components is clean and flexible. The `Guarded s` notation using `rx` and `from` handles the nondeterminism in the message passing. The continuation passing syntax is still a bit cluttered, and (as the authors note) the construction and wiring of components still seems a bit low-level.

Haggis uses monads for widgets and I/O devices.

Haggis [FPJ 1995] is an interface framework which extends the monadic I/O concept to include widgets, which Haggis refers to as "components." All user interface components are device abstractions, much as files, sockets, and the console are abstracted into file handles in UNIX programming. Events relevant to a virtual device (the keyboard, for example) are forwarded to that device by the Haggis implementation. Components can be combined by composition functions; for instance, a composition function for creating a radio group from a collection of buttons has the following (simplified) type signature:

```
radio :: [ Button Int] -> IO (Button Int)
```

It takes a list of buttons, puts them in a layout, and forks a process which implements the exclusive choice that defines radio buttons. These composition functions thus handle communication between components, such as the " deactivate" message from the newly selected radio button to the previously selected one.

To implement hangman in Haggis, we use the `TextField` component to store the `target` and `guesses` strings. To create a `TextField`, we use the `textField` function:

```
textField :: String -> Component (TextField,DisplayHandle)
```

Since the `progress` string is not editable, we can use a `Label` for it:

```
label :: String -> Component (Label,DisplayHandle)
```

Before we can create the components, we need to create a `DisplayContext` (essentially, a window) to display them in. To do so, we use `mkDC`

```
mkDC :: [String] -> IO DC
```

which creates a `DisplayContext` given a style specification (a list of `Strings`).

With the `DisplayContext`, we can create the three components:

```
dc <- mkDC ["*title: Hangman"]
(target,targetDH) <- textField "eggplant" dc
(guesses,guessesDH) <- textField "" dc
(progress,progressDH) <- label "--------" dc
```

(we are using the `do` syntax for monadic I/O described in Chapter 2). The `DisplayHandles` (variables ending in `DH`) are used to specify layout (and, in more sophisticated programs, to capture events over a component). The `vbox` function

```
vbox :: [DisplayHandle] -> DisplayHandle
```

combines a list of `DisplayHandles` vertically into a single `DisplayHandle`, and `realiseDH` creates the column of components on the display:

```
realiseDH (vbox [progressDH,guessesDH,targetDH])
```

We have created the components we need; now we specify the behavior by using the `forkIO` function to create a process that watches the `guesses` component and updates the `progress` component:

```
forkIO (trackGuesses guesses progress)
where
  trackGuesses target guesses progress =
    newGuesses <- waitChangeString guesses
    theTarget <- getString target
    setLabel progress (reveal theTarget newGuesses)
    trackGuesses target guesses progress
```

The event handler blocks (`waitChangeString`) until the string held by `guesses` changes, then it extracts the target string and sets `progress` to the result of the `reveal` function. Hiding the target string would involve using

```
mkTextField :: (String->Picture) -> String
            -> Component (TextField,DisplayHandle)
```

whose first argument is a function which specifies how the string is displayed.

Other environments address issues at the window system and application levels.

EXene [GR 1991] is a multi-threaded window system toolkit built directly upon the X Window System protocol. It is implemented in Concurrent ML and uses threads and channels to encapsulate state. Concurrency and delegation replace object-orientation as the approach to modularity and separation of concerns. EXene demonstrates the applicability of functional programming to the low-level implementation issues of a windowing system.

BriX [Serrarens 1995] is a concurrent window system implemented in Haskell which uses monads for widget communication in a manner similar to that of Haggis. A distinguishing feature of BriX is that it is both concurrent and deterministic, like Fudgets and Clean, and unlike eXene, Gadgets, and Haggis.

1.1.2.3 Interactive functional programming environments reduce some cognitive obstacles, but introduce others.

The Fudgets, Concurrent Clean, Gadget Gofer, and Haggis systems each abstract away some of the details of the traditional I/O models upon which they are built. The Fudgets system uses streams to implement message passing, but the user need not know that, although she or he might wonder why the access functions are named `getSP` and `putSP`, and the I/O multiplexing problems persist. Similarly, a full understanding of continuations is not necessary to use the Fudgets or Gadget Gofer systems; in particular, the difficult types are simplified. Even so, the syntax for receiving and passing messages is clumsy and ill-motivated for the user who does not understand continuations.

In Concurrent Clean, understanding and using uniqueness types and the I/O state retains from the systems model the problem of generating different names for different versions of the I/O state, although the hierarchical I/O state reduces the user's concern with multiplexing.

Haggis, by drawing exclusively upon monadic I/O, avoids all of the difficulties associated with streams, systems, and continuations, and in fact with message passing as well. In its place,

Haggis requires the user to perform explicit event dispatching to enable interface elements to communicate, but it is far from clear whether this approach is markedly easier. Discussion of the monadic I/O system in Haggis is provided in chapter 2.

1.1.3 The strengths of functional programming languages support diverse uses.

1.1.3.1 A wide range of real-world systems are implemented in pure functional languages.

Philip Wadler maintains a list of real-world applications of functional programming, including pure functional programming [Wadler 1998a, Wadler 1998c]. Artificial intelligence systems are well represented: theorem provers, expert system shells, natural language understanding, and speech recognition. Functional programming also appears well suited for systems applications, including a concurrent client-server architecture for telecommunications, a "smart-card" operating system, and, of course, compilers, parsers, and syntax tools. Other real-world uses of functional programming include consistency maintenance for database transactions, query languages for heterogeneous distributed databases, declarative animation, computing nucleic acid shapes, a calculus for digital circuits, airline crew scheduling, and a protocol library for distributed applications (including a web server).

1.1.3.2 Functional languages are particularly popular for certain classes of problems.

Common element recur in these real-world applications, and suggest a list of problem characteristics to which functional languages are well suited. Tasks involving artificial intelligence techniques, such as proof systems, search, or symbolic manipulation, make use of the declarative functional style, the power of lazy evaluation, and the ease with which new datatypes can be created in functional languages.

Compilers, client-server architectures, and database query languages point to the ease of specifying transformations in functional languages, and all are well-suited to demand-based evaluation. In fact, the backward chaining used in expert systems neatly dovetails with lazy evaluation [Wadler 1998a].

1.1.3.3 These applications show the value of functional language tools.

The functional programming tools described in section 1.1.1.1 find their uses in these applications. Higher-order functions are used in theorem provers to manage subgoals and justifications; for instance, in LCF [Wadler 1998a], a proof tactic is a function from a goal to a list of subgoals and a justification, while a justification is a function from proofs of subgoals to proofs of goals. The concurrent functional language Erlang encapsulates common client-server processing into functions that are the analogs of `map` and `fold` for list processing. Tail-calls and higher-order functions maintain the state transitions.

Lazy evaluation plays a crucial role in minimizing memory usage in Natural Expert's database access supports concurrency in CPL/Kleisi's query processing, and in the backward chaining mentioned above. Finally, the type system helps ensure the soundness of theorem provers and supports hierarchical database queries.

1.1.3.4 Functional tools also support experimentation and program development.

Another application of the flexibility of functional tools occurs in program construction. Lazy evaluation supports functions that solve large or unbounded problems, because other functions can be used to control how much of the solution is computed. Changing the size of a

problem thus involves fewer modifications to a program. Higher-order functions, by separating different program computations, allow a much greater interchangeability of program parts; "swapping in" a different parser or filter or transformation is greatly simplified.

Although the strong static type-checking of functional languages appears at first glance to be an impediment to quick program construction (which is aided by the declarative, high-level functional approach), it shortens debugging considerably. Static type-checking also offers the strong advantage of ensuring that the parts of a program fit together properly. Type inference further simplifies the program construction process by eliminating the need to write type signatures — and the user can always write them as a more careful correctness check.

1.2 Spreadsheets

1.2.1 Spreadsheets are expression-based programming environments.

1.2.1.1 Spreadsheet computations consist of expressions.

Spreadsheets consist primarily of a grid of cells; each cell holds a value (such as a number or a character string) or a formula. Values are displayed in a variety of formats, and can be directly edited by the user. Formulae are expressions consisting of constants (numeric, string, ...), arithmetic operators, built-in functions, and cell references. A formula cell displays the current value of the expression it contains.

All of the computation in the spreadsheet occurs via the formulae. When a cell value changes, any formula that references that cell is recomputed and the new value is displayed (and other formula cells that reference the first one are updated, and so on). Similarly, whenever a formula is edited, the newly edited formula is re-evaluated, followed by any cells whose formulae reference it.

1.2.1.2 Spreadsheet formulae serve as one-way constraints to maintain consistency.

Because all relevant formulae are re-evaluated whenever any cell is changed, the spreadsheet contents remain consistent. The formulae are essentially one-way constraints: the values to which they evaluate cannot be edited, so the issue of propagating values backwards is moot.

1.2.1.3 The spreadsheet provides spreadsheet programs with an interface for free.

The spreadsheet environment provides an automatic interface for its computations. Data storage, editing, and display are all bundled together into the spreadsheet cell. Formulae cannot all be visible simultaneously under normal circumstances, but can be viewed and edited on a cell-by-cell basis. The result is that spreadsheet computations need no facilities for reading, editing, or printing data. The interface provided by the spreadsheet (while not always ideal) is always available for any computation.

1.2.2 Uses of spreadsheets center around equation-based models and experimentation.

Spreadsheets have been used for a wide range of applications, many of them educational. ([Neuwirth 1998] lists a substantial selection.) Some of the more prominent areas of use are applied mathematics (Fourier sound synthesis, root-finding, vector addition, series convergence), physical science (molecular weight, ideal gas law, trajectory calculations, chemical equilibrium problems, chemistry simulations), finance and financial modeling (Social Security retirement payments, option pricing), statistics (fitting curves to data), and computer science (state machines, simulation of arithmetic units, algorithm animation, circuits), along with numerous other

miscellaneous examples (combinatorial identities, sums of collections of coins, computing parliament seats from votes, equation solving, ...).

These uses point to general characteristics. Spreadsheets are a powerful problem-solving environment: they can be used to solve a wide range of problems. Spreadsheets are well suited for experimentation and exploration. Changes can be made easily, and new results are immediate. Additionally, the visible, editable, tabular data representation is appropriate for many different tasks.

1.2.3 Spreadsheets increase the accessibility of programming and programming concepts.

1.2.3.1 The spreadsheet has several features which reduce the cognitive cost of writing programs.

Spreadsheets present the programmer with a familar, visible, editable data representation [LO 1987]. No special operations are required to view or modify the data. In traditional programming, data lives in an "inner world" of variables and computation. Modifying the data involves extracting it from this inner world, then pumping the modified data back in. None of this is necessary in the spreadsheet: the data can be modified incrementally.

Spreadsheets typically have a large number of aggregate operations available; the programmer need not construct these from more primitive operations. Since these aggregate operations are built in, there is no extra work such as importing or linking with libraries.

Formulae in spreadsheets specify (one-way) constraints, which are maintained automatically. In traditional programming, consistency is up to the programmer, and typically cannot be specified in so simple a fashion. Automatic consistency maintenance, in conjunction with the visible data representation, provides immediate feedback to the programmer; there is no need to reinvoke the program on the new data. Automatic consistency maintenance also obviates any need for control flow. The absence of a control model and variables frees the programmer from the need to keep track of state changes in the program over time.

1.2.3.2 Spreadsheets provide tools that change the user's interaction with equations and programming.

Neuwirth [1995a, 1995b] has investigated the cognitive implications of the spreadsheet model with a slightly different emphasis. He notes that spreadsheets provide a spatial organization absent in traditional programming, and that clicking on cells to specify formula references is a quasi-gestural, rather than symbolic, means of specifying computational relationships. The use of pointing (clicking) also removes the need for variables to have names.

Neuwirth points out that chains of formulae can show recursion in a highly visual way: for instance, the function 2^n can be expressed in a spreadsheet by a column of identical formula cells. Each formula multiplies the value in the cell above by 2. The base case is a value cell (with a value of 1) at the top of the column. He also demonstrates that spreadsheets support easily converting constants into variables, and numerical representations into graphical ones.

1.3 Other Interactive Environments and Programming Systems

1.3.1 Research in human-computer interaction and programming environments offers insights into cognitive barriers and implementation issues.

A number of other interactive environments address the goal of reducing the cognitive cost of programming interactivity.

User interface toolkits such as Garnet [Myers 1989] and Mickey [Olsen 1989] have explored the construction of combinable interaction primitives. Mickey maps the Macintosh interface onto declarative Pascal, while Garnet provides via Common Lisp a set of interaction abstractions

to the X Window System. Other toolkits such as Tcl/Tk [Ousterhout 1994] and InterViews [LVC 1989] have taken imperative approaches.

The Model-View-Controller model of Smalltalk [GR 1983], the Abstraction-Link-View paradigm of the Rendezvous project [Hill 1992], and the Surface Interaction model of Presenter [Took 1990] are approaches to the problem of connecting display images and interaction events to program entities.

Many newer systems, such as (standalone) Visual Basic [Halverson 1998] and Java [AG 1997], provide component libraries of interface objects. These objects can be connected to other interface objects and to computational objects within a single object-oriented design.

Constraint systems such as DeltaBlue [SMFB 1993, FMB 1990] use multi-way constraints as the primary means of problem specification, and constraint solving as the computational engine. ThingLab [Borning 1981] is an ancestor of DeltaBlue's approach to interface objects.

The Acme [Pike 1994] and Oberon [Wirth 1989] environments explore the integration of interface and programming environment. Squeak [CP 1985] provides a programming-language approach to mouse interaction.

Finally, many environments have integrated programming constructs and graphical objects, including Agentsheets [RS 1995], Boxer [dSA 1986], Cartoonist [Hübscher 1995], ChemTrains [Bell 1992], and LiveWorld [Travers 1994]. Other environments, such as DataSheets [Wilde 1994], SchemePaint [Eisenberg 1995], and VIPR [CDZ 1995] have explored other combinations of programming and direct manipulation.

1.3.2 In NoPumpG, spreadsheet cells control interactive graphics and vice versa.

NoPumpG II [WL 1990] integrates a spreadsheet environment with interactive graphics. In addition to traditional spreadsheet cells, NoPumpG provides "control cells" for graphical primitives. For instance, a line primitive would have an x-coordinate cell and a y-coordinate cell for each end. Moving a primitive changes its control cells, and editing the control cells modifies the primitive. Cells can be also linked to other cells in this bidirectional fashion, and thus each cell can contain a value and a formula.

Cells are placed freely, allowing them to be placed near the graphics they control, but also sacrificing the structure provided by the traditional spreadsheet grid (including, unfortunately, the range-filling operations typically provided). Although cells cannot, therefore, be referred to by location, they can be given mnemonic names.

1.3.3 FunSheet is a spreadsheet based on a functional language.

FunSheet [dHRvE 1995] is a spreadsheet (implemented in Concurrent Clean) in which the formula language is a lazy (but weakly typed) pure functional language. FunSheet's formula language provides higher-order functions. In addition, column references are also functions that can participate in formula expressions; of course, the most common use of a column function is to apply it to an integer to reference a single cell.

FunSheet also provides symbolic evaluation, which can be used to check two different versions of a computation in different parts of a sheet, or to compute a simplified version of an equation.

1.3.4 Forms/3 uses spreadsheet-like cells for visual declarative programming.

The visual declarative language Forms/3 [BA 1994] uses hierarchically-grouped spreadsheet-like cells for specifiying visual abstract data types (VADTs). Cells refer to other cells via formulae. Scoping is accomplished visually, and formulae can be written by direct manipulation. When events occur over special "event receptor" cells, they are updated, triggering formulae which refer to those cells. In addition, a "temporal vector" structure provides access to previous

cell values and supports programming complex interactions (for example, gestures). Forms/3 is weakly typed, but event receptors behave lazily, preserving referential transparency.

The focus of Forms/3 is on purely visual programming of graphical and behavioral abstraction. Forms/3 does not support circular formula references, and it is unclear whether interactions and computations can be bidirectionally connected as in NoPumpG II.

1.4 Thesis

Functional programming languages and spreadsheets offer enhancements to the programming process that are different, yet complementary. The tools provided by functional languages show the power available in the world of pure (stateless) expressions, but functional approaches to interaction are still problematic. Spreadsheets, in contrast, have relatively weak, but still expression-based, formula languages; the great strength of spreadsheets is in program construction and interaction.

Functional programming languages and spreadsheets also share important benefits. Both environments support quick construction of small programs.

1.4.1 The strengths of functional programming and spreadsheets can be combined in a powerful, flexible system.

The goal of this thesis is to address the weaknesses of functional programming languages (cumbersome treatment of interaction) and the weaknesses of spreadsheets (computational limitations and lack of interface flexibility) by combining the expressive computational power of a functional language with the interaction-for-free power of the spreadsheet environment, and expanding the spreadsheet model of interaction to include graphical interaction and customization.

1.4.2 A synthesis of functional programming and spreadsheets would support graphical, interactive exploration of complicated models.

A spreadsheet-like framework built on a functional language is an appropriate medium for the quick construction, development, and exploration of sizeable models. The expressive power and concise syntax provided by the functional language support complex computations, while lazy evaluation allows varying portions of potentially unbounded computations to be used. The modularity provided by higher-order functions and lazy evaluation, coupled with the interactive flexibility of spreadsheet cells, supports easy modification and experimentation during construction and development. The interactive power of the spreadsheet supports adding interactive graphics to functional computations.

Given these strengths, a functional programming-spreadsheet synthesis seems most suited to constructing and exploring mathematical or physical models, interactive search problems (such as game-playing), signal processing and other problems expressible as data transformations, and problems involving iterates or recursive specifications (such as the Newton-Raphson example on page 2).

1.4.3 The programming walkthrough is well suited for evaluating a wide range of programming environments.

1.4.3.1 The programming walkthrough is useful for evaluation and for informing design.

Our tool for examination is the programming walkthrough [BCLRWWZ 1994], a structured, process-oriented method for assessing the ease of constructing programs. The programming walkthrough serves two purposes as an evaluative tool: to compare the ease of program

construction across systems, and to inform the design process of Esquisse, a programming environment we have developed which synthesizes and expands upon the functional and spreadsheet approaches.

1.4.3.2 The examples used in the walkthroughs are interactive mathematical models.

Our approach centers around two example programs: line-fitting and temperature conversion.

In the line-fitting example, the user wishes to explore fitting a line to a set of data. Some of the data points should be treated as outliers; the user can experiment with fitting the line to various subsets of the data. The problem statement for the line-fitting example is:

> Given a list of (x, y) pairs, plot the pairs on the screen as points. Each point should be either blue or yellow; blue points are inliers, while yellow points are outliers. Plot a line that is the least-squares linear fit to the set of inliers (blue points). A mouse click on a point should toggle its status between inlier (blue) and outlier (yellow), or vice versa. The fitted line should always reflect the current set of inliers.

The temperature conversion example features two linked thermometers, with their temperatures displayed textually below them. One thermometer shows the temperature in degrees Fahrenheit, the other shows it in degrees Celsius. Dragging either thermometer or editing either textual display changes the other thermometer(s) and textual display(s) appropriately. The temperature conversion example problem statement is:

> Display two draggable vertical bars on the screen, one representing the Fahrenheit temperature, the other the same temperature in Celsius. Also, display with each bar its temperature in numeric (textual) form. Editing either number should change both bars and the other number, and dragging either bar should change the other bar and both numbers. At all times, the numeric temperatures should agree with the heights of their corresponding bars, and the Fahrenheit number-bar combination and Celsius number-bar combination should be in the correct numeric relationship.

These examples (figure 1.1) capture many of the strengths of the functional-spreadsheet synthesis. Both examples are highly interactive and exploration-based; one is statistical, the other physical. Both of them add interactivity to a computation, yielding a flexible, graphical, interactive model.

1.4.3.3 The walkthroughs explore a functional system and a spreadsheet system.

We strive for a useful synthesis of the functional programming and spreadsheet approaches. To this end, we explore in depth the cognitive implications of programming interaction in each of these systems. In the case of functional programming, we focus on Haskell [HPJW+ 1992, HPF 1997], a widely used functional language, and in particular on its monadic I/O system [PJW 1993], which rapidly became the I/O standard after its introduction. Since we are interested in graphical interaction, we examine Haskell and monadic I/O in the context of Haggis [FPJ 1995], the only functional toolkit for graphical interaction that uses monadic I/O.

In the case of spreadsheets, we study Microsoft Excel, a popular and extensive spreadsheet application. Access to graphical interaction in Excel is provided by Microsoft Excel with Visual Basic for Applications; hence we examine the two together.

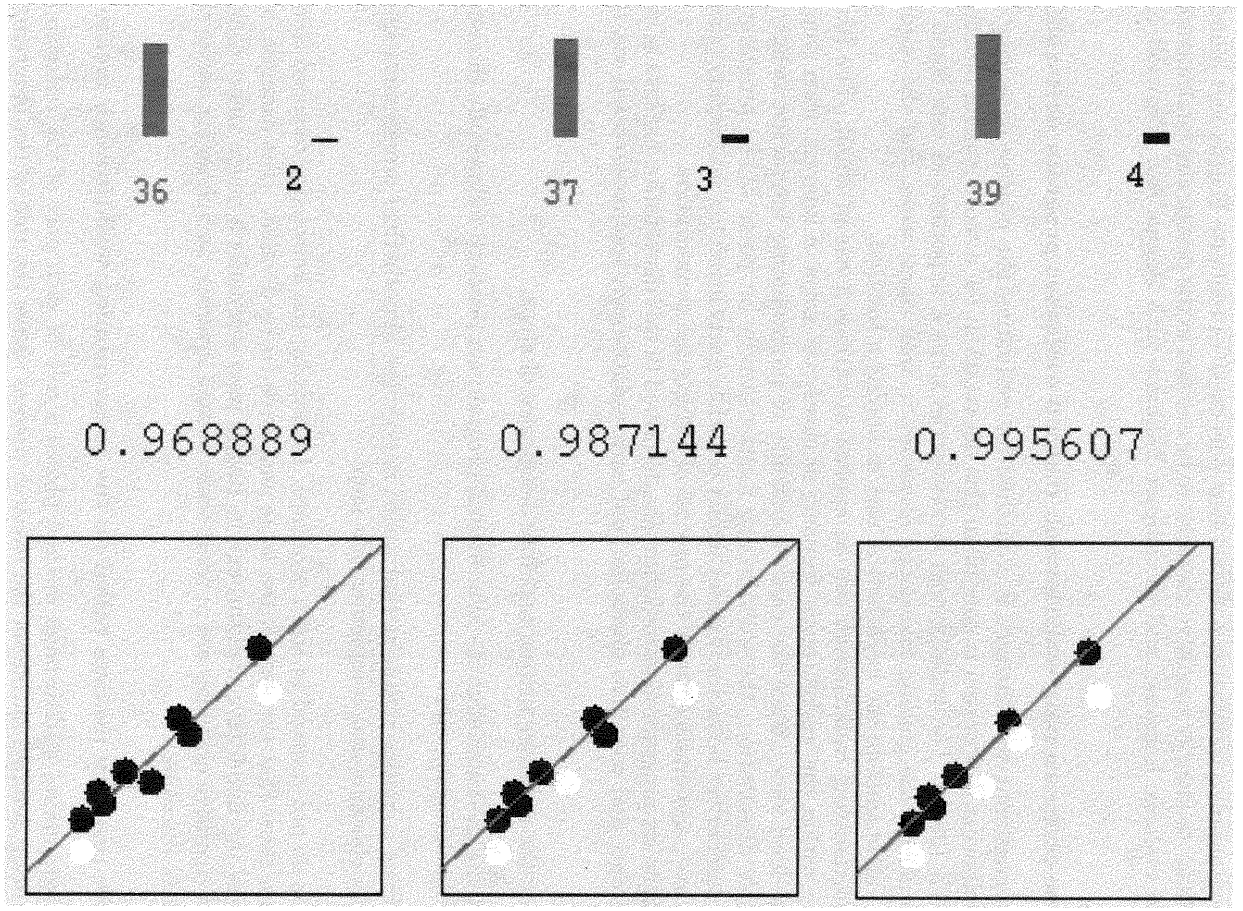


Figure 1.1: The temperature conversion and line-fitting examples.

1.5 Research Process

The dissertation research process consisted of three major activities. The first activity was a comparative study of two existing systems, Haggis and Microsoft Excel with Visual Basic, using the programming walkthrough method. The second activity was the design and implementation of Esquisse. The third activity was an evaluation of Esquisse, alone and in comparison with Haggis and Microsoft Excel with Visual Basic.

The programming walkthrough served four purposes: to evaluate Haggis and Microsoft Excel Visual Basic, to identify cognitive obstacles to be avoided in the design and implementation of Esquisse, to evaluate Esquisse, and to compare Esquisse with Haggis and Microsoft Excel Visual Basic.

1.5.1 The programming walkthrough method evaluates program writeability.

The programming walkthrough is a method for evaluating the process of writing programs. (The following description is specific to this dissertation; for a more general treatment, see [BCLRWWZ 1994].) The walkthrough describes the steps a hypothetical user could take to solve the problem, the guiding knowledge necessary to take those steps, and the cognitive obstacles that might impede the hypothetical user in finding a solution.

The programming walkthrough can be applied at various levels of detail. The level of detail can vary within the walkthrough, focusing in closely on areas of interest, then stepping back for a high-level view. Since the ease or difficulty of programming interactive behavior is the focus of this dissertation, the walkthrough focuses heavily on parts of the programming task involving interactive behavior, focuses lightly on other parts involving interaction (such as creating windows), and omits entirely the process of writing the computation.

Each of the six programming walkthroughs described in this dissertation consists of steps, guiding knowledge, and cognitive obstacles, and each walkthrough was constructed via three activities:

- dividing the program-writing task into a sequence of steps,
- identifying the knowledge the hypothetical programmer needs to successfully choose and complete that step, and
- identifying cognitive obstacles that might impede the hypothetical programmer in successfully choosing and completing that step.

These activities do not proceed in strict sequence. Identifying the knowledge required to complete a step can show that the step is too large and needs to be divided. (For example, steps 21 through 49 of the line-fitting walkthrough in Haggis were originally a single step.) Guiding knowledge and cognitive obstacles are in some sense mutual inverses, and changing one of them frequently affects the other.

1.5.2 The research process relied on the programming walkthrough method.

The programming walkthrough method played a role in each part of the research process. The following detailed outline of the research process shows the role of the programming walkthrough in greater detail.

(1) Comparative Study (chapters 2 and 3)

  (a) implementing the example programs in each of Haggis and Excel with Visual Basic
  (b) writing walkthroughs for each example in each of Haggis and Excel with Visual Basic

(c) using the walkthroughs to identify cognitive obstacles to programming the two examples in Haggis and in Excel with Visual Basic

(2) Design and Implementation (chapter 4)

(a) designing Esquisse with an awareness of these obstacles

(b) implementing Esquisse

(3) Evaluation (chapter 4)

(a) implementing the example programs in Esquisse

(b) writing walkthroughs for each example in Esquisse

(c) using the walkthroughs to identify cognitive obstacles to programming the examples in Esquisse

(d) using the cognitive obstacles found for each system to compare Esquisse to Haggis and Microsoft Excel Visual Basic

1.6 Contributions

This dissertation contributes to the study and design of spreadsheets, of functional programming environments, and (more generally) to the study and design of programming environments for interactive exploration.

Contributions to the study of spreadsheets include:

- increasing the generality of cell values and formulae
- opening the interactive capabilities of spreadsheet cells to customization
- allowing cells to contain both values and multiple (possibly circular) formulae
- describing the challenges of integrating the spreadsheet model with strong typing
- addressing the programming-language issues surrounding cell editing and program construction

Contributions to the study of functional programming environments include:

- providing (customizable) interactivity "for free" with specification of a computation
- providing a richer interaction between programmer and program
- supporting graphical representations for program entities
- providing consistency maintenance of interface with data, and between data elements
- increasing the flexibility of the environment for development and testing

This dissertation addresses the effectiveness of a functional programming-spreadsheet synthesis for quick construction and exploration of interactive and graphical models, including models with multiple equations or parts, complex computations, and unbounded data structures. It also addresses the compatibility of the spreadsheet model and the functional model, particularly in contrast to a spreadsheet-imperative combination.

Chapter 2

Monadic I/O in Haskell and Haggis

Haskell provides access to the stateful world of input and output via the I/O monad. Haggis builds on the Haskell approach by using the I/O monad to provide access to a wide array of stateful interface components. As noted in Chapter 1, Haggis and Haskell use the I/O monad because it provides an efficient, straightforward interface to input and output without compromising the equational reasoning that is central to Haskell.

Section 2.1 introduces the monadic I/O system in some detail, and is based in part on the Gentle Introduction to Haskell, version 1.4, pages 30–33 [HPJW+ 1992]. Section 2.2 (page 31) presents the cognitive obstacles associated with monadic I/O in Haggis, as discovered by the programming walkthrough method.

2.1 Interaction in Haggis is based on Haskell's monadic I/O system.

Since interface components are stateful, Haggis components are built on top of the I/O monad, and are accessed and updated via the monadic combinators `>>`, `>>=`, and `return`; or, equivalently, in `do` contexts.

2.1.1 Monadic I/O creates an "imperative sublanguage" within the functional world of Haskell.

Haskell is a non-strict, or lazy, language. Expressions are not evaluated unless and until they are needed by another part of the computation. For this reason, lazy languages are sometimes referred to as "call-by-need."

Why does a program in a non-strict language do anything at all? The "need" in "call-by-need" originates with the user, who wants to see output. In the terminal I/O model of programming, program output consists of one or more expressions printed on the terminal. In interface programming, such as that of Haggis, program output also includes (in fact largely consists of) stateful changes to the display.

The need to produce output originates with `main`, which denotes the entire behavior of the program; running the program consists of evaluating `main`. But `main` includes stateful effects on the outside world, so it cannot be an ordinary stateless expression. Consider the program:

```
main = putStr "Hello!"
```

The user typed `putStr "Hello!"` to perform an action, not compute a value. Yet a pure functional language like Haskell can only compute values, not perform actions. The solution is that Haskell programs define actions, but do not invoke them. The definition of actions takes place in the orderly world of Haskell expressions. The invocation of actions takes place outside, in the world of program execution. The program shown above defines the action `main` to be the action `putStr "Hello!"`; when the program is run, the action is executed, and `Hello!` appears on the standard output.

To construct actions, Haskell provides atomic actions as system primitives, and the I/O monad provides three operations (`>>`, `>>=`, and `return`) that build composite actions. Composite actions are constructed in a manner similar to the sequential ordering of statements in imperative programming languages.

In imperative languages, sequencing is trivial; in fact, it is unavoidable. In a lazy functional language, in which the order of evaluation is explicitly not specified, sequencing must be explicitly specified. The monadic combinators do just that.

2.1.1.1 Monadic I/O encapsulates state changes inside an abstract data type.

In the systems model described in Chapter 1, the I/O state was a value explicitly passed from one I/O-performing function to another. Monadic I/O encapsulates the I/O state behind the monadic combinators. (An interesting consequence of encapsulating the I/O state is that it turns out not to be needed at all.)

For purposes of explanation, assume (as in Chapter 1) that the state consists of two (lazy) lists of characters, and define the type `IO a` to be that of functions from states to value-state pairs (compare with the systems example in Chapter 1).

```
type State = ([Char],[Char])
type IO a = State -> (a,State)
```

One way to read the type `IO a` is "an action that, when invoked, returns a value of type `a`." For example, in Chapter 1, the type of the `guess` function in the systems model was:

```
guess :: String -> State -> (String,State)
```

and in the monadic model it was:

```
guess :: String -> IO String
```

We can now show that these are equivalent: by substituting `String` for `a` in the definition of `IO a`, we see that `IO String` is equivalent to `State -> (String,State)`, and thus the two definitions of `guess` are identical. We can read the type of guess as "a function that maps a `String` to an action returning a `String`."

Consider a simpler example, that of reading a character from the keyboard. In the pair-of-lists model of the state, reading a character is the equivalent of returning the head of the input list of characters (the character read) and returning a new state consisiting of the tail of the original input list and the original (unmodified) output.

```
getChar :: IO Char
getChar (inputList,outputList) =
  (head inputList,(tail inputList, outputList))
```

In Haskell, `getChar` function would be written with pattern-matching, rather than `head` and `tail`:

```
getChar (char:inputList,outputList) =
  (char,(inputList,outputList))
```

Suppose the user presses the `z` key. Applying `getChar` to the current state yields:

```
getChar ('z':restOfInput,outputSoFar) =          -- pseudocode
  ('z', (restOfInput,outputSoFar))
```

Now consider an output action, the equivalent of `putStr "Hello!"` in the previous section. Displaying "Hello!" on the output is equivalent to appending it to the output list of characters. The input list is unaffected. The catch is that the `putStr` action does not have a useful value to return. The convention in the I/O monad is to use the value `()` as an "empty value." (The value `()` is the only value of the trivial type `()` (also called "unit") in Haskell.)

```
prompt :: IO ()
prompt (inputList,outputList) =
  ((),(inputList,outputList ++ "Press a lower-case letter key"))
```

(The ++ is Haskell's concatenation operator.) Now suppose that we wanted to print the prompt, then read the character. The I/O monad defines >>, which sequences two actions into a composite action. The composite action returns the value of the second action; the value of the first action is discarded (the >> operator thus resembles `progn` in Lisp or `begin` in Scheme). The type of >> is

```
(>>) :: IO a -> IO b -> IO b
```

and for our definition of `State`, we can treat >> as if it were defined

```
(>>) f g = g.snd.f
```

(the Haskell function `snd` selects the second element of a pair), or, equivalently (but less concisely) as

```
(>>) f g state =
  let
    (value,state') = f state
  in
    g state'
```

(The parentheses are necessary because >> is an infix operator. The . denotes function composition in Haskell.) Note that `value`, which was returned from the first action `f`, is unused. Given >>, we can see that

```
(prompt >> getChar) :: IO Char          -- more pseudocode
(prompt >> getChar) ('z':restOfInput,outputSoFar) =
  ('z', (restOfInput,
         outputSoFar ++ "Press a lower-case letter key"))
```

Now suppose that we next wish to print out the character in upper case. We can define an action:

```
printUpper :: Char -> IO ()
printUpper char (inputList,outputList) =
  ((), (inputList, outputList ++ [toUpper c]))
```

In Haskell, `printUpper` would be written:

```
printUpper c = putStr [toUpper c]
```

Now we need to add `printUpper` to our sequence of actions. Using >> will not work, because `printUpper` needs the value returned by the `getChar` action, and >> will throw it away. Instead, we use >>=, which has type

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

We can think of it as having the definition

```
(>>=) g h = (uncurry h).g
```

or equivalently

```
(>>=) g h state =
  let
    (value,state') = g state
  in
    h value state'
```

Now we can put all three actions together:

```
(prompt >> getChar) >>= printUpper) :: IO () -- still more pseudocode
((prompt >> getChar) >>= printUpper) ('z':restOfInput,outputSoFar) =
  ((), (restOfInput,
        outputSoFar ++ "Press a lower-case letter key" ++ "Z"))
```

Our example shows how the combinators >> and >>= sequence actions together. Our example also shows how messy manipulating the state explicitly can be. The I/O monad hides all of these state manipulations in the type `IO a`. Our example, written in terms of the I/O monad, is much simpler:

```
putStr "Press a lower-case letter key"
  >> getChar >>= \ c -> putStr [toUpper c]
```

or, in the `do` notation (see 2.1.2):

```
do
  putStr "Press a lower-case letter key"
  c <- getChar
  putStr [toUpper c]
```

Notice the complete absence of any references whatsoever to the `State`. The absence of `State` has a striking implication: we need not represent the state at all, because the combinators will sequence actions regardless of how the state is represented! In fact, in Haskell, `State` is a (nullary) data constructor whose sole purpose is to "fill in the slots" of the `IO` type:

```
type IO a = State -> (a,State)
```

but the `State` does not actually store anything at all.

2.1.1.2 Compound actions are constructed from sequences of primitive actions.

So how are actions constructed, if not by manipulating the state? Haskell provides primitive actions such as `putStr` and `getChar`, and the combinators support creating compound actions. The fragment of `do` notation above is an action composed from calls to `putStr` and `getChar`. Since each compound action is a sequence of other actions, any action can be "unrolled" into a sequence of primitive actions. Given a program, the Haskell implementation takes the sequence of primitive actions that comprise `main` and executes them (that is, their implementations) in order.

Again, actions are defined in Haskell, but are invoked in the Haskell implementation. The combinators >> and >>= define compound actions (consisting of sequences of primitive actions) within the purely functional world of Haskell. It is the Haskell implementation that invokes (executes) the actions in order.

Monadic I/O (via the >> and >>= combinators) creates an "imperative sublanguage" within the purely functional world of Haskell expressions. Equational reasoning is completely intact, yet portions of a Haskell program can be written in an imperative style to perform imperative tasks. As we will see in section 2.2.1, the dual role of actions hides some conceptual subleties.

2.1.2 The do notation is the canonical form for monadic I/O in Haskell.

Haskell provides an alternate notation for combinations of actions. This do notation hides the sequencing operators and supports a format resembling imperative programming languages. Discussion of monadic I/O in the Gentle Introduction to Haskell [HPJW+ 1992] is almost exclusively in terms of do notation. The combinator notation and do notation are equivalent, of course, and each can be transformed into the other:

Combinator Notation	do Notation
`action1 >>` `action2`	`action1` `action2`
`action1 >>= \ x ->` `action2`	`x <- action1` `action2`
`return expr`	`return expr`

Layout, which is irrelevant in the combinator notation, is used in the do notation to distinguish statements in the absence of the combinators. Because the do notation omits the combinators, we will refer to the first kind of statements as **simple statements**, the second kind as **pattern-binding statements**, and the third kind as **return statements**. Simple statements are equivalent to actions combined using >>, while pattern-binding statements are equivalent to actions combined using >>=. The `return` statement (action), which we have not discussed, is an empty action (that is, one that performs no I/O) that returns a value. It is used to return a value from a composite action, and has type

```
return :: a -> IO a
```

We will use the do notation throughout the remainder of this chapter except when discussing the differences between combinator notation and do notation.

2.1.3 In Haggis, monadic I/O plays a central role.

Haggis uses Haskell's use of the I/O monad to provide access to a wide range of interface components. The final version of the temperature conversion example (algorithm 1) exemplifies how extensively monadic I/O permeates Haggis programs. The program creates a display context, then creates all the components, specifies their layout, and forks an event handler for each of the two bars. The event handler waits until its bar changes value, then explicitly updates its label and the other bar (which then updates the other label). (Infinite recursion is averted by `waitGauge`, which waits for a value change, not merely an update.)

The functional and concurrency features produce an interesting twist: both event handlers are the same function, `track`, given different parameters! Another slightly unusual feature is that `track` is a recursive function with no base case. (Its evaluation does not "run away,"

Algorithm 1 Temperature Conversion Example in Haggis

```
module Main(main) where

import Concurrent
import Haggis

main =
 do
   dc                <- mkDC ["*title: TemperatureConversion"]
   (fBar,fBarDH)     <- mkBar (0,100) 32 YAxis dc
   (cBar,cBarDH)     <- mkBar (0,100) 0  YAxis dc
   (fLabel,fLabelDH) <- label "32" dc
   (cLabel,cLabelDH) <- label "0"  dc
   realiseDH dc (hbox [vbox [fBarDH, fLabelDH],
                       vbox [cBarDH, cLabelDH]])
   forkIO (track convertFtoC fBar cBar fLabel)
   forkIO (track convertCtoF cBar fBar cLabel)
   return ()
 where
   convertFtoC f = (5/9)*(f-32)
   convertCtoF c = (9/5)*c + 32
   track convert src dst srcLabel =
    do
      v <- waitGauge (getLevelGauge src)
      setLabel srcLabel (show v)
      setLevel dst (convert v)
      track convert src dst srcLabel
```

because each invocation of `track` must wait for the level to change.) Notice also that `track` is higher-order: it takes the conversion function as an argument.

The temperature conversion example fits reasonably well into the imperative perspective. As we will see, more complicated programs, such as the line-fitting example, uncover interesting issues.

2.2 The walkthroughs revealed cognitive obstacles to using monadic I/O.

The programming walkthroughs of the temperature conversion (Appendix A) and line-fitting (Appendix B) examples record the knowledge and actions a hypothetical user would need in order to write these two programs in Haggis. The discussion in this section refers to steps in the temperature conversion walkthrough by references of the form "(TC step 3)," and steps in the line-fitting walkthrough by references of the form "(LF step 13)."

A few notes on the walkthroughs are in order as far as the `do` notation is concerned. Haggis was written to run with version 0.26 of the Glasgow Haskell Compiler, which does not support the `do` notation. Since the `do` notation is clearly the preferred form for monadic I/O, and since the dual nature of monadic I/O (combinators versus `do`) is part of the focus of the walkthroughs, the walkthroughs accomodate both styles. Hence, some steps in each walkthrough appear in two forms: a combinator step (labeled, say, `Step 46`), and the same step in `do` notation (labeled `'Do' Step 46`).

The final programs and starting examples are presented in both combinator and `do` notation. Also, the `mapIO` operation used in the line-fitting walkthrough is a precursor to the `accumulate $ map` combination discussed in this section. Screen images of the final line-fitting and temperature conversion examples are shown in figure 2.1.

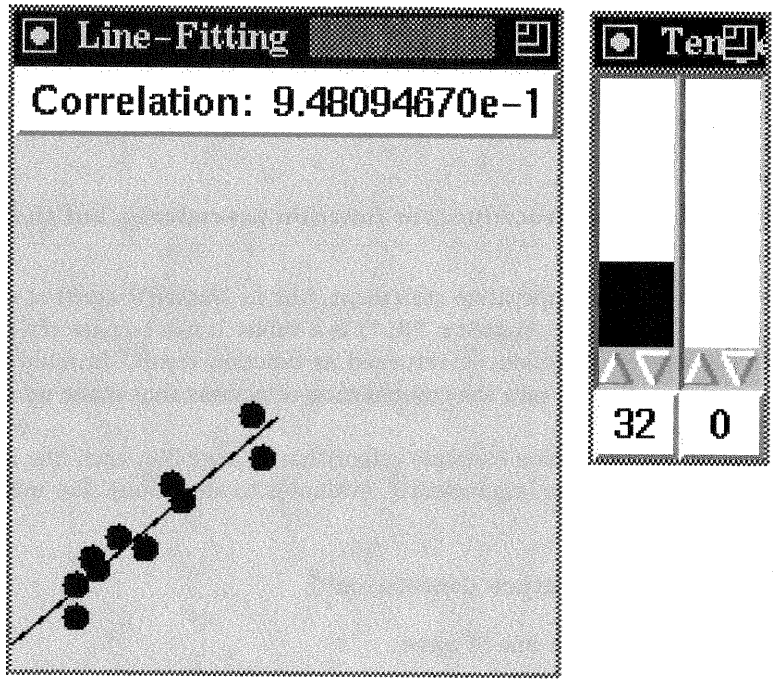


Figure 2.1: The final line-fitting and temperature conversion examples in Haggis.

2.2.1 Monadic actions simultaneously represent imperative statements and functional values.

As noted above, actions are special values that are defined in Haskell, but invoked by the Haskell implementation. Actions, together with the >> and >>= combinators, comprise an "imperative sublanguage" within purely functional Haskell. An important consequence is that the programmer must keep both the functional and imperative interpretations in mind, for the following reasons:

- Actions look like procedures or function procedures, but they are Haskell values.
- Actions can return values, but they are not Haskell functions.
- Actions can be sequenced in an imperative-looking style, but iterative and conditional structures are functional, not imperative.
- Iterative and control structures are expressions that evaluate to (compound) actions.
- Actions returning values can still be used as simple statements, hiding an important piece of information for reasoning about functional programs.
- The `do` notation reinforces the imperative interpretation at the expense of the functional interpretation.
- Sequences of "statements" in the imperative language scope and pattern-match imperatively.

As a result, the programmer cannot simply program one part of the program imperatively and another functionally. If `do` notation were merely an imperative sublanguage of Haskell, she or he would at least have the benefit of a clear separation between these two modes of thinking. The "double identity" (functional and imperative) of monadic I/O means that the programmer must not only deal with two sets of rules, but also how they interact with each other. She or he must be able to move smoothly between the functional and imperative interpretations of a single section of code.

2.2.1.1 Actions look like procedures or function procedures, but they are Haskell values.

An action represents an imperative statement, but in Haskell's world of expressions, an action is just a value. For instance, `(putStr "Hi")` is a value: it can be part of a data structure, it can be named, passed to a function, or returned as function result. In addition (because it is an action), it can be sequenced into the combination of actions that make up `main` so that it will be executed at runtime.

In the temperature conversion example (algorithm 1, page 30), each line in a `do` context (to the right of `<-`) is an action (or, equivalently, evaluates to an action). For instance, the first line of `main`:

```
mkDC ["*title: Temperature Conversion"]
```

is an action. So is the next-to-last line of `main`:

```
forkIO (track convertFtoC fBar cBar fLabel)
```

and the last line:

```
return ()
```

and so is the third-to-last line of the entire program:

```
setLabel srcLabel (show v)
```

All of these are actions. All of them are values, too. Any of them can be part of a data structure, passed to a function, bound to a name, and so on. All of them are returned by functions (`mkDC`, `forkIO`, `return`, and `setLabel`).

2.2.1.2 Actions can return values, but they are not Haskell functions.

Every action in Haskell returns a value when invoked. For example, the action `getChar`, when invoked, performs some action that returns a character. Actions return values, and Haskell functions return values, but these two features are completely separate.

Haskell functions can be used in expressions. Consider the expression:

```
(head listOfRocks):listOfStones
```

The `head` function, applied to the list `listOfRocks`, returns (evaluates to) the first element of the list of rocks, and that result can be used to form a new list consisting of the first rock followed by the list of stones.

In constrast, the action `getChar` cannot be used in expressions:

```
(getChar):moreCharacters            -- incorrect
```

might seem to be a concise way to read a character and return a list consisting of that character followed by the list `moreCharacters`, but `getChar` can only be used in an imperative context:

```
do
  c <- getChar
  return c:moreCharacters
```

The `do` context evaluates to an action that reads a character and then returns a list consisting of that character followed by the list `moreCharacters` .

Conversely, pure Haskell functions cannot be used as if they were actions. Writing

```
do
  r <- head listOfRocks             -- incorrect
  return r:listOfStones
```

would be completely nonsensical.

In the line-fitting example, a similar situation arises. Consider the `newElement` and `plotPoint` actions (algorithm 2). In `newElement`, if `transparentGlyph` were a Haskell function, the second element of its return value could be selected with the `snd` function:

```
-- incorrect
snd (transparentGlyph (withColour c $ fillSolid $ circle r) dc)
```

but the `snd` function does not work, because `transparentGlyph` evaluates to an action. Similarly, in `plotPoint`, if newElement were a Haskell function, the body of plotPoint could become:

```
-- incorrect
placeTransparent canvas coord (newElement colour)
```

but `(newElement colour)` evaluates to an action that returns a `DisplayHandle` (`dh`), not a `DisplayHandle` itself.

Algorithm 2 Line-Fitting Example (`newElement` and `plotPoint`)

```
newElement :: Colour -> IO DisplayHandle
newElement c =
  do
    (_,dh)
      <- transparentGlyph (withColour c $ fillSolid $ circle r) dc
  return dh

plotPoint :: CompositeContainer -> Colour
                                -> Coord
                                -> IO CompositeContainerElt
plotPoint canvas colour coord =
  do
    dh <- newElement colour
    placeTransparent canvas coord dh
```

Understanding the distinction between functions and actions comes into play in the line-fitting walkthrough. In step 111, the hypothetical user constructs the `drawLine` action based on the `newElement` and `plotPoint` actions (algorithm 2). Pasting the `newElement` definition verbatim into `plotPoint` yields:

```
do
  (_,dh) -> transparentGlyph (withColour c $ fillSolid $ circle r) dc
  return dh
  placeTransparent canvas coord dh
```

(The first two lines are from `newElement`, the last from `plotPoint`.) The hypothetical user realizes that the `return dh` line is redundant — it simply passes along the value of `dh` — and can be eliminated:

```
do
  (_,dh) -> transparentGlyph (withColour c $ fillSolid $ circle r) dc
  placeTransparent canvas coord dh
```

As in other programming contexts, knowledge about how to convert between functions and expressions (or between procedures and statement sequences) plays an important role in successful programming.

2.2.1.3 Actions can be sequenced in an imperative-looking style, but iterative and conditional structures are functional, not imperative.

We saw on page 28 that actions can be sequenced (and thus composed into a compound action) by placing them on successive lines:

```
do
  putStr "Press a lower-case letter key"
  c <- getChar
  putStr [toUpper c]
```

A similar format is also used with the combinator notation:

```
putStr "Press a lower-case letter key" >>
getChar                                >>= \ c ->
putStr [toUpper c]
```

although the placement of lambda expressions in the combinator notation differs from that of lambdas in other functional code.

Full-fledged imperative langauges provide other means for combining statements together, most notably conditional and looping constructs. Haskell's monadic I/O "imperative sublanguage" does not, relying instead on the functional tools that Haskell provides.

Suppose that we wish to write the `putStr` function shown above. The `putStr` function evaluates to an action that will print out a character string. It is defined in terms of `putChar`, which does the same for a single character. In an imperative language, we might write:

```
for i <- 1 to (length string) do      -- pseudocode
  putChar (string!!i)                 -- !! is list indexing
```

But the `do` notation does not provide iteration constructs! The functional approach is to use `map`:

```
map putChar string
```

Using `map` is elegant and concise, but it does not work. To understand why, we need to examine the types (in Haskell, `[a]` denotes the type of a list of items of type `a`):

```
string                :: [Char]
putChar               :: Char -> IO ()
map                   :: (a -> b) -> [a] -> [b]
(map putChar)         :: [Char] -> [IO ()]      -- pseudocode
(map putChar string)  :: [IO ()]                -- pseudocode
```

We see that `(map putChar string)` returns a list of actions, and none of them contain values we care about. The source of the trouble is still unclear.

The first problem is that `(map putChar string)` needs to be an action, not a list of actions. because the combinators that underlie the `do` notation sequence individual actions. The second reason is related, but more subtle. The list of actions returned by `(map putChar string)`, like all other lists in Haskell, is evaluated lazily ("by need"). The actions will not actually be evaluated unless their values are used by some other part of the program — and the only way to use actions is to sequence them with `>>` or `>>=` (or their `do`-context equivalents). The expression `(map putChar string)`, by itself, will not result in any I/O at all!

We might write

```
return (map putChar string)
```

to address the first problem: `return (map putChar string))` is an action. However, its type is `IO [IO ()]` — it is an action, which when invoked, returns a list of actions. The actions in the list (that is, all the `putChar` actions) will still never be sequenced, and thus never invoked.

Fortunately, the Haskell Standard Prelude provides:

```
sequence :: [IO a] -> IO ()
sequence = foldr (>>) (return ())
```

(The type is actually over all monads.) The correct solution is thus

```
sequence (map putChar string)
```

which not only has the correct type, that is, `IO ()`; it will also sequence the actions properly, thus ensuring their execution.

A similar issue arose in the line-fitting example (steps 45–52) around plotting the points. A function to plot a single point is relatively simple:

```
plotPoint :: CompositeContainer
          -> Colour
          -> Coord
          -> IO CompositeContainerElt
plotPoint canvas colour coord =
  dh <- newElement
  placeTransparent canvas coord dh
```

Here, the hypothetical user is in a position similar to the one we were describing with `putChar`: she or he would like to define

```
plotPoints :: CompositeContainer
           -> [Coord]
           -> ???        -- something here, probably IO type
plotPoints canvas = map (plotPoint canvas blue)
```

which will not work for the same reason that `(map putChar string)` did not — it returns a list of actions, not an action. The type of the list is `[IO CompositeContainerElt]`, so here the actions in the list return useful values that the hypothetical user needs. The `accumulate` function

```
accumulate :: [IO a] -> IO [a]
```

is like `sequence`, but it collects all the return values into a list instead of discarding them:

```
plotPoints :: CompositeContainer
           -> [Coord]
           -> IO [CompositeContainerElt]
plotPoints canvas = accumulate (map (plotPoint canvas blue))
```

The line-fitting walkthrough shows that the need for sequence and accumulate is a nonobvious consequence of the combination of a higher-order function and imperative actions that are also values (LF step 48).

2.2.1.4 The iterative and control structures are expressions that evaluate to (compound) actions.

Unlike the pseudo-iterative use of `map` in the previous section, conditionals like `if-then-else` and `case` do not collide with lazy evaluation. In fact, in `do` notation, they appear to be identical to their imperative counterparts. Consider the event handler from the line-fitting example (as of step 127, algorithm 3). (The code has been trimmed down to save space and provide better focus. The full final program is in Appendix B.)

Algorithm 3 `trackCtrl` (abbreviated) from the Line-Fitting Example, step 127

```
trackCtrl :: Mouse
          -> CompositeContainer
          -> Gauge [(Coord,Bool)]
          -> IO ()
trackCtrl mouse canvas theDataGauge =
  do
    deviceEvent <- getMouseDown mouse
    case whichButton of
      1 ->
        do
          -- toggle the point that was clicked
          -- update the data structure that holds the points
          -- (numerous statements omitted)
          return ()
      _ ->
        sendMouseEv mouse deviceEvent
    trackCtrl mouse canvas theDataGauge
```

Notice that one of the three statements in the outermost `do` sequence is a compound action consisting of a `case` expression (the other two are primitive actions). As it happens, the `case` expression is a simple statement (in other words, it does not return a useful value), and so the statement sequence comfortably resembles its analog in an imperative language.

At step 127 in the line-fitting example, the hypothetical user has just written a statement to draw the fitted line. She or he needs to keep track of the line (via its `CompositeContainerElt`)

so that it can be deleted when the line is redrawn after the next change. But the `CompositeContainerElt` for the line (named `theNewLineCCE` in the example) is created inside the `case` expression, and it needs to be returned from that expression (recall that the `case` expression evaluates to an action). (We are not considering here the well-known programming issue of keeping all branches of the case consistent.)

Six steps later, the `case` expression has been converted from a simple statement to a pattern-matching statement (algorithm 4):

Algorithm 4 `trackCtrl` (abbreviated) from the Line-Fitting Example, step 133

```
trackCtrl :: Mouse
          -> CompositeContainer
          -> Gauge [(Coord,Bool)]
          -> CompositeContainerElt
          -> IO ()
trackCtrl mouse canvas theDataGauge theLineCCE =
  do
    deviceEvent <- getMouseDown mouse
    thePossiblyNewLineCCE
      <- (case whichButton of
        1 ->
          do
            -- toggle the point that was clicked
            -- update the data structure that holds the points
            -- (numerous statements omitted)
            theNewLineCCE <- drawLine canvas theNewLineCoords
            return theNewLineCCE
        _ ->
          sendMouseEv mouse deviceEvent
          return theLineCCE)   -- end of case expression
    trackCtrl mouse canvas theDataGauge thePossiblyNewLineCCE
```

Notice the parentheses around the `case` expression. They are not strictly necessary, but are included to illustrate the extent of the expression. (In the line-fitting walkthrough, the `case` expression is 35 lines long.) Here, the analogy with imperative languages breaks down, because control constructs in imperative languages do not return values.

2.2.1.5 Actions returning values can still be used as simple statements, hiding an important piece of information for reasoning about functional programs.

Section 2.2.1.2 discussed how the return values of actions can be extracted with `<-`. However, the return value of an action need only be extracted if it is needed. Consider `tracker`, the setup function in the line-fitting example (algorithm 5; the full example is in Appendix B).

From the starting example, the hypothetical user has no information about whether placeTransparent returns a value, or what type that value might be. She or he might assume that it returns no useful value, that is, that its return type is `IO ()`. The real issue arises when the program is modified to use the return value of `placeTransparent`.

In the line-fitting walkthrough, the hypothetical user's initial version of the `drawLine` function (step 112, algorithm 6) discards the return value of `placeTransparent`. In step 113, the hypothetical user discovers that the `CompositeContainerElt` is needed from `placeTransparent`, and modifies the statement accordingly (step 114, algorithm 7). Making the modification shown in step 113 requires knowing that `placeTransparent` returns a useful value

Algorithm 5 The event handler for a press of button 2 in the starting example for the line-fitting walkthrough

```
2 ->
  dh <- newElement
  placeTransparent canvas (x,y) dh
  return ()
```

Algorithm 6 The `drawLine` function from the line-fitting example (step 112)

```
drawLine :: CompositeContainer ->
            (Coord,Coord)
            -> IO ???
drawLine canvas ((x1,y1),(x2,y2)) =
  do
    (_,dh)
       <- transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc
    placeTransparent canvas (r,r) dh
    -- not finished yet
```

Algorithm 7 The `drawLine` function from the line-fitting example (step 117)

```
drawLine :: CompositeContainer ->
            (Coord,Coord)
            -> IO CompositeContainerElt
drawLine canvas ((x1,y1),(x2,y2)) =
  do
    (_,dh)
       <- transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc
    lineCCE <- placeTransparent canvas (r,r) dh
    -- not finished yet
```

and knowing what type that value is. Neither of these can be learned from a program in which the value is discarded. Understanding this interaction between expression-oriented language primitives and the imperative orientation of monadic I/O and `do` contexts is complex, unintuitive, and error-prone.

2.2.1.6 The `do` notation reinforces the imperative interpretation at the expense of the functional interpretation.

As mentioned in section 2.1.2, the `do` notation hides the sequencing operators, which (in general) declutters the syntax and presents a more-or-less familiar imperative appearance. (The combinator notation has the disadvantage of not looking particularly imperative or particularly functional.) In situations where the role of the sequencing operators is conceptually important, their absence in the `do` notation is a drawback.

One such situation was discussed in sections 2.2.1.3 and 2.2.1.4: combining actions into other actions. Lists can be combined with `:` or `++`, functions with `.` or `$`, and so on. In each case, the combining operator is explicit. (Function application is expressed by juxtaposition, which is not an explicit operator, but function application is the most common operation in Haskell, and one of the ones that the programmer understands best.) In the `do` notation, the operator that combines actions is "appearing on successive lines!" In most programming languages, including Haskell, line breaks are either irrelevant, or they separate statements (in imperative languages, and in Haskell's `do` notation!), or (in Haskell) equations in a definition — to have a line break as a combinator is unprecedented, and understandably opaque.

The second situation has to do with the types and type signatures of actions. Since the type of a `do` construct is the same as the type of its last statement, the absence of combinators is not a liability in that instance. The plot thickens, however, when the program uses actions outside of the typical `do` structure; the need to reason about types becomes greater (for instance, in section 2.2.1.3), and the absence of the sequencing operators is more of a drawback.

The do notation reinforces the confusion between actions and functions (section 2.2.1.2), but the imperative appearance of the `do` notation also helps distinguish its unusual scoping and pattern-matching rules.

2.2.1.7 Sequences of "statements" in the imperative language scope and pattern-match imperatively.

Writing Haskell programs involves writing **definitions** that consist of one or more **equations**. Haskell has three semantic contexts for definitions: `let` expressions, `where` clauses, and the top level context. The `do` context differs sharply from these other three contexts in two areas: pattern-matching semantics and scoping. (The combinator notation also follows these rules, which are a consequence of the use of lambda expressions to extract the return values of actions.)

Pattern-matching semantics in `do` notation must be failure-free.

One of the central features of Haskell is the use of pattern-matching in definitions. Consider the definition of the length of a list [HPF 1997]:

```
length          :: [a] -> Int
length []       =  0
length (x:xs)   =  1 + length xs
```

The pattern-matching works as follows: in an expression, say `length [17,800]`, a match with the first equation fails, because `[17,800]` cannot match `[]`. A match with the second equation is tried (and it successfully matches `17` with `x` and `[800]` with `xs`).

Defining functions via multiple equations means that patterns used as function arguments need not match (like `[]` above); the next equation may contain a pattern that matches (like

`(x:xs)` above). (If none of the equations has a match, an error occurs.) Pattern-matches that may not succeed, such as `[]` and `(x:xs)` in the definition of `length` above, are called **refutable.**

In contrast, some pattern-matches always succeed (are **irrefutable**), such as:

```
oneToFive :: [Int]
oneToFive =  [1,2,3,4,5]
```

The reasons for the distinction between refutable and irrefutable pattern-matches have to do with recursion in pattern-matched definitions, and are beyond the scope of this discussion, but the implications are important.

Equations in which the left-hand side is a single pattern, like `oneToFive` above, are called **pattern-bindings**. Pattern bindings are always irrefutable; they always succeed (even when later use of the bound variables will cause an error). In the equation

```
(x:xs) = []
```

the match succeeds. Any use of `x` or `xs` will cause an error, but the definition itself is fine — and if `x` and `xs` are never evaluated, no error occurs.

The catch is that the pattern-matching semantics of `<-` in `do` notation are neither refutable nor irrefutable. Consider the statement

```
(x:xs) <- getLine
```

which will cause a static typing error. Why? Because pattern-bindings in do contexts are neither refutable nor irrefutable. They cannot be refutable (as patterns in function definitions can be) because there is only one binding (that is, there is no analogue to the multiple equations of a function definition). But they are not irrefutable, either, because `x` and `xs` will always be evaluated. (They are actually the arguments of a lamdba expression, and arguments to lambdas are strict, not lazy.)

The result is that the programmer cannot write the `getLine` statement above, even if she or he knows that `(x:xs)` will always match the return value of `getLine`, and even if the result of `getLine` goes unused. Being unable to use pattern-matching is unusual in Haskell.

Variables defined by `<-` are only in scope in the following statements.

The scope rules for `do` contexts are also unique within Haskell. Bindings in `let`, `where`, and top-level contexts are mutually recursive (that is, order is irrelevant). Consider

```
validIndexes = [1..maxIndex]
maxIndex = 10
```

The definition of `validIndexes` refers to `maxIndex`, but the definitions can occur in either order. (Equation binding contexts behave like `letrec` in Lisp or Scheme.) In the `do` context, however, order is relevant (as it is in Lisp's and Scheme's `let`):

```
do
  c <- getChar
  putChar c
```

Reversing the order of these two statements would cause an error: `c` in `putChar c` would be undefined. Here, the reason is clear: it would be ridiculous to try to print `c` before it has been read! In more complicated programs, these dependencies are less obvious. Just as imperative programmers must take care to sequence bindings in the proper order, so must functional programmers using `do` notation.

2.2.2 The cognitive obstacles to programming in Haggis center around the need to treat monadic I/O as both imperative and functional.

Monadic I/O is powerful, but it is also complicated. Actions look like procedures or function procedures, but they are Haskell values. They can return values, but they are not Haskell functions. Return values can be invisibly discarded, hiding an important piece of information for reasoning about functional programs. Iterative and conditional structures are not imperative constructs, but functional expressions that evaluate to compound actions. The `do` notation reinforces the imperative interpretation at the expense of the functional interpretation, and sequences of statements in either notation scope and pattern-match imperatively.

In short, the interference between the pure functional features of Haskell and the embedded imperative features of monadic I/O present cognitive challenges significantly greater than those of functional programming alone.

Chapter 3

Microsoft Excel with Visual Basic

Microsoft Excel, a popular commercial spreadsheet, and its extension language, Visual Basic for Applications, are designed for users with relatively little programming experience. Although Excel provides the advantages common to spreadsheets generally, Excel Visual Basic has design weaknesses and, more broadly, suffers from the conflict between the imperative model underlying Visual Basic and the expression-oriented model of spreadsheets.

Section3.1 sketches the structure of Excel with Visual Basic. Section 3.2 (page 45) describes the design weaknesses discovered by the programming walkthrough method, and section 3.3 (page 52) articulates the conceptual conflict between the spreadsheet and imperative models. Finally, section 3.4 (page 56) relates the Excel/Visual Basic combination to the design principles underlying Esquisse.

3.1 Visual Basic for Applications is an extension language for Microsoft Excel.

Microsoft Excel[1] is a popular commercial spreadsheet application. Visual Basic[2] (specifically, the Microsoft Excel implementation of the Microsoft Visual Basic Programming System, Applications Edition[3]) is an extension language provided with Excel for automating tasks, adding custom features and functions, and even creating complete applications [Microsoft 1994].

Visual Basic is an extension of BASIC. Visual Basic (like BASIC) is an imperative programming language with variables, scoping, iteration, conditionals, procedures and functions. It also provides an integrated editor (with syntax checking) and (of greatest interest to us) graphics and interface support.

3.1.1 Visual Basic contains Excel-specific features.

As an embedded application language, Visual Basic has an extensive set of Excel-specific features in addition to its general-purpose programming constructs. These features include data structures that correspond to spreadsheet entities (for example, ranges of cells), and access to virtually all spreadsheet functions. Since this spreadsheet-specific functionality is much higher-level than the programming primitives of BASIC (and since interface programming uses this functionality heavily), Visual Basic seems much more reminiscent of Excel than BASIC.

3.1.1.1 Excel-specific features form Visual Basic's object model.

Visual Basic has objects that correspond to Microsoft Excel entities. Examples include cells, ranges of cells, worksheets, charts, controls such as scrollbars, graphic objects, and files. Certain collections of objects (for example, "all of the rectangles on the active sheet") are also

[1] Microsoft Excel is a trademark of Microsoft Corporation.

[2] Visual Basic is a trademark of Microsoft Corporation.

[3] In this chapter, Visual Basic refers to the version of Visual Basic for Applications bundled with Excel, not the standalone Visual Basic language and development environment.

objects. Some objects (such as `ActiveSheet`, `ActiveCell`, `Selection`) reflect the current state of the interface.

3.1.1.2 Macro recording provides another link between Excel and Visual Basic.

Visual Basic replaces and augments the spreadsheet macro language of previous versions of Microsoft Excel. Any manipulation that the user performs with the interface can be captured by the macro recording facility, yielding a Visual Basic procedure that reproduces the direct-manipulation actions.

3.1.1.3 User actions can trigger Visual Basic procedures

Visual Basic procedures have another important connection to Excel: they can handle events that occur in the Excel interface. Visual Basic does this by the use of **assigned macros**: parameterless procedures bound to an object that are called when an event occurs on that object. Macros can be assigned via a menu option or from within Visual Basic.

3.1.2 The spreadsheet model contributes both positively and negatively.

Since Visual Basic is tied in to Excel, the utility and useability of the Visual Basic extension depends in part on the power and ease of use of the spreadsheet model. Much of what has been learned about spreadsheets applies to the Excel/Visual Basic combination.

3.1.2.1 The presence of the spreadsheet offers powerful computational and interactive leverage.

One of the most attractive features of Microsoft Excel with Visual Basic is that the spreadsheet alone is so powerful, both computationally and interactively: spreadsheets store data in a visible, manipulable format, and Excel, like virtually all spreadsheets, has an extensive library of built-in functions. Since spreadsheet formulae are one-way constraints that are automatically maintained, some of a program's "responses" to interactive events can be handled by formulae.

Visual Basic can make good use of these spreadsheet features: it can access and update spreadsheet cells, and Visual Basic procedures can be triggered by spreadsheet events, such as the user editing a cell or the spreadsheet recalculating. As a result, the computation and some of the interface functionality and constraints can be handled by the spreadsheet, and the Visual Basic procedures can control and respond to the spreadsheet.

3.1.2.2 The combination of the Excel spreadsheet and Visual Basic has certain drawbacks.

These advantages are accompanied by certain shortcomings. Spreadsheets do not support modularity nor abstractions, which help manage the complexity of user-defined functionality, and they can be difficult to debug or redesign [LO 1987, p. 252]. The use of spreadsheet functions for interface constraints is hampered by their unidirectionality, as many interface constraints are bi-directional. Because a spreadsheet cell can contain either a value or a formula, but not both, reciprocal pairs of unidirectional constraints are out of the question.

Visual Basic suffers, too, partly by comparison with Excel's extensive library of built-in functions and aggregate operations. Like those in many traditional programming languages, Visual Basic's core primitives are powerful only in combination, and synthesizing the proper combination for a particular task requires substantial knowledge and skill [LO 1987, p. 253].

Visual Basic does have an object hierarchy with objects, properties, and methods — but only for the existing Excel object structure. There is no facility for user-defined objects, nor for inheritance (either class or prototype). As a result, Visual Basic's object model is not useful for constructing new abstractions, but only for using existing ones.

In fairness, Visual Basic is presented as a tool for adding custom interface functionality to Excel, and should be considered in that light. Even so, although Visual Basic and Excel complement each other, there are areas, such as support for abstraction, where neither does particularly well.

3.2 The walkthroughs revealed challenges due to uneven design.

Excel and its Visual Basic present a conceptual counterpoint to Haskell's monadic I/O system. While monadic I/O is the result of careful theory-driven crafting, spreadsheets and the BASIC language (upon which Visual Basic is built) were designed to be approachable by non-programmers (or at least, in the case of BASIC, novice programmers). As a result, a number of the cognitive challenges presented by Microsoft Excel and Visual Basic for Applications stem from arbitrary limitations and (apparently) ad hoc design decisions, rather than from the emergent consequences of a theoretical design.

The programming walkthroughs of the temperature conversion (Appendix C) and line-fitting (Appendix D) examples record the knowledge and actions a hypothetical user would need in order to write these two programs in Microsoft Excel with Visual Basic. The discussion in this section refers to steps in the temperature conversion walkthrough by references of the form "(TC step 23)," and steps in the line-fitting walkthrough by references of the form "(LF step 22)." Screen images of the final temperature conversion and line-fitting examples are shown in figures 3.1 and 3.2, respectively.

3.2.1 The macro recording feature facilitates learning Visual Basic, but is inflexible.

A useful start to a Visual Basic extension program is the macro recording facility, for two reasons. First, it enables users to write procedures by direct manipulation. Both the line-fitting and temperature conversion walkthroughs made use of the this advantage. Second, macro recording converts common (and for most users, accessible) interface knowledge into information about writing groups of statements in Visual Basic. Both walkthroughs used macro recording to gather information about programming-language statements that duplicated desired interface functionality.

Since novice programmers have difficulty learning what primitives are appropriate to the task at hand and how to synthesize them to produce the desired outcome, we might expect macro recording to aid the transition from end user to programmer. Neither walkthrough offers much support for this view, because a significant amount of knowledge is required to understand the resulting procedure. There are several reasons for this.

Since macro statements are not displayed as the actions are taken, the programmer must figure out after recording which statements correspond to which actions (TC step 23, LF step 22). (Actually, she or he can switch to a Module and look at the macro being recorded; unfortunately, the macro will also record the switch to the Module, adding to the clutter.)

The linear, context-insensitive nature of macro recording is inflexible. The macro recording facility creates only one combination of primitives that produce the same outcome as the user's action. Since the macro recorder cannot know the user's intent, the procedure produced may or may not be well-suited to the user's needs. And excellent example of this is absolute versus relative cell addressing (TC step 43). Each has its uses, but the macro recorder has little information to help it guess which addressing mechanism is appropriate to the user's task.

Another limitation is that macros can only record what the interface can do, and thus are no help with non-interface Visual Basic features. Concepts such as the object hierarchy, attributes, and selection are not supported by macro recording. Recorded procedures also lack conditional and iteration constructs, which are crucial to programming and challenging to understand (LF step 33). Furthermore, the statements generated in a single recording session are

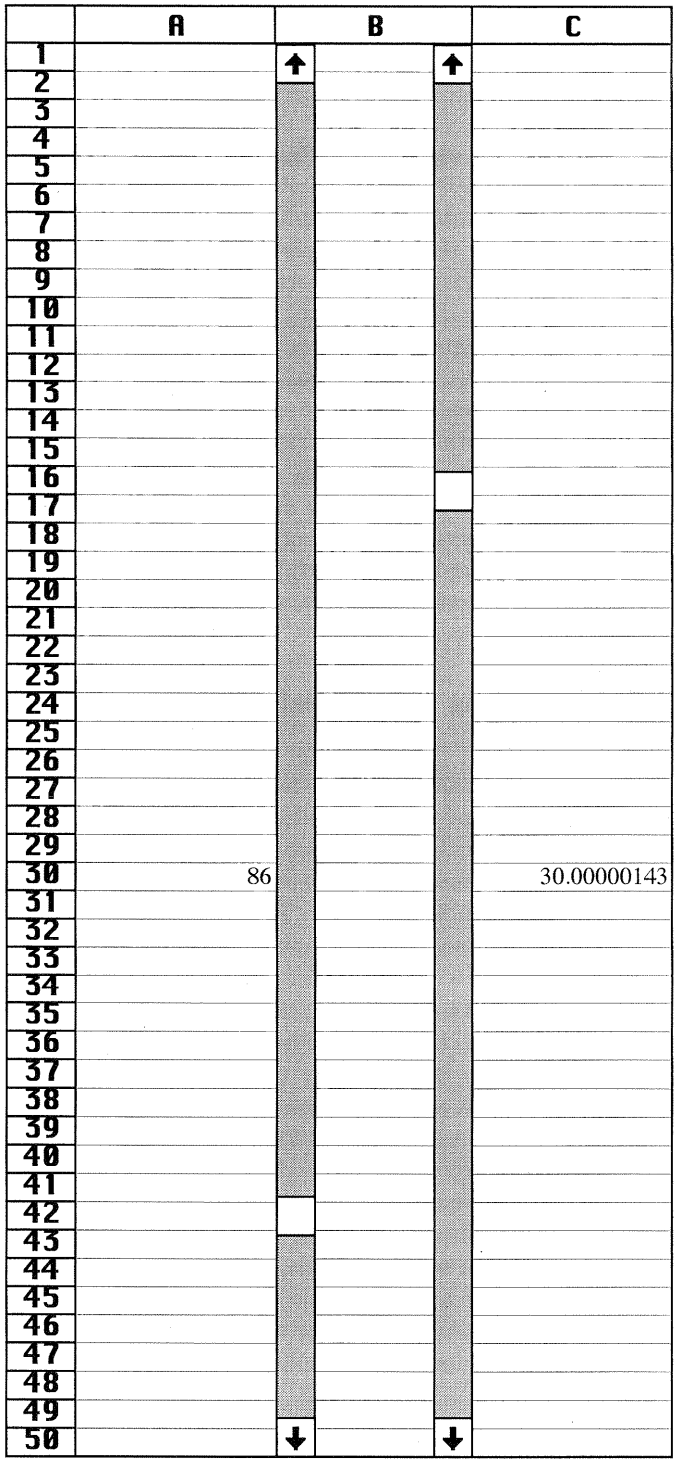


Figure 3.1: Final temperature conversion program in Microsoft Excel with Visual Basic.

	A	B	C	D	E	F	G	H	I	J	K
1	Number of Points:			10							
2											
3	All Data		Selected Data								
4	x	y	selected?	x	y						
5	90	96	TRUE	90	96						
6	94	80	FALSE								
7	40	50	TRUE	40	50						
8	24	20	TRUE	24	20						
9	24	32	TRUE	24	32						
10	30	42	TRUE	30	42						
11	50	46	TRUE	50	46						
12	32	38	TRUE	32	38						
13	60	70	TRUE	60	70						
14	64	64	TRUE	64	64						
15											
16											
17	Slope:		0.9908								
18	Intercept:		5.3105								
19	Correlation:		0.9636								
20							x	y	x'	y'	
21	Bounding Box (xmin, ymin, xmax, ymax):						0.0000	0.0000	100.0000	100.0000	
22	Vertical Intercepts (x_1, y_1, x_2, y_2):						0.0000	5.3105	100.0000	104.3940	
23	Fitted Line (x1, y1, x2, y2):						0.0000	5.3105	95.5654	100.0000	

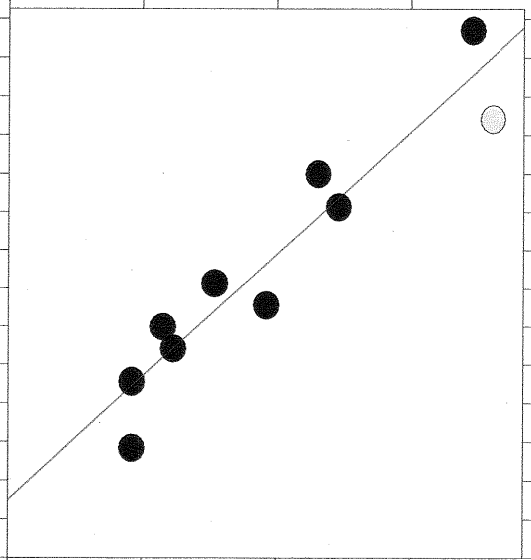


Figure 3.2: Final line-fitting example in Microsoft Excel with Visual Basic.

placed in a single procedure, which does not help (in fact, probably hinders) learning about modularization of a program into procedures.

Macros can also provide irrelevant or undesired information. In the temperature conversion example (step 23), linking a scrollbar to a cell causes all seven of the attribute values to be recorded in the macro, even the six that did not change (algorithm 8):

Algorithm 8 Macro recorded during the temperature conversion walkthrough (step 23).

```
Sub Macro1()
    ActiveSheet.ScrollBars.Add(74, 2, 15, 649).Select
    With Selection
        .Value = 0
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
        .LinkedCell = "$A$30"
        .Display3DShading = True
    End With
    ActiveSheet.ScrollBars.Add(131, 2, 15, 648).Select
    With Selection
        .Value = 0
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
        .LinkedCell = "$C$30"
        .Display3DShading = True
    End With
End Sub
```

Finding the seventh one that did change is thus more difficult. In the macro from the line-fitting example (step 23, algorithm 9):

Color attributes are set, not to color values, but to color table indexes. When the user only clicked on one color, she or he can conclude (cautiously) that the index saved corresponds to the color she or he clicked. If the user clicked on several colors in succession, the color attribute for that object is set several times, increasing the potential for confusion. For example, in `Macro1`, the first color (with an index of 2) to which the oval is set is the default color, not the color in which the user was drawing — creating the oval and setting its color produced not one but two color-setting statements.

In short, we note that macro recording is most effective when the user knows and can perform the exact sequence of direct-manipulation actions required. A clear-cut action sequence greatly simplifies the macro by eliminating the clutter of backtracking and changes (TC step 20, LF steps 20 and 66).

3.2.2 Knowing when to use the Excel's computational and interactive leverage requires solid knowledge of the capabilities of both Excel and Visual Basic.

Unfortunately, the application provides little help regarding when to use these features. Knowing when to use Visual Basic procedures, when to record a macro, and how to set up a mixed spreadsheet/Visual Basic program (that is, almost any Visual Basic program) can be a tricky task.

The user must figure out when Visual Basic procedures are necessary, and when the spreadsheet can do the job. For instance, the temperature conversion example specifies that two cells and two scrollbars be connected so that a change to any of these four components will change the rest accordingly. The Excel interface supports linking a scrollbar to a cell, so that a change to either will change the other, so constructing two scrollbar-cell pairs is relatively easy (TC step 5). Linking one cell to another via a formula is also a standard interaction in Excel — it is the canonical computation — but it cannot be done in this case, because cells linked to scrollbars cannot contain formulae, and because cells cannot contain both formulas and values (TC step 2). Thus the user's likely first intution about using formulae to perform the desired temperature conversion is completely useless, if not counterproductive.

In contrast, in the line-fitting example, the user's intuition to perform computation in the spreadsheet is essential (LF steps 1 through 7). The slope, intercept and correlation of the group of points can be easily calculated by spreadsheet functions that refer to the range of spreadsheet cells in which the points are stored. Since these functions ignore blank cells in the ranges, computing these functions for the selected points is only slightly more complicated than computing them for all of the points (LF step 4). Calling the spreadsheet function and accessing the spreadsheet cells from Visual Basic is significantly more cumbersome (and of course, writing the functions in Visual Basic would be a substantial undertaking).

The user must also have a good grasp of when macro recording is useful and when it is not. Macro recording is useful when:

- the desired program needs to do something that the user can do by direct manipulation, and
- the user knows how to do that something by direct manipulation, but not by programming.

In both walkthroughs (TC steps 8 and 10, LF step 15), it was clear that the user needed to know how to create scrollbars or graphic objects by direct manipulation. Unless the user has a thorough knowledge of the interface, she or he might not assume that scrollbars (for instance) can be laid out by direct manipulation (TC step 8). Once knowing that, the user would almost certainly choose to use direct manipulation and macro recording to find out the necessary programming statements.

Knowing that the program needs to create the scrollbars (in the temperature conversion example) or the graphic objects (in the line-fitting example) is less obvious (TC step 9, LF step 14). Since macros assigned to objects (including graphic objects and controls) do not stay assigned when the workbook (collection of spreadsheets and program modules) is closed and reopened, they must be reassigned each time Excel is run and the workbook containing the program is opened. (Perhaps an "assign macro to be run when this workbook is opened" option would address this problem effectively).

Algorithm 9 Macro recorded during the line-fitting walkthrough (step 23).

```
Sub Macro1()
    ActiveSheet.Rectangles.Add(449, 27, 402, 369).Select
    Selection.Interior.ColorIndex = 2
    ActiveSheet.Ovals.Add(797, 54, 16, 14).Select
    Selection.Interior.ColorIndex = 2
    Selection.Interior.ColorIndex = 5
    Selection.Interior.ColorIndex = 6
    ActiveSheet.Lines.Add(483, 346, 808, 78).Select
    Selection.Border.ColorIndex = 3
End Sub
```

The user has two basic options. One is to manually reassign each macro to the appropriate object when the workbook is opened. Given that there may be a number of objects, each needing a different macro, this approach requires the user to either recall all the relationships or record them explicitly.

The other alternative is to have Visual Basic "in charge" of the entire setup, including creating the controls and/or graphic objects. With this approach, the user need only remember to run the "set up everything" macro upon opening the workbook. The drawback, of course, is that the user must construct a complete program, including the parts that could be left to direct manipulation. This alternative was chosen in both walkthroughs (TC step 9, LF step 14). However, for constructing the direct-manipulation parts of the program, macro recording is the easiest way to "write" the Visual Basic code.

3.2.3 Visual Basic's object structure is somewhat formidable to the novice.

Visual Basic has objects that correspond to Microsoft Excel entities, some of which reflect the current state of the interface. As in object-oriented programming, objects are operated on by their properties and methods. For instance, graphic objects and controls are created by an `Add` method. When objects are created by direct manipulation, they remain selected. Conceptually paralleling this, Visual Basic supports creation and selection in a single statement, such as this one from the line-fitting recorded macro (LF step 23) shown earlier:

```
ActiveSheet.Rectangles.Add(449, 27, 402, 369).Select
```

When an object is selected, its properties and methods are also available via the selection, and in fact, accessing and modifying the properties and methods of the selection is often much easier than determining Visual Basic's static name for that object. (Visual Basic's name for the newly created rectangle would be something like `ActiveSheet.Rectangles(1)`.)

Unfortunately, both walkthroughs suggest that the object structure of Visual Basic is somewhat formidable to the novice user (which is, in fact, a major reason why manipulating the selection is easier). The code fragment above suggests a reading like "on the active sheet, with its rectangles, add somehow or maybe select or somehow both ..." (TC steps 22 and 26, LF step 24). The presence of two "actions," `Add` and `Select`, in the same statement, compounds the inference problem, as the user must confront the pair of actions as a combination before having an opportunity to understand them separately. The name `Add` is not particularly suggestive: `Draw` or `Create` might be better (LF step 22). Finally, the use of the plural `Rectangles` is confusing, because before this statement executes, there are no rectangles. To call a method of a nonexistent group of objects to create one of them seems unnecessarily obscure.

Putting all of this together, it's hard to justify the user's successful comprehension of this code fragment. In fact, the chief leverage available to the hypothetical user in each walkthrough was the lack of other possible matches to the user's actions: "I drew something, and these are the only statements that were recorded" (TC step 23, LF step 22).

3.2.4 Visual Basic procedures can be triggered by some events, but not others.

What makes Visual Basic most useful, of course, is not the ability to create screen objects (or any other interface-duplicating functionality), but the ability to handle events. Visual Basic does this by the use of **assigned macros**: parameterless procedures bound to an object that are called when an event occurs on that object.

Knowing how to use assigned macros involves four challenges: knowing what event triggers are available, knowing which one is appropriate, knowing how to assign procedures to objects and events, and knowing what events will trigger the macro.

Of these three, assigning macros is probably the easiest, since the user can select an object by direct manipulation and use the "Assign Macro... " menu command. With macro recording, the user can see how Visual Basic assigns macros. In the walkthrough of the line-fitting example

(step 67), with macro recording on, the user selects an oval and assigns a (preexisting) macro to it. The procedure that Visual Basic writes is:

```
Sub Macro2()
    ActiveSheet.DrawingObjects("Oval 2").Select
    Selection.OnAction = "TogglePoint"
End Sub
```

where `TogglePoint` is the macro the user chose to assign.[4] So assigning a macro is simple: set the `OnAction` property (or other property, see below) to a string containing the name of the macro (see also TC step 37). To users with programming knowledge, the use of a string and the lack of mention of potential parameters may be confusing (LF steps 69 and 70).

The recorded macro also assists with two other challenges: it informs the user of the existence of the `OnAction` property, and that it can be used with graphic objects. In general, the on-line help or the user manual are the ways to learn about available event triggers and their uses; only a few can be discovered by macro recording. The documentation is also necessary for the user to find out what is not available (a much harder task; TC step 47, LF step 73); for instance, dragging an object cannot trigger a macro (TC step 3).

The appropriateness of formulae varies significantly across problems. In the temperature conversion example, there are no spreadsheet formulae, there only two cells that can be edited, and the procedure called must be able to determine which cell was edited (so that it can update the other one; TC step 46 and 49). In this case, the `OnEntry` trigger (TC steps 47 and 48) is the appropriate choice. In the line-fitting example, there are a large number of cells and formulae, and there is no need to determine which cell was edited (if any), so the `OnCalculate` property (LF step 73) is by far the best (it also helps with the case in which a point is clicked, and the user has not interacted with any cells at all.)

Knowing what events will trigger the macro is perhaps the most subtle of the challenges. For instance, the `OnEntry` callback (used in the temperature conversion example) is triggered when a cell is edited by the user, but not when the cell is changed by a procedure, a spreadsheet formula, or even by the user via a cut or paste operation (TC step 33).

The user must also know how spreadsheet functions, linked cells, and explicit Visual Basic procedures interact with each other — or at least, in what order they are evaluated. Consider the temperature conversion example (TC step 42). When one of the two scrollbars is moved, the cell linked to it will change, and then the Visual Basic procedure that updates the other cell and scrollbar will be called. This ordering is important, because the procedure uses the value in the linked cell, and it only makes sense to do so if the cell and scrollbar have the same (new) value. Knowing this fact saves the user some work and some extra programming, greatly simplifiying the writing of the temperature conversion example.

3.2.5 Finding out how a Visual Basic procedure was called is complicated.

In contrast to many other interactive toolkits, Visual Basic procedures assigned to graphic objects, controls, worksheets, and so on do not take arguments (TC step 49, LF step 56). In many situations, of course, the appropriate response depends on which object was the subject of the interaction. In the line-fitting example, a single procedure is called regardless of which point is clicked, yet that procedure needs to toggle the status of that point before recalculating the correlation and fitted line (LF 56). In the temperature-conversion example (TC step 49), a single procedure responds to `OnEntry` events, but the procedure needs to update the unedited cell from the edited one, not vice versa.

Visual Basic's approach is to set the `Caller` property of the `Application` object to the non-Visual Basic entity that called Visual Basic (TC step 51, LF step 57). A procedure called by

[4] Note that in this statement, the oval is referred to as `ActiveSheet.DrawingObjects("Oval 2")`, whereas the statement that created it used `ActiveSheet.Ovals` — another unjustified complication.

another procedure will have access to the name of the outside entity, and not to the procedure that called it. The `Caller` property can be set to a diverse range of values: cell references, graphic objects, menu items, and so on, as appropriate. In the temperature conversion example (TC step 54), when a cell is edited, the `Caller` property is set to that cell's cell reference; `Cells(30, 1)`, for example.

In the line-fitting example (LF step 57), the `Caller` property is set to the name (`"Oval 2"`) of the point (graphic object) that was clicked. Unfortunately, a name of the form `"Oval 2"` is not useful as is. The user must use the `Evaluate` method of the `Application` object to convert this name into a useful index. Thus, the proper incantation is (LF step 57):

```
index = Application.Evaluate(Application.Caller).Index
```

which means "find out who the caller is, get the object associated with that name, and then get that object's index." (Notice that, much as with the `Add` and `Select` situation above, the statement returns the value of a property of the return value of a function call. Expressing several actions in a single statement is concise and powerful, but the need to synthesize a complex statement from several simple parts, which themselves are not well understood, is a common obstacle to learning traditional programming [LO 1987].) In any event, the index (which records the order of creation) can be used to compute the row in the spreadsheet that holds the coordinates and status of the clicked point; further conversion or translation is not necessary.

The alternative, having a separate procedure assigned to each object, is a reasonable solution if the objects have different behaviors or the number of objects is small and known a priori. When a large number of objects are created, or the exact number is not known initially, the programmer is stuck with `Application.Evaluate(Application.Caller).Index`. In the case of cell editing events, separate procedures are not an option, because macros can only be assigned to the `OnEntry` properties of `Applications` and `Worksheets` (not to cells or ranges) and thus the programmer must use `Application.Caller` (but fortunately not `Application.Evaluate`) to determine which cell was edited (TC step 58).

3.3 The programming models of spreadsheets and Visual Basic conflict.

The programming walkthrough analysis of Excel and Visual Basic identified limitations and design flaws that are relatively shallow. What about deeper issues? Are the spreadsheet and imperative programming models compatible? An example-based analysis shows a significant mismatch between the two.

3.3.1 Users will need to add functionality to write programs.

Consider the correlation computation in the line-fitting example. Microsoft Excel provides a correlation function, and the walkthrough uses this built-in function. In the general case, programmers will need functionality that is not built-in (no matter how extensive the built-in functionality may be), and they will need to create it.

The correlation r is calculated from three quantities: S_{xx}, S_{yy}, and S_{xy}(algorithm 10), which in turn depend on the sums of the x and y values ($\sum x$ and $\sum y$), the sums of their squares ($\sum x^2$ and $\sum y^2$), and the sum of their pairwise products ($\sum xy$).

Suppose, that Excel did not include a correlation function. The programmer could write the correlation function in essentially two ways: in the spreadsheet, or in Visual Basic.

3.3.2 The spreadsheet and the extension language are both environments for specifying computations.

Figures 3.3

Algorithm 10 Least-Squares Correlation for Paired Data (Lyman Ott, Statistics)

$$r = \frac{S_{xy}}{\sqrt{S_{xx}S_{yy}}}$$

$$S_{xx} = \sum x^2 - \frac{(\sum x)^2}{n}$$

$$S_{xy} = \sum xy - \frac{\sum x \sum y}{n}$$

$$S_{yy} = \sum y^2 - \frac{(\sum y)^2}{n}$$

	A	B	C	D	E	F	G
1	Number of Points:			10			
2							
3		All Data			Squares and Products		
4		x	y		x^2	y^2	x*y
5		90	96		8100	9216	8640
6		94	80		8836	6400	7520
7		40	50		1600	2500	2000
8		24	20		576	400	480
9		24	32		576	1024	768
10		30	42		900	1764	1260
11		50	46		2500	2116	2300
12		32	38		1024	1444	1216
13		60	70		3600	4900	4200
14		64	64		4096	4096	4096
15							
16		x	y		x^2	y^2	x*y
17	Sums:	508	538		31808	33860	32480
18							
19					Sxx	Syy	Sxy
20					6001.6	4915.6	5149.6
21	Correlation:						
22	Built-In Function:			0.94809			
23	This Spreadsheet			0.94809			
24	Visual Basic Function			0.94809			

Figure 3.3: Spreadsheet implementation of the correlation function.

and 3.4 show a spreadsheet implementation of the correlation function. The spreadsheet stores the data in two columns, one for x, one for y, and uses simple formulae to create columns of squares of the x and y values, and a column of their products. Summing formulae at the bottom of the columns compute $\sum x$, $\sum y$, $\sum x^2$, $\sum y^2$, and $\sum xy$. The final four formulae compute S_{xx}, S_{yy}, S_{xy}, and r. (In spite of the `A1`-style names used in the spreadsheet, the calculation is readable.)

A Visual Basic implementation appears in algorithm 11. A loop from `1` to `n` computes

Algorithm 11 Visual Basic implementation of the correlation function

```
Function MyCorrelation(xyRange)
  sumX = sumX2 = sumY = sumY2 = sumXY = 0
  n = xyRange.Rows.Count
  For i = 1 To n
    sumX = sumX + xyRange(i, 1)
    sumX2 = sumX2 + xyRange(i, 1) * xyRange(i, 1)
    sumY = sumY + xyRange(i, 2)
    sumY2 = sumY2 + xyRange(i, 2) * xyRange(i, 2)
    sumXY = sumXY + xyRange(i, 1) * xyRange(i, 2)
  Next i
  sxx = sumX2 - (sumX * sumX) / n
  syy = sumY2 - (sumY * sumY) / n
  sxy = sumXY - (sumX * sumY) / n
  MyCorrelation = sxy / Sqr(sxx * syy)
End Function
```

$\sum x$ (`sumX`), $\sum x^2$ (`sumX2`), $\sum y$ (`sumY`), $\sum y^2$ (`sumY2`), and $\sum xy$ (`sumXY`), which are then used to compute S_{xx}, S_{yy}, S_{xy}, and finally the correlation.

3.3.3 The two computational environments (the spreadsheet and the extension language) should be as compatible as possible.

Ideally, the spreadsheet formula implementation and the Visual Basic implementation of the correlation function should be as close to identical as possible. Mastering a single computational model is easier for the user than mastering two different models. Furthermore, since the spreadsheet and Visual Basic have different uses, the user might want to divide the computation between the two models. The user might also wish to translate a computation from Visual Basic into the spreadsheet or vice versa. Dividing or translating the computation is greatly facilitated by a shared computational model. The user will have greater flexibility if translating a computation from one environment to the other is simple and straightforward.

A comparison of figures 3.3 and 3.4 with algorithm 11 shows some similiarity and some sharp differences. The last four lines of the Visual Basic implementation map nicely onto the formulae in rows 20 and 22–24 of the spreadsheet implementation. S_{xx}, S_{yy}, S_{xy}, and the correlation are specified by expression bindings (formulae in the spreadsheet, assignment statements in Visual Basic).

The assignment statements in the `for` loop do not map over as well to the spreadsheet. For instance, in the statement

```
sumX2 = sumX2 + xyRange(i, 1) * xyRange(i, 1)
```

the expression

```
xyRange(i, 1) * xyRange(i, 1)
```

	A	B	C	D	E	F	G
1	Number of Points:			10			
2							
3		All Data					
4		x	y		x^2	y^2	x*y
5		90	96		=B5*B5	=C5*C5	=B5*C5
6		94	80		=B6*B6	=C6*C6	=B6*C6
7		40	50		=B7*B7	=C7*C7	=B7*C7
8		24	20		=B8*B8	=C8*C8	=B8*C8
9		24	32		=B9*B9	=C9*C9	=B9*C9
10		30	42		=B10*B10	=C10*C10	=B10*C10
11		50	46		=B11*B11	=C11*C11	=B11*C11
12		32	38		=B12*B12	=C12*C12	=B12*C12
13		60	70		=B13*B13	=C13*C13	=B13*C13
14		64	64		=B14*B14	=C14*C14	=B14*C14
15							
16		x	y		x^2	y^2	x*y
17	Sums:	=SUM(B5:B15)	=SUM(C5:C15)		=SUM(E5:E15)	=SUM(F5:F15)	=SUM(G5:G15)
18							
19					Sxx	Syy	Sxy
20					=E17-(B17*B17)/D	=F17-(C17*C17)/D1	=G17-(B17*C17)/D1
21	Correlation:						
22	Built-In Function:			=CORREL(C5:C14,B5:B14)			
23	This Spreadsheet			=G20/SQRT(E20*F20)			
24	Visual Basic Function			=mycorrelation(B5:C14)			

Figure 3.4: Spreadsheet implementation of the correlation function (formula view).

computes x_i^2 for the current loop index, closely resembling the computation carried out by the formulae in column E (rows 5–14). Computation of y_i^2 and $x_i y_i$ is analogous.

The poorest mapping is between the assignment part of the statements in the `for` loop and the summation formulae on row 17. The expression-based computation of x_i^2 above fits into its assignment statement

`sumX2 = sumX2 +` x_i^2

but the assignment part of the statement has no analog in the spreadsheet, and the assignment statement computes $\sum x^2$ only in combination with the `for` loop.

3.3.4 Translating the correlation function from spreadsheet formulae to Visual Basic (and vice-versa) means re-expressing the computation.

To convert the Visual Basic `MyCorrelation` function into a group of spreadsheet formulae, the user must create a column of formulae for the x_i^2, y_i^2, and $x_i y_i$ expressions in the assignment statements in the `for` loop, then place summation formulae for $\sum x$, $\sum x^2$, $\sum y$, $\sum y^2$, and $\sum xy$ at the bottoms of the columns. To convert the spreadsheet formulae that compute the correlation into Visual Basic code, the user can call the spreadsheet's `Sum` function from Visual Basic to compute $\sum x$ and $\sum y$, but she or he must still write a loop to compute $\sum x^2$, $\sum y^2$, and $\sum xy$, because the x_i^2, y_i^2, and $x_i y_i$must be calculated individually, then summed.

3.3.5 The spreadsheet formula language is not compatible with an imperative language.

In short, the spreadsheet and imperative computational models look to be a poor match. Although imperative languages have some expression-oriented features (expressions and pure functions), typical imperative programs, such as `MyCorrelation`, are highly stateful. Functions in most imperative languages can have side-effects (as can expressions in some languages, notably C). Furthermore, imperative combining structures, such as the `for` loop, depend heavily on statefulness. Finally, although some assignment statements resemble equation bindings, statements like `x = x + 1` do not map onto equation bindings at all.

3.4 Visual Basic offers inspiration for a spreadsheet-like functional programming environment.

In spite of some design flaws found by the walkthroughs, Microsoft Excel with Visual Basic shows the power of the spreadsheet model as an environment for describing computation and, in a limited fashion, interface constraints. The leverage provided by the spreadsheet (especially in the line-fitting example) allowed the Visual Basic programs to be much shorter and simpler than standalone programs could have been.

However, a comparison of the spreadsheet computational model of Excel and the imperative computational model of Visual Basic points to difficulties getting the two models to work well together. We will see in chapter 4 that the functional model is a much better match to the spreadsheet model, computationally speaking.

The walkthrough experience also suggests areas for improvement. As noted above, bidirectional constraints (formulae) would further simplify the temperature conversion example. A clean, coherent language model with more support for abstraction and a more consistent set of design choices would simplify the manipulation of data and interface objects. Finally, the power of automatic consistency maintenance in the spreadsheet, especially in contrast with the need to maintain consistency explicitly in Visual Basic, suggests that the spreadsheet model be expanded to include interface objects as well as data, rather than enhancing the spreadsheet model with a new and different model.

Chapter 4

Esquisse

Having examined monadic I/O in Haggis, and the combination of Excel with Visual Basic, we now turn to Esquisse. Esquisse has been designed and implemented to articulate a vision: that the cognitive costs of writing interactive programs can be much lower in a programming environment that combines functional programming languages and the spreadsheet model of interaction.

Section 1 discusses the benefits of combining spreadsheets and functional programming. Sections 2 and 3 describe Esquisse and its implementation. Section 4 provides two extended examples that illustrate the benefits of Esquisse, and chapter 5 discusses some of Esquisse's limitations. Section 6 evaluates Esquisse via the programming walkthrough, and analyzes the compatibility of spreadsheets and functional languages. Section 7 discusses some higher-level issues.

4.1 The strengths of functional programming and spreadsheets can be combined in a powerful, flexible system.

The functional and spreadsheet models have significant common ground: both lack control flow and use functions to specify one-way constraints (a functional program can be viewed as a set of constraints), and spreadsheet formula languages, like functional languages, are stateless and expression-based.

The functional and spreadsheet models also complement each other well. The functional language offers computational power and abstraction mechanisms like higher-order functions and lazy evaluation. The spreadsheet's powerful interaction and computational model demonstrates that complex computations can be carried out, and data can be freely manipulated, without requiring the user to specify control flow or input and output, or to explicitly maintain relationships among data items. Both environments support quick program construction.

Combining a functional language with a spreadsheet offers an opportunity to overcome the weaknesses of each individually. Spreadsheets lack modularity and abstractions; functional languages provide higher-order functions and lazy evaluation. Functional programming languages support interactivity with some difficulty; interaction is central to the spreadsheet model. Spreadsheets are clumsy at solving large, complex problems; functional languages provide computational power.

Esquisse, as a synthesis of these two approaches, presents significant advantages. Esquisse provides support for quick construction of computationally significant interactive programs. It expands the power of spreadsheets by extending the interaction model to include graphical representations and interaction, and it supports customization of interactive appearance and behavior. It provides a form of stateful interaction without the messy cognitive challenges required by existing functional programming approaches. In addition, Esquisse formalizes several important features of the spreadsheet model.

Esquisse is especially suited for constructing interactive models for experimentation. Two examples are given in section 4.4, page 67.

4.2 Esquisse is a spreadsheet-inspired programming environment built around a functional language.

Esquisse combines a functional programming language with an extension of the spreadsheet model to capitalize on the assets of both.

4.2.1 The design process for Esquisse centers around the programming walkthrough method.

Esquisse grew out of a prototype written as a first attempt to articulate this synthesis of functional programming and spreadsheet-style interaction [Blough 1996]. The design of Esquisse is the result of programming walkthroughs performed on that prototype and on the Haggis and Excel with Visual Basic systems.

4.2.2 Esquisse draws upon and extends the spreadsheet metaphor for storage, display, and interaction.

Esquisse is a direct outgrowth of the spreadsheet model: cells store and display values, cells are linked by formulae to other cells, cell values can be edited, and when a cell's value changes, cells whose formulae depend on its value are updated also. All of these features have been expanded: cell values, formulae, and their display and interaction have all been broadened. The expression-based, stateless formula language of the spreadsheet has been expanded into a full-fledged pure functional programming language. Esquisse uses the functional language Gofer, a variant of Haskell.

4.2.2.1 The fundamental unit of computation and interaction is the cell.

In Esquisse, as in a spreadsheet, cells store values, display them, and support interaction with them. Esquisse cells differ from their spreadsheet counterparts in two simple ways: they are not laid out in a grid, and each cell has a type which is determined by the type of value it contains.

Unlike spreadsheets, Esquisse "opens up" the mechanics of the cell display and interaction to the user.

Ignoring formulae for the moment, an Esquisse cell consists of a value and two functions.

```
data Cell a = Cell a (DisplayFunction a) (EventFunction a)
type DisplayFunction a = a -> CellPicture
type EventFunction a = Event -> a -> a
```

(The actual types are slightly more complicated; see section 4.3.1, page 63.)

4.2.2.2 A cell's display function and event function govern its interactive behavior.

The display function maps the cell value to a graphical representation (which can also be textual) of type `CellPicture` (based on the `Picture` type of [FPJ 1995]). More precisely, the display function produces **the only** graphical depiction of the cell. For example, the graphical depiction of the word "happy" would be `text "happy"`; the graphical depiction of a circle of radius 3 would be `circle 3`. Whenever the cell's value changes, the display function (with the cell value as an argument) is evaluated to produce a new depiction. Thus, the cell's value and depiction are always consistent.

The event function maps an event and the current cell value to a new cell value. For example, the standard text event function would map the event "the `h` key was pressed" and the value `""` to the value `"h"`. The event function is evaluated whenever an event (keypress, mouse click, drag, ...) occurs over the graphical depiction of the cell.

The display function and event function are dependent on each other and on the cell's value. First, all must share the same base type (denoted by `a` in the above definitions). Second, since all that the user has to interact with is the graphical depiction specified by the display function, the shape and extent of that depiction determine which events, spatially speaking, are reported to the cell.

4.2.2.3 A cell's formulae govern its computational behavior.

A cell's value can be changed not only by events that occur over its graphical depiction, but also by evaluation of its formulae.

A formula consists of a function and a collection of cells (the "source cells" for the function). When the value of one of the source cells changes, the function is applied to the values of the source cells, and the result is the new value of the cell. In a typical spreadsheet, the cell `F3` might contain the formula `=C3*D3`. In Esquisse, cell `f3`'s formula would have source cells `c3` and `d3`. The function could be written literally as `\ c d -> c*d`, but more succintly as `(*)`, which is how the multiplication operator (and in fact, any infix operator) is written as a standalone function in Haskell.

Implementation note: since functions can be cell values, each formula function is stored in a cell. Because a formula function cell is created as part of a formula, it is not displayed, and it can only be edited in the editor. The Esquisse implementation is discussed further in section 4.3 (page 63).

4.2.3 The Esquisse interface supports creating, editing, and interacting with cells.

Esquisse provides a direct-manipulation interface, shown in figure 4.1. The Modes panel (near the top of the leftmost column) is the control center of the interface. The modes work as follows:

- Select mode supports the selection of one or more cells. When cells are selected, the Operations panel is active.
- The Draw, Text, and Expression modes support the creation of a single cell. In Draw mode, the Drawing panel is available. In Text and Expression modes, the Fonts panel is available. The Colors panel is available in all three modes.
- Interact mode supports interaction with a cell via its event function. In a sense, Select mode and the cell creation modes are just support structure; Interact mode holds the core functionality of Esquisse. In Interact mode, none of the other panels (Operations, Drawing, Colors, Fonts) are available.

The Refresh, Load, Save, and Quit buttons are available at any time. The Ready indicator changes to Working while Esquisse is processing.

4.2.3.1 Cells are created by direct manipulation., and all interaction with a cell is governed by its event function.

Graphical cells are drawn on the screen. For instance, each bar in the temperature conversion example was created by clicking on the filled rectangle button, clicking on the appropriate color, and dragging out the rectangle. The rectangle cell's value is `((x,y),(w,h)) :: ((Int,Int),(Int,Int)` where `x` and `y` are the lower left corner, and `w` and `h` are the width and height respectively. (Esquisse's coordinate system is Cartesian, with the origin in the lower left corner of the work area.) Of course, the display function maps this value to a rectangle of width `w` and height `h` with its lower left corner at `(x,y)`.

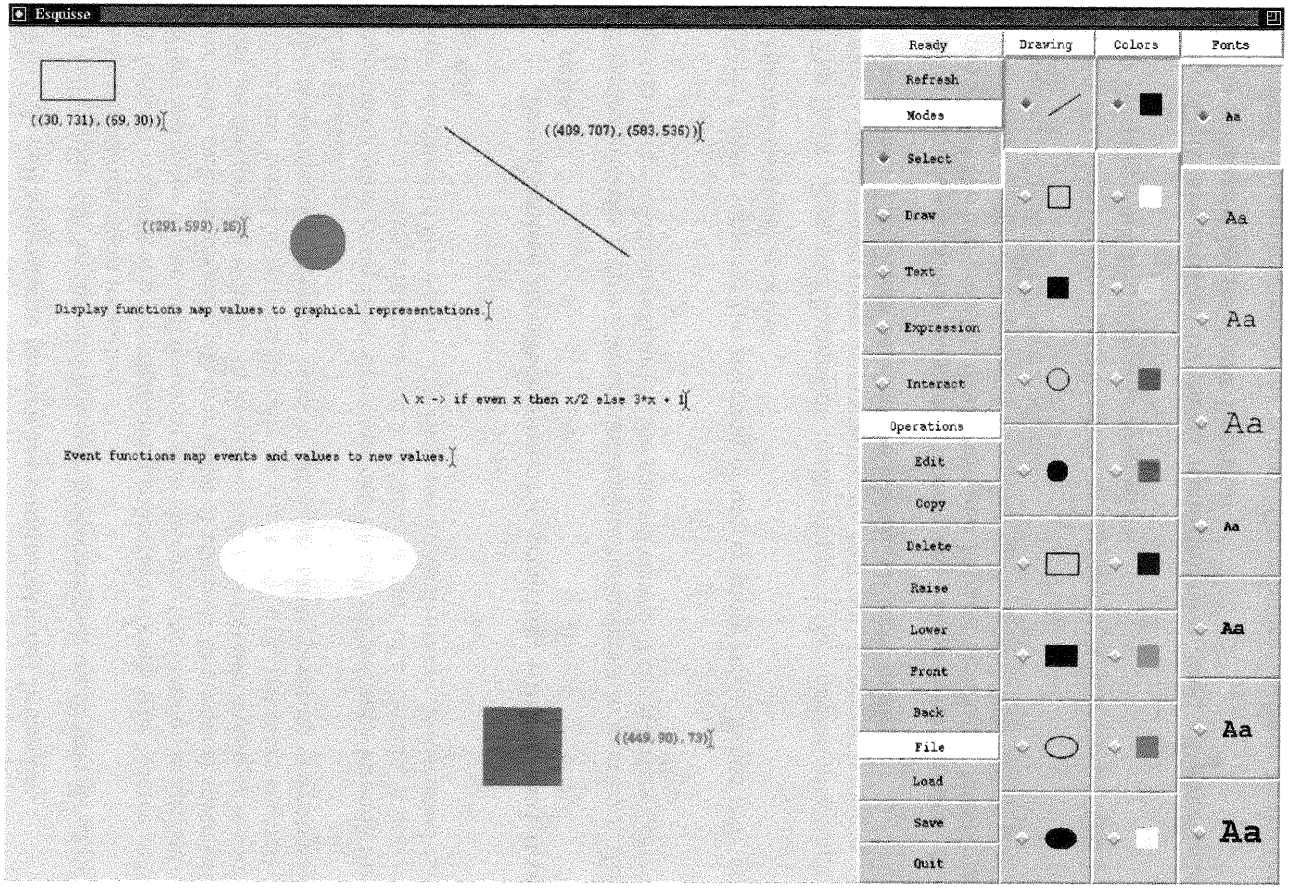


Figure 4.1: The Esquisse interface.

Once the graphical cell is created (drawn), the interface mode switches automatically to Interact. The default event function for these graphics cells implements dragging: the default event function applied to an event matching `MouseDrag _ dx dy` (where `dx` and `dy` are the x- and y-displacements) and a cell value `((x,y),(w,h))` will evaluate to the new cell value `((x+dx,y+dy),(w,h))`.

Text and expression cells are placed by clicking on the screen. The two number cells in the temperature conversion example were created by selecting the appropriate color and font, and clicking in the work area. Once the cell has been placed (it is displayed as a lone text cursor), the interface mode switches automatically to Interact. The default event function for text and expression cells supports text editing: for example, the default event function applied to an event matching `Key Down Key_Delete` maps a cell value to a the same value minus the character before the cursor. For more on the role of the cursor, see section 4.3.1.

Default interactions and custom interactions are treated identically. Custom interactions are simply customized event functions; no more, no less. The user can edit any cell's event function to create a custom interaction. For instance, in the temperature conversion example, the two bars have a custom interaction of resizing from one end of the bar, instead of having the default interaction of dragging. This is acheived by a simple change to the event function (see the Esquisse temperature conversion walkthrough appendix).

4.2.3.2 The Esquisse editor supports modification of a cell's value, display function, event function, and formulae.

After selecting a group of cells (in Select mode), the user can open editors for them by clicking the Edit button. The editor (shown in figure 4.2) displays the name, type, value, display function, and event function, along with any formulae the cell may have (see the Esquisse walkthrough appendices). (Esquisse does not support interaction with any cell in the work area while a cell has an open editor.)

Each field of the editor is in fact an expression cell, and editing is supported by the default text interaction function. When the editor is closed, the value of the cell in each field of the editor is extracted and the original cell is updated.

4.2.3.3 Expression cells are re-evaluated after each event.

Expression cells are in some sense the most "generic" cells in Esquisse: they can hold arbitrary values, and they are displayed textually. As a result, they are good medium for introducing Esquisse's approach to type inference and error handling.

Since Esquisse is a functional programming environment, Esquisse cells have types. However, editing an expression cell poses problems for type inference. Consider the function for converting Celsius to Fahrenheit:

```
\ c -> round ((9.0/5.0)*(fromInteger c)) + 32
```

with a type of `Int -> Int`, and suppose that the user is typing this function into an expression cell. For the first few keystrokes, the value cannot be parsed, and thus cannot be assigned a type. When the user has written:

```
\ c -> round
```

Gofer infers that it has type `a -> (Float -> Int)`. But when the user finishes typing the function, it has type `Int -> Int` — the type has changed simply in the course of entering the function!

Because this situation occurs frequently, Esquisse cells can change type when they are edited. This does not weaken the type system, because the types of the display function, event function, and formulae must all match the type of the value (formula updates with a type

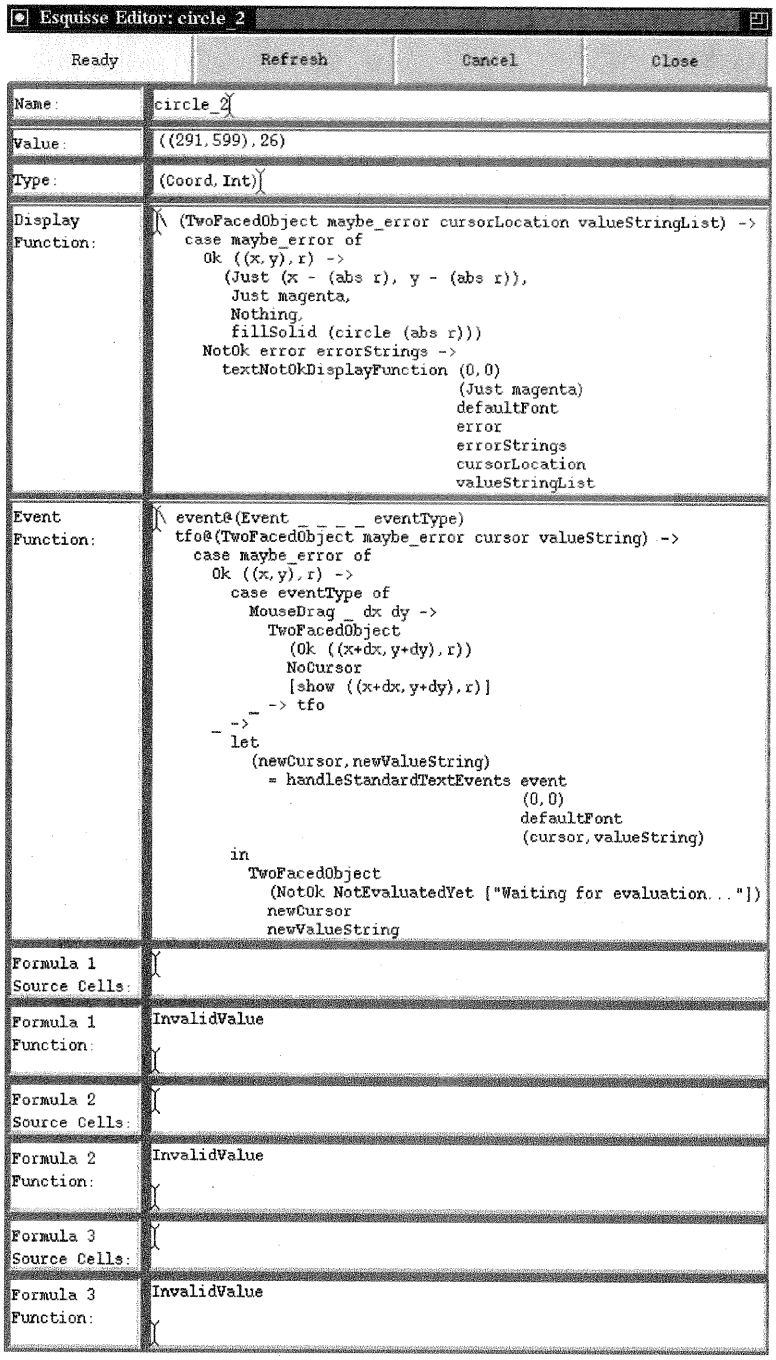


Figure 4.2: The Esquisse editor.

different from that of a cell's value will not change the type of that cell, but will cause a type error).

Error handling is also handled incrementally. If an expression cell's value cannot be computed, the error is displayed along with the value. The error message changes with continued editing, and when editing yields a valid value, the value is displayed without an error message.

4.3 The implementation of Esquisse cleanly separates the purely functional from the stateful.

The implementation of Esquisse, as well as its design, raises issues related to the implementation of programming environments and interpreters. Additionally, the particular implementation chosen for Esquisse helps demonstrate the purity of its functional model, and clearly specifies the boundary between the purely functional and the imperative parts of Esquisse.

4.3.1 Some additional types are required to support the needs of the cell framework.

The implementation of editing in Esquisse differs from that in a tradional interpreter. In an interpreter, expressions are parsed from textual to evaluable form, then bound or executed. The resulting expressions are printed out, but not all expressions have meaningful printable representations. Typing `\ x -> x + 3` into the Gofer interpreter yields `v121` — the essential structure of the function is not displayed.

Because Esquisse cells store editable values, a printable representation must be available. A cell containing `\ x -> x + 3` must always have (or be able to generate) the text string `"\ x -> x + 3"`, so that the user can edit it. Even if the function's display function is non-textual, the Esquisse editor will, and must, display the function textually.

4.3.1.1 `TwoFacedObjects` store texual and evaluated versions of the same value.

Because some values (such as functions) cannot be printed from their evaluated forms, Esquisse maintains a textual version of each value along with the value:

```
data TwoFacedObject a = TwoFacedObject a Cursor [String]
```

A `TwoFacedObject` stores both the value and the printable representation, along with a `Cursor`, which stores the position of the cursor in the printable representation:

```
data Cursor
 = NoCursor                  -- no cursor
 | OneLineCursor Int         -- cursor in one-line string
 | MultiLineCursor (Int,Int) -- cursor (row,col) in
                             -- multi-line string
```

The `\ x -> x + 3` function as a `TwoFacedObject` would be:

```
TwoFacedObject
  (\ x -> x + 3)
  (MultiLineCursor (1,6))
  ["\ x ->","  x + 3"]
```

The textual representation is formatted on two lines, with the cursor is at the end of the first line:

```
\ x ->
  x + 3
```

Keeping the value and the textual representations consistent cannot be done within Gofer, because setting the text from the value requires a function that can print out all values, and setting the value from the text requires `eval`. (If it could be done within Gofer, there would be no need to implement `TwoFacedObjects` in the first place.) Esquisse can and does maintain the consistency of `TwoFacedObjects`, except for those that are currently exposed to the user (see section 4.3.1.3 about how `TwoFacedObjects` could be put under the hood).

4.3.1.2 The algebraic type `MaybeError` supports spreadsheet-style error handling.

Error handling in Esquisse is modeled on error handling in spreadsheets. Because cells are persistent and editable, and because an error may only affect some cells, Esquisse, like spreadsheets, cannot simply terminate with an error message.

In the approach taken in Esquisse, a value of type `a` is actually stored as a value of type `MaybeError a`:

```
data MaybeError a = Ok a | NotOk Error [String]
```

Normal values are tagged by `Ok`: the lambda expression we have been using would be `(Ok (\ x -> x + 3))`. If we had omitted the initial `\`, writing instead `x -> x + 3`, the appropriate `MaybeError` value would be `(NotOk InvalidValue ["Syntax error"])`. The error values are:

```
data Error
 = InvalidValue
 | UnreadableString
 | NotEvaluatedYet
 | DisplayFunctionError
 | EventFunctionError
 | FormulaError
 | DivOError
 | EmptyCell
 | NoInstance
 | TypeError
 | UndefinedName
```

Notice that in the example above, erroneous `MaybeError` values do **not** store a printable representation of the erroneous value. To meet that need, Esquisse uses `TwoFacedObject` and `MaybeError` in combination, and the base type of a `Cell` storing values of type `a` is not `a`, but `TwoFacedObject (MaybeError a)`. Since display functions and event functions can be edited and can contain errors, they too must have this more complex type. As a result, the actual type `Cell` is:

```
type DisplayFunction a = TwoFacedObject (MaybeError a)
                       -> CellPicture
type EventFunction a = Event -> TwoFacedObject (MaybeError a)
                             -> TwoFacedObject (MaybeError a)

data Cell a = Cell (TwoFacedObject (MaybeError a))
                   (TwoFacedObject (MaybeError (DisplayFunction a)))
                   (TwoFacedObject (MaybeError (EventFunction a)))
```

Since formula functions are stored in their own cells, they too have this editing and error-handling support.

4.3.1.3 Hiding some of the cell internals would simplify display and event functions.

The `TwoFacedObjects` and `MaybeError` values show through to the programmer in the display and event functions for a cell (see the Esquisse walkthrough appendices for screen images). Our experience with Esquisse suggests that these implementation details could be more fully hidden.

The `MaybeError` type could be hidden entirely, at the cost of not allowing customizable error handling.

Esquisse provides default display and event function cases for error handling for cases in which a cell contains an error value. The display function case simply displays the error and the textual representation that, when evaluated, caused it. The event function case supports text editing of the erroneous text.

The added flexibility of being able to change the display and event functions to handle error values differently may not be worth the added clutter. If so, the `MaybeError` type and its display and event function cases could be pushed down into the implementation without trouble. (The errors associated with formula evaluation and updating are already inaccessible to the user.)

The `TwoFacedObject` type could be encapsulated behind a set of access and update functions.

Getting rid of `TwoFacedObjects` is less trivial. The salient point here is that, at various points, the user's functions (display functions, event functions, and formulae) need to access or update the evaluated or textual versions of the value's `TwoFacedObject` (and, of course, have the two stay consistent). Direct references to `TwoFacedObjects` could be replaced with functions named, say, `showValue`, `showText`, and `eval`, which could perform the necessary operations without exposing the `TwoFacedObjects`.

4.3.2 Esquisse is implemented as a front end to the Gofer interpreter.

Esquisse is not written as a monolithic system for three reasons. First, as a programming environment, Esquisse needs to be able to evaluate expressions, and pure functional languages do not support evaluation within the language. Second, available functional language interpreters like Gofer [Jones 1991] are designed specifically for evaluating expressions.

Third, and most importantly, implementing Esquisse as a front end to the Gofer interpreter provides a clean separation of the purely functional portion from the non-functional portion. This division helps us see how little non-functional "glue" is required to hold the functional pieces together. The front end communicates with Gofer via a pair of UNIX[1] pipes.

Referring to Esquisse as a "front end" to the Gofer interpreter is misleading, because in fact a substantial fraction of the mechanics of Esquisse are also written in Gofer. In fact, the only parts of Esquisse not written in Gofer are the non-functional ones: the interface and a portion of the cell mechanism. The non-Gofer parts of Esquisse are written in Haggis [FPJ 1995] and Concurrent Haskell [PJGF 1996].

The following portions of Esquisse are implemented in Gofer:

- the `Cell`, `TwoFacedObject`, `Cursor`, `MaybeError`, and `Error` datatypes
- functions that create the editor cells from the original cell
- the `CellPicture` datatype (a trimmed-down and slightly modified `Picture` datatype)
- the `Event` type (a modification of `DeviceEvent` from `Haggis`)
- basic cell types: `Expression`, `Text`, `Line`, `Circle`, `Square`, `Rectangle`, `Ellipse`

[1] UNIX is a trademark of AT&T Bell Laboratories.

The remainder of Esquisse is implemented in Haggis, specifically:

- the interface, including interface modes, drawing, and communication with I/O devices
- cell names, including storage of all cell names and the names of the currently selected cells
- cell updates
- formula bindings (for example, cell `d`'s value is the function in cell `f` applied to the value in cell `s`)
- propagation of formula updates
- detection of events, routing events to cells, rendering of cell pictures
- communication with Gofer (having Gofer evaluate expressions and bindings, parsing the results)

Most of these front-end items are obviously stateful: the inferface, event detection and routing, picture rendering, and storage of the list of all cells and the list of selected cells. The inclusion of formulae is perhaps less clear. Formulae cannot be part of the cell framework for two reasons: typing and propagation. Since propagation is one of the mechanisms for stateful update of cells, it cannot be implemented in the pure functional part of the cell framework.

Typing is more subtle. If formulae are included in cells, then they influence the type of the cell, much as the display function and event function do. The difference is that the types of the display and event functions are completely determined from the base type of the cell, while formulae can have any type that returns the base type of the cell. For example a cell of type `Cell Int` could have formulae of any of the following types:

```
Float -> Int
[a] -> Int
Int -> Int -> Int
```

and so on. Including formula types in the type of the cell would have two consequences: cells of the same base type that had different formulae would have different types, obscuring their fundamental commonality; and cell types would quickly become quite complicated. Hence, the solution in Esquisse is to put each formula in its own purely functional cell, but to connect the cells and update them (via formula propagation) from outside the pure functional framework.

The implentation handles a typical interaction as follows:

(1) An event (a click, a drag, a keypress) occurs in (the graphical representation on the screen of) a cell. Haggis has Gofer evaluate the cell event function on the event and the cell's value. The result is a new cell value. (Because Gofer "displays" the value to Haggis, the cell value cannot be an infinite structure; in fact, it cannot even be a moderately large finite structure. This is a consequence of the implementation, not a conceptual shortcoming.)

(2) Haggis has Gofer create a new cell, differing from the existing cell solely by having the new value. We call this "updating" the cell, although it is purely functional in nature. Cells are indexed by the implementation, so the name of the new cell is the name of the old cell with the index incremented.

(3) Haggis has Gofer evaluate the cell's display function applied to the cell value. The result is a picture, which Haggis displays.

(4) For each formula that depends on the value of the cell, Haggis looks up the formula, including the destination cell, the source cells, and the formula function cell. Haggis then has Gofer evaluate the value of the formula function cell (that is, the formula function) applied to the values of the source cells. Haggis checks if the result of this function application is the same as the value of the destination cell. If so, Haggis does nothing. (This is how infinite loops are avoided, although they can still occur if mutually recursive formulae do not have a fixed point — see section 4.5.2.2, page 81 for more details). If the result of applying the formula to the source cells differs from the value of the destination cell, Haggis "updates" the destination cell as described in step 2 and displays the destination cell's value as described in step 3.

(5) Step 4 is repeated for any formulae depending upon the destination cell. This process is repeated until all formulae have been evaluated at least once and formula cycles have reached fixed points.

Steps 3 and 4 can be performed concurrently, although the current implementation of Esquisse does not do this. The order of formula evaluation for the (possibly multiple) formulae in a given cell is explicitly not specified, because the cell framework is fundamentally declarative (for more detail, see section 4.5.2.1, page 80). Because of this, all of the formula evaluations in step 4 could be concurrent.

Although the order of formula evaluation for the formulae dependent upon a given cell is unspecified, the connectivity of the cells enforces a partial order that is breadth-first in nature: formulae that depend upon the cell in which the event occur are evaluated; their evaluation can trigger the evaluation of other formulae (which depend on cells that depend on formulae that depend on the initial cell), and so on.

4.4 Substantial examples show Esquisse payoffs.

Although the walkthroughs (see section 4.6.1, page 83) demonstrate the cognitive benefits of Esquisse, the larger examples described in this section show the payoffs of the functional and spreadsheet models together, and of Esquisse's interactive machinery.

4.4.1 The logistic map is a simple system with chaotic dynamics.

The logistic map is a simple dynamical system that exhibits deterministic chaos. The heart of the logistic map is the equation:

$$x_{n+1} = s \cdot x_n \cdot (1 - x_n)$$

Iterating this function on a starting value yields a sequence of values that follows one of several patterns, depending on the value of the control parameter s. For $s \in [0, 1]$, the sequence converges on the fixed point 0. For $s \in [1, 3]$, the sequence converges on a fixed point which increases from 0 to approximately 0.664801. (Zero remains a fixed point, but is unstable for $s > 1$.) At $s = 3$, the fixed point bifurcates into a limit cycle of length two. The next two bifurcations, at $s \approx 3.45$ and $s \approx 3.55$, produces limit cycles of length four and eight, respectively. The bifurcations continue to double until $s \approx 3.57$, at which point the cycle never repeats. For a more thorough explanation, see chapter 5 of Peak and Frame [PF 1995].

There is a graphical method for exploring this system. We start by plotting the logistic map and the line $y = x$ on the same plot. We then choose a starting point on the x-axis, and draw a vertical line up to the curve. Next, we draw a horizontal line to the $y = x$ line. We continue to move vertically to the curve, then horizontally to the line, until we reach a fixed point (if there is one) or feel we have seen enough. Figure 4.3 shows six iterations for $s = 3.5$, with a starting point of approximately 0.7.

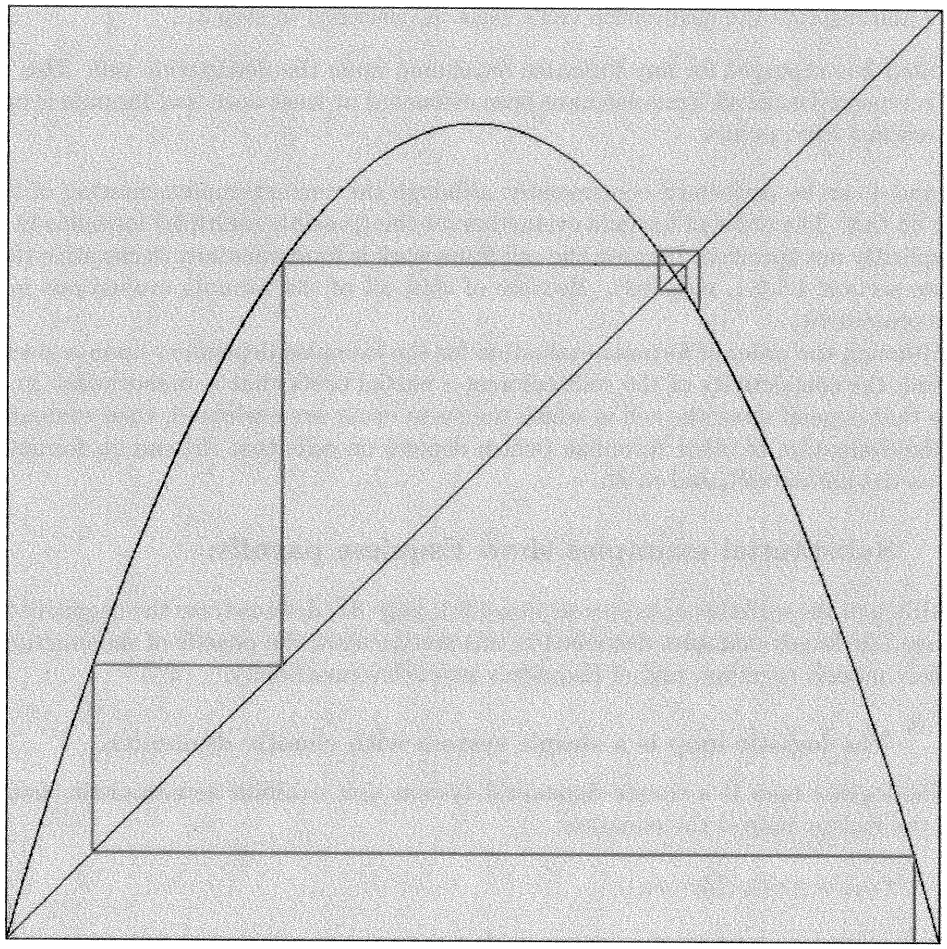


Figure 4.3: The graphical method for iterating the logistic map.

4.4.1.1 The logistic map example uses cells containing functions and formulae returning functions.

To implement this example in Esquisse, we use the Expression button to create a (textual) cell containing the logistic map:

```
\ s -> (\ x -> s*x*(1.0-x))
```

which we will refer to as `logistic`. We also create expression cells to hold the control parameter `s` and the number of iterations `n`. (Esquisse does not implement renaming cells from their default names, but we use mnemonic names here for clarity.)

The curve and the line, like the logistic map, can be stored as function values. The line is simply `id` (the identity function in Haskell), while the curve is `\ x -> s*x*(1.0-x)`, but for a specific value of s. We can do this by passing a specific value for s to the logistic function above. For example, `logisitic 2` evaluates to the function:

```
\ x -> 2*x*(1.0-x)
```

To ensure that the `curve` cell contains a logistic map function with the proper value of s, the `curve` cell contains a formula referring to the `logistic` cell and the `s` cell, and containing the function

```
\ lm s -> lm s
```

The formula takes the general logistic map and the value of `s`, and applies the logistic map to `s` to get a specific logistic map function. (We could even have used `($)` instead of the lambda expression.)

4.4.1.2 The logistic map example requires customized display and event functions.

Displaying the curve and line requires some customization, for Esquisse does not (currently) provide plotting cells. We start with a `square` cell of the desired size, which we draw with the interface. The display function shown in algorithm 12 is the default for the square

Algorithm 12 The default display function for `square` cells.

```
Ok ((x,y),s) -> (Just (x,y),
                 Just black,
                 Nothing,
                 square s)
```

(only the `Ok` case is shown; see section 4.3.1.1 on page 63 for details).

We modify the display function (algorithm 13) so that applies the `polyline` function to a list of $(x, f(x))$ pairs. The pairs are scaled from the unit interval to the width and height of the plot. Notice that the customized display function draws the plot with an origin of `(0,0)`, which means that the curve and line cells will be superimposed. Notice also that the type of the customized display function has been changed from

```
((Int,Int),Int) -> Picture
```

to

```
(Float -> Float) -> Picture
```

The event function must also be modified, so that its base type is `Float -> Float` rather than `((Int,Int),Int)`. As we shall see, the `curve` and `line` cells will not be interactive, so their event functions can simply ignore the event and return the cell value unchanged.

We also need a cell `x0` for the starting value x_0. We could simply type values into an expression cell (as with `s` and `n`), but the ability to click on the plot to specify a starting value allows us to interact directly with the displayed result, so we create a customized cell.

Since the depiction of a cell determines what events it senses, we create another `square` cell of the same size as the `curve` and `line` cells. Since it will only contain a (changing) x-coordinate, we "hard-wire" the display function to draw a square regardless of the value (algorithm 14); this has the added effect of visually framing the example. We customize the event function (algorithm 14), which initially supports dragging the square (algorithm 15), so that it ignores the existing value and maps the (screen-based) x-coordinate of any mouse click to a value in the interval $[0, 1]$. The customized event function also ignores drag events, so that the cell cannot be moved around inadverdently.

4.4.1.3 Functional tools come in handy for tracing the evolution of the map.

The only cell left to be created is the `trace` cell, which will display the evolution of the map via the graphical method described in on page 67. We decide that the `trace` cell will contain the list of points, and the display function (algorithm 17) will draw the polyline from those points. The `trace` cell overlaps with the `line`, `curve`, and `x0` cells, and, like the `line` and `curve` cells, its event function ignores all events, returning the cell value unchanged.

To link the trace to the logistic map and starting value, we add a formula to the `trace` cell (algorithm 18). The source cells for the formula are `logistic`, `s`, `n`, and `x0`. The formula thus guarantees that the trace will reflect the current values of the logistic map function, the control parameter s, the number of iterations `n` to trace, and the starting value x_0.

The `trace` formula makes heavy use of functional tools. In the `let` expression, the value of `l` is an infinite list of iterates of the logistic map on the starting value. The small, recursively defined `dup` function maps a list to a new list with each element appearing twice in a row. The last line of the formula performs most of the computation. The built-in function `zip` creates a list of pairs from two lists; in this case, the duplicated list of points is zipped with itself, but with the copy that is to become the y-coordinate "shifted" by preceding it with a zero. Evaluating the `zip` expression thus yields a sequence of alternating vertical and horizontal line segments that trace the iteration. Finally, `take` selects only the first `n` pairs of lines. The formula thus sets the cell value to the trace of the first `n` iterates.

In the completed example (figure 4.4), clicking anywhere over the plot generates a new trace starting at the x-coordinate of the click. In addition, the value of the control parameter $s$can be edited to change the height of the curve; the trace will redraw from the existing x_0, but with the new curve. Of course, the number of iterations (that is, the length of the trace) can be edited similarly. Even the map itself can be changed; for example, it could be replaced by the tent map

Algorithm 13 The customized display function for the `line` and `curve` cells.

```
Ok f -> (Just (0,0),
         Just black,
         Nothing,
         polyline
           [(x, (round.(700.0*))
                (f ((/700.0).fromInteger) x)))
            | x <- [0..700]])
```

Algorithm 14 Customized display function for the starting value cell.

```
Ok _ -> (Just (0,0),
         Just black,
         Nothing,
         square 700)
```

Algorithm 15 The default event function for square cells.

```
\ event@(Event _ _ _ _ eventType)
  tfo@(TwoFacedObject maybe_error cursor valueString) ->
    case maybe_error of
      Ok ((x,y),r) ->
        case eventType of
          MouseDrag _ dx dy ->
            TwoFacedObject (Ok ((x+dx,y+dy),r))
                           NoCursor
                           [show ((x+dx,y+dy),r)]
          _ -> tfo
      NotOk _ _ ->
      ...
```

Algorithm 16 The customized event function for the logistic map starting value.

```
\ event@(Event _ x _ _ eventType)
  tfo@(TwoFacedObject maybe_error
                      cursor
                      valueString) ->
    case maybe_error of
      Ok _ ->
        case eventType of
          MouseButton Down _ ->
            TwoFacedObject
              (Ok ((fromInteger x)/700.0))
              NoCursor
              [show ((fromInteger x)/700.0)]
          _ -> tfo
      NotOk ->
        ...
```

Algorithm 17 The display function for the `trace` cell.

```
Ok pairList ->
  (Just (0,0), Just red, Nothing,
   polyline
     $ map (\ (x,y) ->
              (round (700.0*x), round (700.0*y)))
           pairList)
```

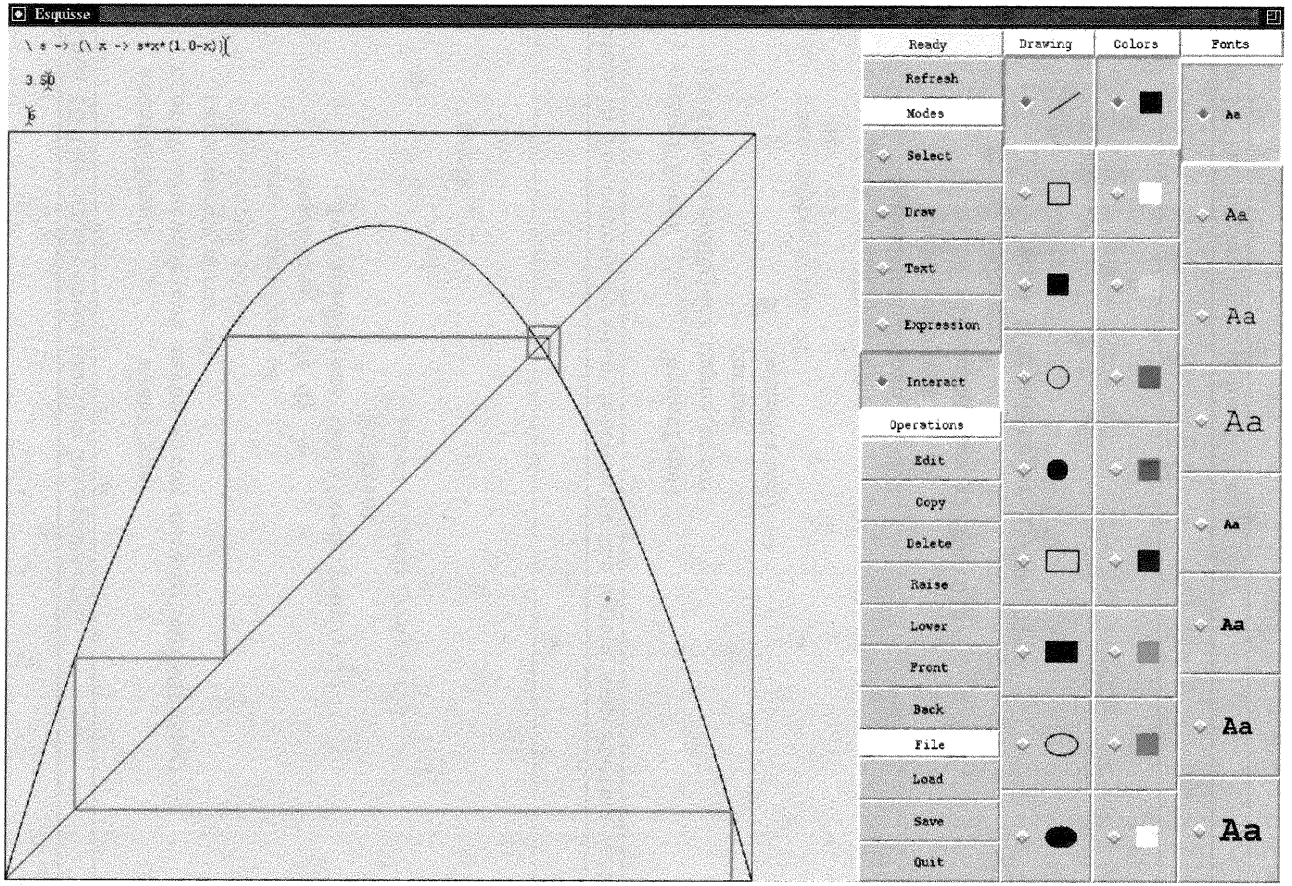


Figure 4.4: The logistic map example implemented in Esquisse.

$$x_{n+1} = s \cdot x_n \quad \text{if } x_n < 0.5$$
$$x_{n+1} = s \cdot (1 - x_n) \quad \text{otherwise}$$

or sine map

$$x_{n+1} = (\tfrac{s}{\pi}) \cdot sin(\pi \cdot x_n)$$

(two other maps exhibiting dynamical chaos) with a just a few keystrokes.

4.4.2 One-dimensional cellular automata (1DCAs) produce global patterns with local rules.

Cellular automata (CAs) are arrays of cells[2] that evolve according to uniform local rules. The simplest such automata are the one-dimensional cellular automata. A one-dimensional celluar automaton (1DCA) consists of a (typically infinite) row of cells. Each cell can have two states: 0 or 1. The state of a cell in the next generation is determined by its state and the states of its left and right neighbors. These three states together constitute the cell's "neighborhood." The rule specifies the mapping between these current states and the future state. For example, a cell might have state 1 in the next generation if and only if both of its neighbors have state 1 in this generation.

Wolfram [Wolfram 1994, p. 419] presents a notation for 1DCA rules. The above rule would be

111	110	101	100	011	010	001	000
1	0	1	0	0	0	0	0

in this notation. The top row, which is the same for all rules, depicts the current states of the left neighbor, the cell itself, and the right neighbor. The bottom row depicts the new state of the cell. Thus, the bottom row specifies that the cell will have state 1 if its both left and right neighbors currently have state 1.

This notation has the advantage that the bottom row can be viewed as a binary number, thus giving each rule a unique number between 0 and 255. The rule shown above is 128 + 32 = 160.

4.4.2.1 The 1DCA example uses higher-order functions to create cellular automaton rules.

In implementing a one-dimensional celluar automaton in Esquisse, one of the first questions to come up is how to implement rules. From a computational standpoint, rules should be functions from the current state of the neighborhood to the future state of the cell. From the standpoint of editing, the eight-digit standard notation makes the most sense.

[2] Cells in cellular automata are conceptually separate and distinct from the cells in Esquisse.

Algorithm 18 The formula for the `trace` cell.

```
\ lm s n x0 ->
  let
    l = iterate (lm s) x0
    dup [] = []
    dup (x:xs) = x:x:(dup xs)
  in
    take (2*n) $ zip (dup l) (0.0:(dup $ tail l))
```

Higher-order functions support both representations cleanly. The eight-digit notation is easily supported by a simple list of 0s and 1s (stored in an expression cell that we will call `ruleList`[3]). A higher-order function `ruleMaker` (algorithm 19) maps the rule list to a function

Algorithm 19 The value of the `ruleMaker` cell.

```
\ ruleList ->
    \ l c r -> ruleList!!(4*l+2*c+r)
```

that looks up the current neighborhood configuration and returns the corresponding next state. (List indexing is specified by `!!` in Haskell.)

4.4.2.2 The 1DCA example uses bidirectional formulae to maintain two rule representations.

The ability to also specify rules by number is a convenient feature. We can create an expression cell `ruleNumber` to store the number, and write two formulae to keep `ruleList` and `ruleNumber` consistent. Converting the list to the number is easy (algorithm 20); converting the

Algorithm 20 The formula in the `ruleNumber` cell.

```
ruleListToRuleNumber ruleList
  = sum $ zipWith (*) ruleList $ map (2^) [0..]
```

number back to the list is more complicated (algorithm 21). Together, the two formulae ensure that when one cell is edited, the other will change to maintain consistency.

4.4.2.3 The function that evolves the automaton can be placed in a cell.

Having set up the rule, we need a function to evolve the automaton forward in time. Algorithm 22 shows the value of the `evolve` cell. The `evolve` function defines `oneStep` (which maps a state to the next state), then uses it (and the `iterate` function) to create a list of successive states. The left and right neighbors of the cell are grouped with the cell by zipping three "copies" of the cells, each shifted a different amount. Concatenating an infinite list of zeros onto the end of the start state ensures that none of the automaton's cells will be accidentally dropped off the end of the list.

We also need a start state. We choose a single seed; that is, a 1 preceded and followed by 0s:

```
(take 250 $ repeat 0) ++ [1]
```

The trailing 0s will be added by `evolve`.

4.4.2.4 The evolution of the automaton is an infinite list of infinite lists.

Given a rule and a start state, `evolve` returns an infinite list of states, representing the entire future of the automaton. Each state is also an infinite list. In principle, we could simply store this list in a cell; due to the unfortunate implementation choice described in section 4.3.2, this is not possible. Of course, we do not want to display the entire list; we want to display some spatial extent over some number of generations. Instead of storing the list in a cell, we store a

[3] Actually, `ruleList` stores the rule in reverse order. This could be corrected by adding `reverse` to `ruleList`'s incoming and outgoing formulae.

Algorithm 21 The formula in the `ruleList` cell.

```
ruleNumberToRuleList digits n
  = ruleNumberToRuleList' n [] $ reverse
      $ take digits $ map (2^) [0..]
  where
    ruleNumberToRuleList' 0 result [] = result
    ruleNumberToRuleList' n result (p:ps)
      = if p <= n
        then ruleNumberToRuleList' (n-p) (1:result) ps
        else ruleNumberToRuleList' n     (0:result) ps
```

Algorithm 22 The value of the `evolve` cell.

```
\ rule start ->
    let
      oneStep rule cells
        = tail $ zipWith3 rule
                          (0:0:cells) (0:cells) cells
    in
      iterate (oneStep rule) (start ++ (repeat 0))
```

function that, when given a width and a number of generations to display, returns that portion of the automaton's evolution.

To create the plot, we draw a square cell. We could start with any type of cell. A graphics cell seems a reasonable prototype for a plot cell, and a square seems as good a choice as any. We change its value to be the function described above (algorithm 23).

Algorithm 23 The value of the `plot` cell.

```
\ width numGenerations ->
   (map (take width)) . (take numGenerations)
        $ evolve (ruleMaker ruleList) start
```

The `plot` cell needs for `evolve`, `ruleMaker`, `ruleList`, and `start` to be defined. The solution is to put a formula in the `plot` cell that defines the value (algorithm 24). When the

Algorithm 24 The formula in the `plot` cell.

```
\ evolve ruleMaker ruleList start ->
    \ width numGenerations ->
        (map (take width)) . (take numGenerations)
             $ evolve (ruleMaker ruleList) start
```

`plot` cell value is set by the formula, `evolve`, `ruleMaker`, `ruleList`, and `start` will be the values of the source cells of the formula.

The display function of the plot cell is shown in algorithm 25. The `plot` display function passes a width and a number of generations to the function value of the cell, converts the list of lists of 0s and 1s to coordinates, and displays the resulting points (using the `segments` function).

4.4.2.5 Esquisse supports the visibility of the automaton machinery.

The complete example is shown in figure 4.5. All of the parts of the computation relevant to cellular automata are visible and editable: the rule as a number, the rule as a list, the starting configuration, the `ruleMaker`, and the `evolve` function. The formulae and the display function for the `plot` cell are not visible: the conversion formulae between the number and list representations are not part of the automaton, and the formula and display function for the `plot` cell are mostly concerned with plotting, not the mechanics of the automaton.

The user can modify the rule number, rule list, or starting configuration, and the plot will automatically update. The rule shown is rule 90, which produces a Sierpinski gasket from a starting configuration consisting of a single 1 surrounded by 0s.

4.4.3 The logistic map and 1DCA examples show the power of Esquisse.

The benefits of the Esquisse approach can be grouped into three categories: benefits due to functional programming, benefits due to the spreadsheet mechanics, and benefits that arise from the combination of the two.

4.4.3.1 Higher-order functions and lazy evaluation contribute power.

Both the logistic map and the one-dimensional cellular automaton make heavy use of higher-order functions. The logistic map can be expressed as a function that takes a control parameter s and returns a mapping function. Rules for the cellular automaton can be generated by a higher-order function that takes a list denoting the rule, and returns the corresponding

Algorithm 25 The display function of the `plot` cell.

```
Ok f ->
  let
    listToXs list = listToXs' 0 [] list
    listToXs' _ result [] = result
    listToXs' n result (b:bs)
      = listToXs' (n+1) (if b == 0
                         then result
                         else n:result) bs
    listToYs list = listToYs' 0 [] list
    listToYs' _ result [] = result
    listToYs' n result (l:ls)
      = listToYs' (n+1) (if null l
                         then result
                         else (n,l):result) ls
    yxsToPoints (y,xs)
      = map (\ x -> ((x,300-y),(x+1,300-y))) xs
  in
    (Just (100,150),
     Just red,
     Nothing,
     segments
      . concat
      . (map yxsToPoints)
      . listToYs
      . (map listToXs)
      $ f 500 235)
```

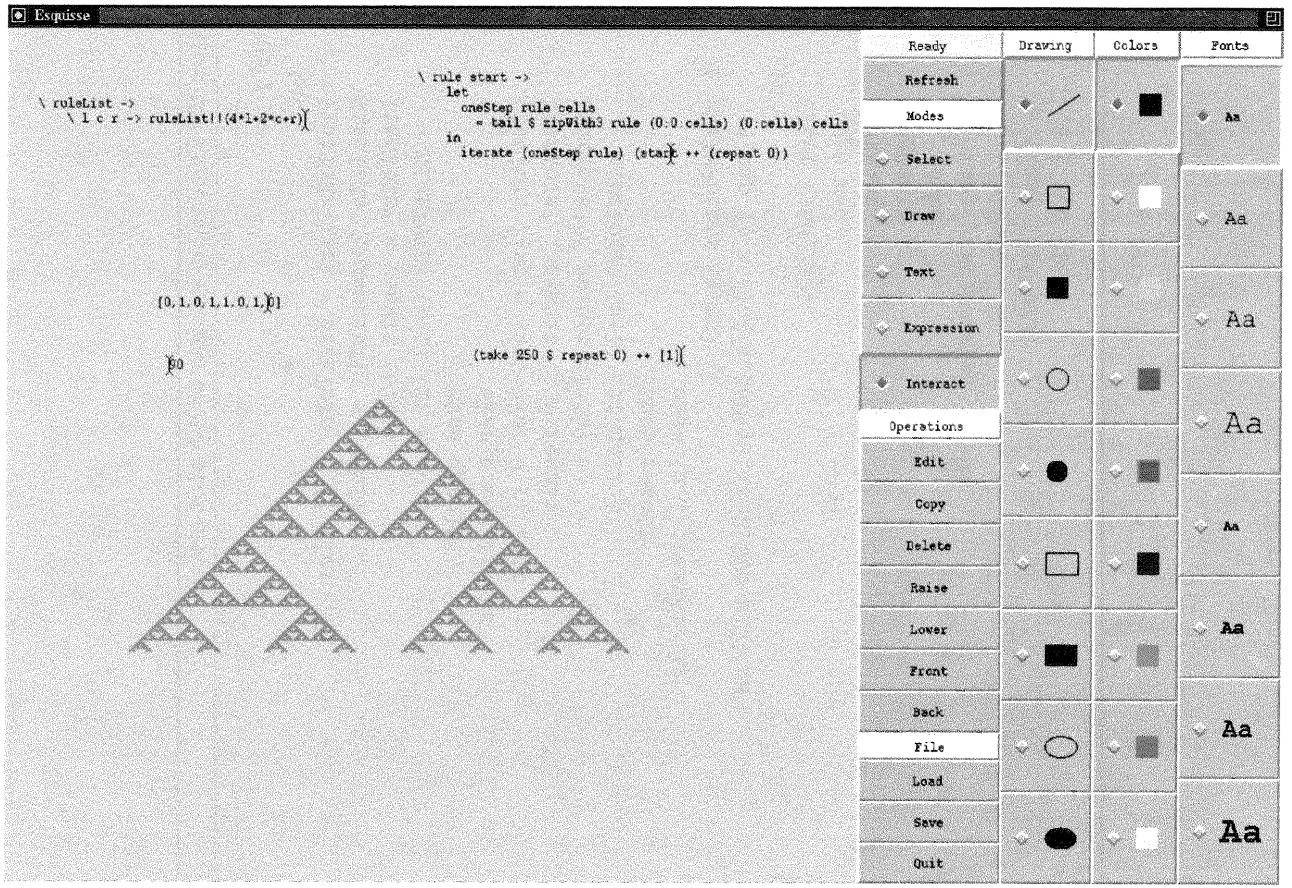


Figure 4.5: The one-dimensional cellular automaton in Esquisse (rule 90).

rule function. Both examples use `iterate` and variants of `zip` (both higher-order functions) to generate sequences of results.

In both examples, the sequences of results are infinite. In the cellular automaton example, the individual states are also infinite. Consequently, the computations are quite general: editing the `plot` cell (in the cellular automaton) or the `trace` cell (in the logistic map) is sufficient to change how much of the result is computed. Both cells apply transformations to the results in the course of plotting them, and these transformations can be applied to infinite lists as easily as finite data.

The presence of high-level operations such as `map`, `zip`, `zipWith3`, and `iterate` provide for quick construction of the examples.

4.4.3.2 Visible state, consistency maintenance, and built-in interactivity provide accessibility.

Interactivity is central to both examples. In the cellular automaton, the power comes from the ability to explore the space of one-dimensional cellular automata by modifying the rule and the start state. In addition, the `evolve` and `ruleMaker` functions are available for reference, or even for modification. In the logistic map example, the control parameter, the number of iterations, and even the logistic map itself can be edited. Furthermore, clicking in the plot "edits" the starting value for the iterations. No special work is required (no I/O needs to be performed) to make the expressions cells visible and editable, and the starting value cell is customized with little difficulty.

The formulae in the cellular automaton example maintain the consistency of the two rule representations, and ensure that the plot is consistent with the rule and start state. In the logistic map example, the `curve` and `trace` formulae keep them consistent with the logistic map and control parameter; and, in the case of the trace, the starting value and the number of iterations. The formulae allow relationships to be specified once by the programmer, and maintained by Esquisse.

4.4.3.3 Esquisse supports functions in cells, customizable display and interaction, and interactive graphics.

A provocative benefit of the spreadsheet-functional language combination is that functions can be cell values. As a result, computation is not doomed to be hidden in formulae, but can be placed in cell values. The logistic map, the `ruleMaker`, and the `evolve` function, in addition to being higher-order functions, are also cell values. As cell values, they can be arguments to and return values from formulae (which themselves are thus higher-order functions). Also, as cell values, they are visible and editable.

Esquisse extends the spreadsheet model by allowing customization of display and interactivity. The `plot` cell in the cellular automaton example and the `curve`, `line`, `x0`, and `trace` cells in the logistic map example are all (radical) customizations of existing cells. Although cells like curve, line, and plot (slightly modified) are candidates for "standard" cells, the `trace` and `x0` cells are the sort created for specialized use. Programming is for situations in which the solution to a problem does not exist yet; customization of appearance and behavior supports that flexibility.

Esquisse also extends the spreadsheet model by supporting graphical representations of, and interactions with, cell values. The `trace` and `x0` cells just mentioned together implement an unusual and powerful graphical interaction. The line-fitting and temperature conversion examples support interaction with cell values in a more direct way.

4.5 The design and implementation of Esquisse reveal some issues and limitations.

The process of designing and implementing Esquisse involved numerous design issues, some of whose consequences are discussed here.

4.5.1 Animation and simulations require minimal additional features.

4.5.1.1 Clock cells would support animation.

Clock cells are perhaps the simplest enhancement for adding animation. A cell whose formula references the clock cell can then update itself each time the clock changes, and thus can (for example) move around or change in some other animated fashion.

Esquisse can support animation in an extremely primitive fashion via an imitation clock cell. The user can create a counter cell that increments each time it is clicked. While the user clicks on this cell repeatedly, it behaves as the clock cell in the previous paragraph.

An important design question is how compatible the clock is with the largely functional model of Esquisse. The clock could simply be an add-on, but a principled integration is preferable.

4.5.1.2 Access to previous cell values would support simulations.

Esquisse can support certain simulations; in fact, the logistic map and the one-dimensional cellular automaton are simulations that work well in Esquisse. Broader classes of simulations, such as numerical integration and interactive simulation games, support interaction during the simulation. Such simulations can be readily implemented with a small extension to Esquisse: a language construct for accessing the previous value of a cell (or, in some cases, more than one previous value). This is simpler than it might seem, because all of the values of all cells in Esquisse are persistent. Esquisse cell values are purely functional, and all of them are available via the Gofer interpreter. The only necessary addition is a language construct that can be translated to reference the previous cell (or cells).

4.5.2 In unusual cases, formulae can be nondeterministic, or can cycle endlessly.

4.5.2.1 The unspecified ordering of formulae can affect some formula combinations.

Since Esquisse provides formulae for consistency maintenance, rather than stateful update, the order of evaluation for a given cell's formulae is deliberately left unspecified. For many applications, the order of evaluation does not affect the final result. Consider the temperature conversion example, with a Kelvin thermometer added. All three thermometers are storing a single quantity, the temperature, and each formula is a function of exactly one variable, so no indeterminacies are possible.

Now consider a three-cell implementation of the ideal gas law $PV = nRT$, with cells for pressure, volume, and temperature. Each cell has a single formula of two variables: the pressure cell's formula is $P = \frac{nRT}{V}$, the volume cell's is $V = \frac{nRT}{P}$, and the temperature cell's is $T = \frac{PV}{nR}$. Now suppose the value of the pressure cell changes. Esquisse will do one of two things: update the temperature, or update the volume – the user cannot choose. Modifying the order in which the formulae are listed in each cell is no help; each cell has only one formula. Worse, in the actual physical situation described by the ideal gas law, the temperature and volume may both change, and Esquisse does not support this outcome at all.

Specifying constraints as separate entities, coupled with some of the techniques used in constraint programming, might yield a more satisfactory solution.

4.5.2.2 Formula cycles that lack fixed points will "run away."

Esquisse handles circular formula references by finding a fixed point, as described in section 4.3.2, page 65. This works well when a formula cycle has a fixed point (as in the temperature conversion example, or the rule number to rule list conversion in the one-dimensional cellular automaton. Requiring a fixed point makes sense for consistency maintenance: two cells a and b with formulae $a = b + 1$ and $b = a + 1$ cannot be made consistent.

Floating-point numbers bring additional complications. In the temperature conversion example, the temperatures are stored as integers, not floating-point numbers, because the error in floating point computation can produce either cycles that evaluate indefinitely, or runaway evaluation due to accumulated error. Redefining equality on floating-point numbers is not a good solution, because the acceptable error range varies. As with formula indeterminacy, more sophisticated constraint mechanisms are the most promising avenue.

4.5.3 Several factors restrict cell generality.

4.5.3.1 Cells can only be created via the interface, not by formulae or other cells.

Esquisse provides stateful interactivity in combination with a functional language by keeping (the stateful parts of) the cell framework "out of reach" of the functional language. Hence, cells are stateful containers for functional computation. Cells can be created with the interface, but not from within the functional language.

4.5.3.2 Creating groups of cells is not supported.

Also, groups of cells cannot be modularized or abstracted to a single cell. Consider the temperature conversion example. A thermometer with both a bar and a numeric display (with either modifiable) is an obvious candidate for a container cell. Creating a composite cell brings up questions of what the value, display function, and event function are for that cell. Of course, container cells could have a different structure; the issue then becomes whether the benefits of container cells warrant complicating Esquisse further.

Although the partitioning of Esquisse into stateful and stateless is central to its cognitive benefits (section 4.6), several complications ensue. First, the interface and the language have different functionality, which hinders automating an interactive process, and, conversely, unpacking a computational process to make it more interactive. Second, since cell values are values in a functional language, cells could be created by the language only by being the values of other cells. In other words, cells would need to be first-class objects

4.5.3.3 First-class cells present typing and complexity problems.

Having cells as first-class objects presents two difficulties: typing and complexity. The typing issue is closely related to the typing problems surrounding formulae (see section 4.3.2, page 65). Putting cells within cells (even without their formulae) can quickly result in cells with quite complex type signatures. (The purely functional `Cell` implementation has no difficulty storing and inferring the types of arbitrary cell nestings; the problem is that if typing is to be useful, type signatures need to be relatively concise.)

The other difficulty lies in managing the relationships between nested cells: connections between cells at the same level, connections between cells and their container cells, connections (if permitted) between cells in different containers. (The issue of connections also relates to scoping.) Further research may uncover a principled solution.

4.5.3.4 Declarations cannot be put in cells.

Another difficulty arises when new data types are needed. Haskell provides a powerful mechanism for creating arbitrary data structures, but even first-class cells cannot hold data declarations. Even if a cell did contain a data declaration, other cells would have no access to it. Modules address this issue in Haskell; perhaps an useful analogy between modules and cells (or container cells) can be found.

4.5.4 Two design issues arise concerning the implementation of interaction with cells.

4.5.4.1 No computational mechanism is provided to deal with overlapping cell depictions.

In the line-fitting example, the fitted line, the points, and the clipping rectangle cells are all superimposed. In the logistic map example, the curve, line, trace, and starting value cells are superimposed. The benefits are obvious: several components of a visual display (a plot in both cases) can be treated independently, but displayed together. In both examples, some cells (the points, the starting value) are chiefly for interaction, some cells (the fitted line, the trace) display the results of the interaction, and some cells (the clipping rectangle, the curve and the line) are basically graphical constants. When cells are superimposed, only the top cell can receive events. The user can change the stacking order via the Esquisse interface, but a more principled approach to stacking or superimposing cells might eliminate this need.

4.5.4.2 Cell display and event functions and formulae are reevaluated after each keystroke.

Textual editing of cell values presents an interesting design choice. In Esquisse, when an event occurs over a cell, the event and display functions are evaluated, followed by formulae that reference that cell. For graphical (non-textual) cells, propagating formulae after each event make sense, because each event is individually meaningful. For textual cells, especially expression cells, individual events (such as deleting a character) are meaningful only in the context of the larger sequence of events (changing a numeric value). In particular, the intermediate states of the textual cell are almost certainly meaningless. Propagating their values via formulae will result in propagating errors, not meaningful values. In contrast, the event and display functions do need to be updated, so that the value changes and the user sees the change. An edit mode for cells, analogous to that provided by spreadsheets, would address this difficulty; the issue remaining is how to integrate such a mode into the cell framework.

4.5.5 Embedding the functional type inference system in Esquisse has two drawbacks.

Esquisse uses the type inference system of Gofer somewhat naively; both of the following issues could be mitigated by a type inference system tailored for Esquisse.

4.5.5.1 Using `show` in display functions risks unresolved type overloading.

As discussed in section 4.3.1.1, page 63, `TwoFacedObjects` maintain textual and evaluated versions of each value. When a value changes due to a formula update, the textual version of the value must be updated. For many, but not all, types, the overloaded function `show` will evaluate to a readable string that represents the value. However, `show` must be able to infer the type of its argument from the surrounding context, and sometimes the context does not resolve the issue. This problem also occurs in traditional functional programs, so it is not unique to Esquisse, but it occurs more often in Esquisse because the parts of a cell are evaluated separately

before evaluated as a unit. A better solution is desirable because unresolved overloading can be confusing, and because other information is available from other parts of the cell.

4.5.5.2 The editing process can result in premature type specialization.

Another consequence of type inference and the re-evaluation of textual cells after each event is premature type specialization. In the line-fitting example, the cell containing all of the points has a formula:

```
\ c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 ->
   [c1,c2,c3,c4,c5,c6,c7,c8,c9,c10]
```

This function has the very general type:

```
a -> a -> a -> a -> a -> a -> a -> a -> a -> a -> [a]
```

but a simple typographical error, such as `10` for `c10`, can cause the type inferencer to specialize the type `a` to (in this case) type `Int` . A more sophisticated handling of type inference is called for.

4.6 Esquisse compares favorably with existing programming systems.

4.6.1 Esquisse successfully addresses some of the cognitive challenges to programming.

The programming walkthroughs show that Esquisse's features work together to support program construction. Some of these features are essentially spreadsheet features, while others significantly extend spreadsheet functionality.

As with the other walkthroughs in this dissertation, the programming walkthroughs of the temperature conversion (Appendix E) and line-fitting (Appendix F) examples record the knowledge and actions a hypothetical user would need in order to write these two programs in Esquisse. The discussion in this section refers to steps in the temperature conversion walkthrough by references of the form "(TC step 23)," and steps in the line-fitting walkthrough by references of the form "(LF step 22)." Screen images of the final line-fitting and temperature conversion examples are shown in figure 4.6.

4.6.1.1 Esquisse shares many features with spreadsheets.

Esquisse stores data in a visible, manipulable representation. In the temperature conversion example, the Fahrenheit and Celsius temperatures are stored in the heights of the rectangles and the in the numeric expression cells. All the data in the temperature conversion example are interactive. In the line-fitting example, the position of each point stores its (x,y) pair, and the color of each point stores its status. The fitted line (LF step 38) is stored as a line (a pair of (x,y) pairs), the clipping rectangle (LF step 31) is stored as a square, and the correlation is stored as a floating-point number (LF step 43), and is displayed textually. In the line-fitting example, the points are manipulable (LF step 15). The other items are also manipulable, but manipulating them is nonsensical in the context of the example. Furthermore, they can be made non-interactive by replacing their event functions with `\ _ v -> v`, the event function that simply returns the value unchanged.

Computation takes place on the visible data, not in a separate "inner world" of computation. The formula functions in the temperature conversion example (TC steps 29, 32) are applied to the values of the numeric expression cells. The fitted line and correlation function (LF steps 34, 35) is applied to the list of points, which is computed (by a trivial function) from the points themselves (LF steps 23, 24). In both of these cases, all of the function arguments

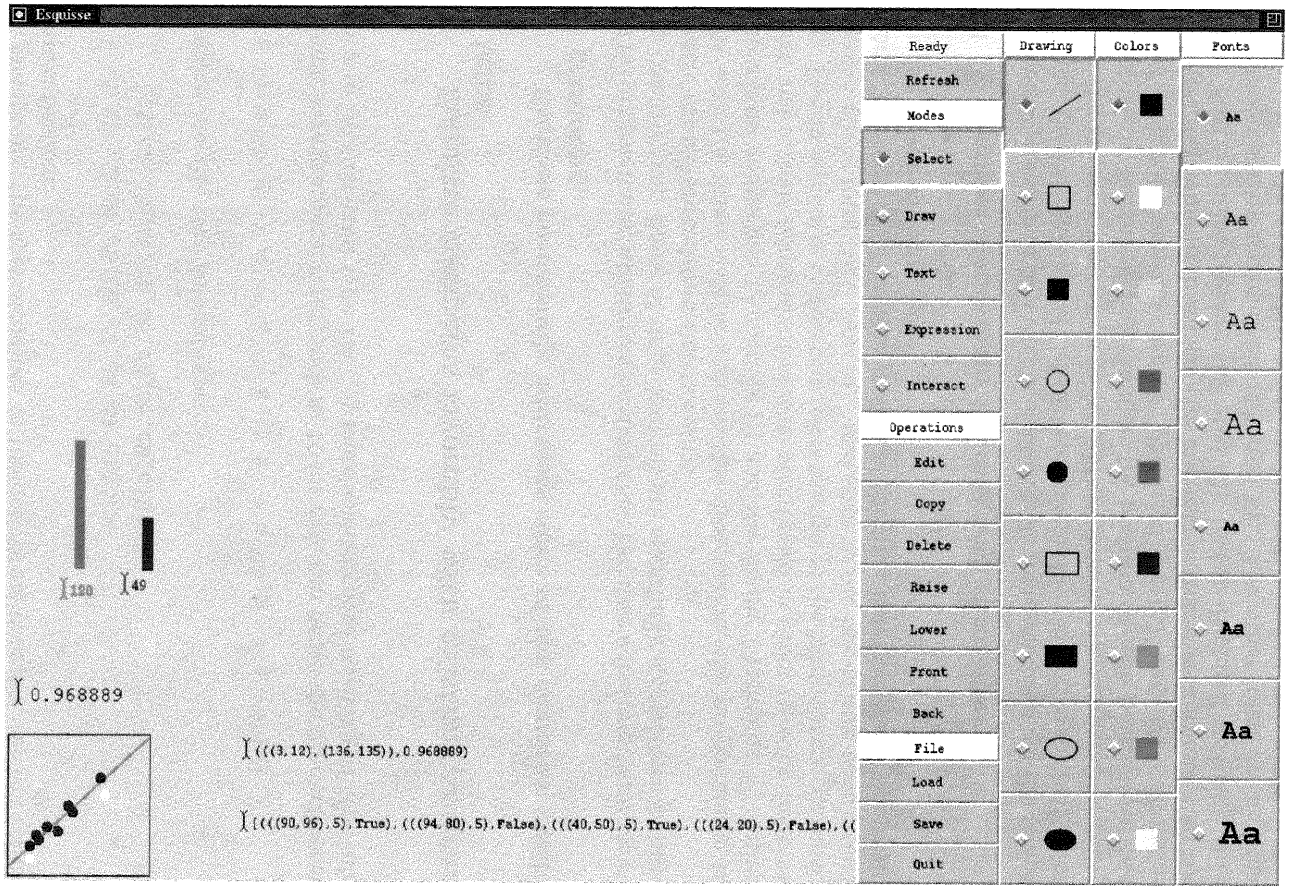


Figure 4.6: The line-fitting and temperature conversion examples in Esquisse.

are visible, manipulable data items; none of them required translation from the world of the user to the world of computation.

Although the data are continuously visible, Esquisse formulae, like their spreadsheet counterparts, are not. This has the advantage of not cluttering the display, but the acute disadvantage of hiding the computational machinery that is the heart of the program. This contributes to difficulty of debugging and redesign [LO 1987]. A solution available in Esquisse, but not in spreadsheets, is to put the formula functions in cells, and simply call the functions from the formulae. In the line-fitting example, the user could have put the `fittedLineAndCorrelation` formula in a cell (named `expression_20`, say). The cell holding the resulting (line, correlation) pair (named `expression_13` in the walkthrough) could have source cells `expression_20`, `expression_11`, and `square_14`, and the formula function:

```
\ fittedLineAndCorrelationFunction listOfPoints ((x,y),s) ->

    fittedLineAndCorrelationFunction
            (map (fst.fst)
                 (filter snd listOfPoints))
            ((x,y),(x+s,y+s))
```

That is, the function could be passed in like any other parameter. Taking this a step further, the user could extract the body of the lambda expression and place it in a cell, leaving a function like `\ f l s -> f l s` as the formula function. Although somewhat ad hoc, this does make essentially all of the formula visible. Furthermore, it allows the user to control which formulae she or he sees.

Interaction with the data is supported by the interface (by the cells) and, at most, needs to be customized by the user. The numeric expression cells in the temperature conversion example (TC steps 4, 5, 9, 10) support editing with no programming whatsoever. The circles in the line-fitting example (LF steps 5–17) required significant editing, but they initially had dragging behavior, as did the rectangles in the temperature-conversion example. The rectangles in the temperature conversion example required only a small modification to become draggable bars (TC steps 15–17).

Formulae provide automatic consistency maintenance. The temperature conversion formulae (TC steps 29, 32) keep the heights of the rectangles and the values in the number cells consistent with each other. The fitted line and correlation formula (LF step 35) keeps the fitted line and correlation coefficient consistent with the points and the clipping rectangle.

Because consistency is maintained automatically and the formulae are pure functions, there is no flow of control. In a sense, this is unsurprising, because functional languages are stateless. It is significant, however, that the line-fitting example responds to events and recomputes results automatically, and the temperature conversion program resolves cyclic formulae to a fixed point, all without flow of control.

Changes to the data produce visible results immediately, without explicit re-execution of the "program." Clicking a point, dragging a temperature bar, or editing a numeric expression cell produces immediate results, both in the affected cell and in the cells linked to it by formulae. No explicit updates need to be (or even can be) programmed.

Even the smallest fragments of a program (including the initial cells themselves) "work" on their own, and thus programs can be constructed incrementally. In Esquisse, interaction can be tested after almost every change, with no additional effort. As soon as the user creates a numeric expression cell, she or he can edit it. The same goes for graphic cells and dragging. After customizing a temperature rectangle, the user can immediately test the customization with no additional effort. (The customized point in the line-fitting example is analogous.) After linking one temperature rectangle to its numeric counterpart, the user can test that they are linked properly. Finally, after linking the two numeric cells together, the final version can be tested. This is a far cry from writing test programs, and a significant improvement to evaluating

expressions repeatedly at a prompt.

4.6.1.2 Esquisse also provides features that significantly extend spreadsheets.

Cells are not confined to a grid but can be positioned arbitrarily. This allows the free formatting of the temperature conversion example, and it is necessary in the line-fitting example, where the points are cells whose locations are determined by their values. In fairness, the lack of a grid has two drawbacks: cells must be created explicitly, and they obviously cannot be referred to by row and column.

Cells can contain arbitrary data values, including functions and (in principle) infinite data structures. This aids modularity and abstraction, as a function can be stored in a single cell and be an argument to many formula functions, rather than be copied into each of them. The benefits of infinite data structures for modularization are discussed in Chapter 1; storing them in Esquisse cells brings those advantages into the cell framework. Esquisse does not, however, currently provide another potentially useful abstraction – the facility to group cells together to form composite cells.

Cells contain both values and (possibly multiple) formulae. This makes bidirectional constraints (implemented by cyclic, or mutually recursive, formulae) possible, such as (all of) the formulae in the temperature conversion example. Multiple formulae support bidirectional constraints for cell groups of three or more: in the temperature conversion example, the Fahrenheit number cell has a formula that depended on the Fahrenheit rectangle, and another formula that depended on the Celsius number cell.

Cells can not only be created by typing (TC steps 4, 5, 9, 10) but also by drawing (TC steps 1–10), and all graphics objects are also cells. Drawing a rectangle or a circle creates not just a depiction, but a interactive component that can be linked to other components with formulae (TC step 11, LF step 5).

The interactivity provided by Esquisse is not fixed, but can be tailored to a wide range of interactions. Both the temperature conversion and line-fitting examples made use of this feature. This flexibility of Esquisse frees the user from dependency upon built-in functionality.

Custom interactions need not be written from scratch: the cells provided by Esquisse can be used as examples. The temperature conversion example customizes the rectangle cell (TC steps 12–17), and the line-fitting example customizes the circle cell (LF steps 5–17). Both examples use several non-customized cells "off the shelf" — we would expect this to be even more true with a larger selection of default cell types.

Customization of a cell consists of editing at most three relatively simple items: the value, the display function, and the event function. The value, display function, and event function can each be edited in isolation, provided their types do not change; for instance, the temperature conversion example involved only a simple event function customization. If any one of them changes type, however, all three must be edited. The customization of the circle cell in the line-fitting example falls into this category: the value was modified to add the (Boolean) status, the display function was modified to display the status, and the event function was modified to toggle the status; the modifications also ensured that the three of them had compatible types.

4.6.2 Spreadsheets and functional languages have compatible programming models.

In chapter 3, we examined the conflict between the spreadsheet and imperative programming models. A similar analysis of the spreadsheet programming model and the functional programming model shows far greater compatibility.

4.6.2.1 Computations in a spreadsheet and in a functional language map closely on to one another.

Consider again the spreadsheet implementation of the correlation function (figures 3.3 and 3.4), and the functional implementation shown in algorithm 26. The correlation is specified

Algorithm 26 Haskell implementation of the correlation function with more intermediate values.

```
correlation :: [(Int,Int)] -> Float

correlation theCoordinateList = sxy / sqrt (sxx * syy)
  where
    sxx = sumX2 - (square sumX) / n
    syy = sumY2 - (square sumY) / n
    sxy = sumXY - sumX * sumY / n
    sumX2 = sum (map square xs)
    sumY2 = sum (map square ys)
    sumXY = sum (zipWith (*) xs ys)
    sumX = sum xs
    sumY = sum ys
    square n = n * n
    xs = map fromIntegral xList
    ys = map fromIntegral yList
    (xList, yList) = unzip theCoordinateList
    n = fromIntegral $ length theCoordinateList
```

in term of S_{xx}, S_{yy}, and S_{xy}, which are in turn computed from $\sum x^2$ (`sumX2`), $\sum x$ (`sumX`), $\sum y^2$ (`sumY2`), $\sum y$ (`sumY`), and $\sum xy$ (`sumXY`) . The lists of x- and y-coordinates are extracted from the list of pairs with `unzip`, and `fromIntegral` enables the integer coordinates to be converted to floating-point numbers.

As in the Visual Basic implementation in chapter 3, the equations for S_{xx}, S_{yy}, S_{xy}, and the correlation map nicely onto rows 20 and 22–24 of the spreadsheet implementation. In addition, the equations for `sumX` and `sumY` are identical to the formulae in B17 and C17 of the spreadsheet implementation. The relationship between the equations for `sumX2`, `sumY2`, and `sumXY` and the spreadsheet formulae for $\sum x^2$, $\sum y^2$, and $\sum xy$ is only slightly more complicated: the `sum` expression in each equation corresponds to the formula in E17, F17, or G17 (as appropriate), and the `map square` (or `zipWith (*)`, in the case of $\sum xy$) expression corresponds to the formulae in column E, F, or G (as appropriate), rows 5–14.

The analogy between `map` (or `zipWith`) and columns of identical formulae is quite striking. Both techniques take a sequence (or more than one sequence) of values and create a new sequence of values, each of which is a function of the corresponding value(s) in the original sequence(s). Furthermore, although there is a spreadsheet formula for each value, spreadsheet users enter the formula once, then drag to fill the rest of the column with that formula. Thus entering the formula corresponds to specifying the function to be mapped, and filling the column corresponds to specifying `map` or `zipWith`.

The functional implementation has a few additional ingredients not found in the spreadsheet. As in the Visual Basic implementation, the (x, y) pairs must be separated, and the number of points calculated. Unique to the functional implementation is the need to convert the integers to floating-point numbers; both the spreadsheet and Visual Basic implementations use silent coercion.

4.6.2.2 Translating a computation from spreadsheet formulae to a function definition (or vice-versa) is straightforward.

Converting the Haskell `correlation` function into a group of spreadsheet formulae is fairly direct. Simple equations, such as those for `sumX`, `sumY`, `sxx`, `syy`, `sxy`, and the correlation itself, translate directly into formulae, and translating those formulae into equations is simple also. Converting mapping functions, such as those used in the equations for `sumX2`, `sumY2`, and `sumXY`, into columns of formulae is less direct, and converting columns of formulae into mapping functions is the most difficult of the conversions, but (as noted above) they are analogous. A statement sequence in a loop (as in Visual Basic) does not correspond as closely to a column of formulae. (The analogy here between mapping functions and columns of formulae resembles Neuwirth's [1995a, 1995b] comparisons of recursion and chains of formula discussed in chapter 1.)

4.6.2.3 The evaluation models of spreadsheets and functional languages are similar.

Spreadsheets and functional languages have additional common ground. Spreadsheet formula languages are expression-based and stateless. Spreadsheet recalculation, like lazy evaluation, is demand-driven: only cells whose values depend on cells that have changed are updated. Equation bindings (as noted above) map well onto spreadsheet formulae, as both specify relationships, not actions.

4.6.2.4 Functional languages support program transformation.

Of course, there is more than one way to write a functional program, and some functional programs will map onto spreadsheets better than others. An alternate, and more concise, version of the correlation function is shown in algorithm 27. The equations for `sumX2`, `sumX`, `sumY2`, `sumY`, and `sumXY` are absent, and thus the equations for `sxx`, `syy`, and `sxy` are slightly more complicated.

Algorithm 27 A more concise Haskell implementation of the correlation function.

```
correlation :: [(Int,Int)] -> Float

correlation theCoordinateList = sxy / sqrt (sxx * syy)
  where
    sxx = sum (map square xs) - (square (sum xs)) / n
    syy = sum (map square ys) - (square (sum ys)) / n
    sxy = sum (zipWith (*) xs ys) - (sum xs) * (sum ys) / n
    square n = n * n
    xs = map fromIntegral xList
    ys = map fromIntegral yList
    (xList, yList) = unzip theCoordinateList
    n = fromIntegral $ length theCoordinateList
```

Translation between this concise verion and the spreadsheet version of the computation is more difficult. Essentially, the version in algorithm 26 lies between the two. Fortunately, introducing intermediate values in expression-based languages is almost trivial. Extracting `sum (map square xs)` to create an equation for `sumX2` involves a minimal transformation of the program. The reverse transformation — substituting `sumX2`'s expression into the equation for `sxx` — is, if anything, even simpler. (For comparison, consider extracting the `sumX2` computation from the `For` loop in the Visual Basic `MyCorrelation` function from chapter 3.)

4.6.2.5 Imperative languages have functional features, but are still imperative.

The discussion here and in chapter 3 has portrayed the functional and imperative programming models as diametrically opposed. In reality, however, imperative languages have functional languages, and functional languages have stateful operations. Even Haskell, as a pure functional language, has the imperative sublanguage of monadic I/O, quarantined behind the monadic operators so that equational reasoning remains intact.

Perhaps the most imperative of all languages is assembler, which provides no expressions, and no pure functions. Fortran provides expressions, and pure (and impure) functions, but no higher-order functions. Pascal, C, and Modula are still quite imperative, but do provide a limited form of higher-order functions. In all of these languages, expressions are not necessarily pure, because they can contain calls to impure functions. C expressions can contain stateful operators as well as impure functions.

Crossing the boundary into functional languages are Lisp and Scheme, which are impure, weakly typed languages, but do provide true higher-order functions. Scheme also provides explicit laziness with `delay` and `force`. The functional language ML is the pioneer of the strong static typing, type inference, and parametric polymorphism that have become mainstays of functional programming. Finally, the pure functional languages Haskell, Miranda, and Lazy ML provide lazy evaluation.

From this, we might be tempted to mitigate the conflict between spreadsheets and imperative languages by using the "more functional" features of imperative languages. However, the fact that functional features can bridge the gap between the imperative and spreadsheet programming models argues for, not against, the superiority of functional languages as companions of spreadsheets. The imperative model for combining operations (stateful loops, for example) does not transfer over to spreadsheets as well as the mapping operations of the functional model do.

4.6.2.6 There are some mismatches between spreadsheets and functional languages.

Spreadsheets and functional languages are highly compatible, but they do not match perfectly. Spreadsheet cells cannot contain functions, and spreadsheets do not support higher-order functions. Spreadsheet cells cannot contain aggregate values, and spreadsheet formulae do not provide pattern-matching.

The most compelling mismatch is that spreadsheet cells are weakly typed. Since spreadsheet cells can be edited, the type of value in a cell can change over time. Weak typing thus simplifies the editing situation: values with incorrect types cause formula errors (see section 4.2.3.3, page 61 for how Esquisse handles strong typing and spreadsheet-style editing). It may be that strong typing is fundamentally incompatible with the spreadsheet model. It may be that the right way to integrate strong typing into the spreadsheet model remains to be discovered. For now, the advantages of types for spreadsheet (or Esquisse) cells are unclear.

4.6.2.7 There are some mismatches between spreadsheets and programming languages generally.

Some additional differences between spreadsheets and functional languages have to do with general-purpose programming, not functional programming in particular. Spreadsheets have no local environments, no modules, and no scoping. Spreadsheets do not support complex data structures smoothly nor in any general fashion. And spreadsheets have no recursion, except in the manner described by Neuwirth [Neuwirth 1995b].

4.6.2.8 Functional languages provide a good match to the spreadsheet model.

In summary, the features of functional languages (with the exception of strong typing) are well suited to use with spreadsheets. Some higher-order functions can correspond to columns of formulae. Spreadsheet recalculation and lazy evaluation are both demand-driven. Most importantly, spreadsheets and functional languages are expression-based and stateless, vastly aiding transformations within and between the two models.

4.6.3 The evaluation of Esquisse offers insights about learning costs and long-term power.

4.6.3.1 Esquisse can be general-purpose, but is best at merging interaction with computation.

Esquisse provides a rich, interactive interface to a general-purpose functional programming language. As a result, an Esquisse program can, in principle, perform any computation that a functional program can. But formal universality does not carry much practical weight. A more relevant question is: for what problems is Esquisse suited? We could write a compiler in Esquisse, but the benefits Esquisse offers would be underutilized.

As discussed in chapter 1, the strengths of spreadsheets and functional programming languages together are suited for experimentation and exploration of (among other applications) substantial equational models. The examples in section 4.4, page 67, bear this out. Game-playing and search programs are also appropriate applications. In short, Esquisse's strengths are best used for interactive computations.

4.6.3.2 The programming walkthroughs demonstrate the low learning costs of Esquisse.

The line-fitting and temperature conversion examples are highly interactive. The programming walkthroughs for those examples in Esquisse showed that interactivity can be programmed with surprising ease and facility.

The line-fitting and temperature conversion examples are instances of very simple functional programs. Neither example uses higher-order functions or lazy evaluation. In contrast, the logistic map and one-dimensional cellular automaton examples are full-blown functional programs, making heavy use of both higher-order functions and lazy evaluation. The contrast between the two suggests that Esquisse could be used successfully by programmers with different levels of functional programming skill.

Chapter 5

Conclusions

Having drawn on the strengths of spreadsheets and functional programming, Esquisse has made contributions to both programming models. Esquisse has also demonstrated that the cognitive cost of writing interactive programs can be lower than in Haggis or Microsoft Excel with Visual Basic.

5.1 Esquisse describes the interactive structure of the spreadsheet cell in functional terms, and extends it in a principled manner.

First, we have upgraded the spreadsheet to give it the consistency and power of a functional programming environment. We have expanded the range of possible cell values to include arbitrary values, including functions and aggregate data structures, including infinite data structures. We have also expanded the formula language to a full-fledged functional language, with higher-order functions, lazy evaluation, referential transparency, and type inference. Both of these increase the modularity and abstraction available to the user.

Second, we have unpacked the interactive capabilities of the spreadsheet cell into two pure functions, one of which specifies the cell's depiction (graphical, textual, or a combination), the other of which describes how the cell's value changes when the user interacts with the cell. This simple cell structure turns out to have surprising power, supporting a wide range of interactive behaviors.

Third, we have expanded the role of formulae by allowing an arbitrary number of formulae per cell, and by allowing each cell to contain both a value and zero or more formulae. Order of evaluation of formulae is not specified, preserving the declarative nature of the functional language. This enhancement supports multi-way constraints, a common feature in interactive applications.

Fourth, we have identified the features of spreadsheets that mesh poorly with the functional model. Formulae (when included as part of the cell structure) make cells' types complicated, and they reduce the usefulness of typing. Cell updates and type changes due to editing could be better resolved with the functional model. The semantics of mutually recursive formulae still warrant further investigation. Also, we note that providing cells at multiple abstraction levels complicates typing.

Fifth, we have described some of the programming-language issues surrounding the use of the spreadsheet's editing model for program construction. We have presented an preliminary approach, the `TwoFacedObject`, for managing these issues. In a similar vein, we have shown a functional approach to spreadsheet error values.

Comparison of the programming walkthroughs of the temperature conversion and line-fitting examples in Excel with Visual Basic and in Esquisse clearly shows the benefits of Esquisse's principled, consistent approach and powerful extensions.

5.2 The design and development of Esquisse have also informed the study of functional programming environments.

As an environment for functional programming, Esquisse has clear advantages over traditional interpreter-based environments. Like interpreters, it supports incremental program construction, but unlike them, it supports incremental construction of multiple program fragments concurrently on the screen, rather than through the file system and a text editor. Program construction in Esquisse is as natural with graphical entities as well as textual ones.

In Esquisse's functional programming environment, interaction is almost free; complexities such as interface toolkits and functional access to the stateful world of input and output are handled by the cell structure and its implementation. Creating custom interactions is accomplished by editing relatively simple functions, not monadic or continuation-passing functions.

Esquisse offers a much richer interaction between programmer and programming environment than a typical interpreter. Since all parts of the program are continuously visible and manipulable, the cost of switching focus from one part of the program to another is arguably less than that of switching files in an editor. The graphical capabilities of cells offer a much wider range of display possibilities for function results (plots, for example), and the interactive capabilities of combinations of cells offer greater flexibility for testing functions. The function to be tested can be linked via a formula to a cell holding its argument. Changing the value in the argument cell requires less effort and is less error-prone than reentering an expression at an interpreter prompt, and is more flexible than mapping a function over one or more argument lists.

These advantages of Esquisse are even more striking when compared with Haggis. In Haggis, components and their event handlers must be created explicitly, and the programmer must write imperative monadic statements to maintain invariant relationships or constraints between components. (The programmer must also deal with the peculiarities of mingling functional and imperative constructs, as described in Chapter 2). The components thus created are opaque; they cannot be perused without access to their source code; customizing them requires editing and recompiling the program as a whole. Although components can be rather elegantly composed (as described in [FPJ 1995]), the infrastructure required to test a component is much higher than that of pure functional programming or Esquisse. Comparison of the programming walkthroughs of the temperature conversion and line-fitting examples in Haggis and in Esquisse clearly shows the benefits of Esquisse's built-in interaction structure.

5.3 Esquisse offers strong support for creating, customizing, and maintaining the consistency of interface components and data structures.

Haggis, Excel with Visual Basic, and Esquisse all provide mechanisms for creating interactive programs. Esquisse offers substantial support for direct-manipulation creation, customization, and automatic consistency maintenance of interactive data structures, using a pure functional language. Each of these features improves upon Haggis, Excel with Visual Basic, or both, as shown in the following table:

	Haggis	Visual Basic	Esquisse
component creation	within language only	direct manipulation (some provided)	direct manipulation, customized
component behavior	all programmed	some provided, some programmed	all provided (some customized)
access and update	monadic I/O	Visual Basic's object model	formulae (pure functions)
consistency maintenance	mostly explicit	explicit except in spreadsheet	automatic (formulae)
data structures, interface components	some separate data structures required	some separate interface components required	data structures are interface components

The walkthrough examples show these benefits. Esquisse offers some interaction for free; for example, the number cells in temperature conversion example can be edited as soon as they have been created. Linking components together is simple: formulae connect the temperature cells together, and formulae update the line and correlation when points change. Some customizations are easy, such as changing the temperature bars so that dragging changes their heights instead of moving them. Even sophisticated customizations are manageable, such as changing the points in the line-fitting example to store the status, or creating plot cells for the logistic map and one-dimensional cellular automaton. The changes affect the value, the display function, and the event function, but they are still relatively simple.

In summary, the results of the programming walkthroughs of the temperature conversion and line-fitting examples in each of the three systems suggest that implementing these examples in Esquisse involves significantly less cognitive effort than implementing them in Haggis, and that Esquisse also compares favorably with Excel with Visual Basic.

Bibliography

[ACM 1993] Hal Abelson, Natalya Cohen, Jim Miller. Introduction to SWAT: the Scheme Widget Application Toolkit. Massachusetts Institute of Technology, July 1993.

[AP 1994] Peter Achten, Rinus Plasmeijer. The Ins and Outs of Clean I/O. Journal of Functional Programming 5:1, January 1995, pages 81–110.

[AG 1997] Ken Arnold, James Gosling. The Java Programming Language (Java series, second edition). Addison-Wesley, 1997.

[Backus 1978] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. In Communications of the ACM, 21:8, pages 613–641.

[BC 1993] Brigham Bell, Clayton Lewis. ChemTrains: a language for creating behaving pictures. In IEEE Symposium on Visual Languages, 1993, pages 188–195.

[BCLRWWZ 1994] Brigham Bell, Wayne Citrin, Clayton Lewis, John Rieman, Robert Weaver, Nicholas Wilde, Benjamin Zorn. Using the programming walkthrough to aid in programming language design. Journal of Software Practice and Experience 24:1, 1994, pages 1–25.

[BW 1988] Richard Bird, Philip Wadler. Introduction to Functional Programming. Prentice Hall, Hertfordshire, UK, 1988.

[Blough 1996] Eric Blough. Lowering the Marginal Cost of Programming Interactivity in a Functional Programming Environment. Ph.D. Proposal, University of Colorado, May 1996.

[BG 1987] Polly Brown, John Gould. An experimental study of people creating spreadsheets. ACM Transactions on Office Information Systems 5:3, July 1987, pages 258–272.

[BA 1994] Margaret M Burnett, Allen L Ambler. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. Journal of Visual Languages and Computing 5, 1994, pages 29–60.

[CP 1985] Luca Cardelli, Rob Pike. Squeak: a language for communicating with mice. In Proceedings of SIGGRAPH'85, 1985, pages 199–204.

[CH 1993] Magnus Carlsson, Thomas Hallgren. FUDGETS: a graphical user interface in a lazy functional language. In Proceedings of the 6th ACM Conference on Functional Programming and Computer Architecture, pages 321-330. ACM Press, 1993.

[CDZ 1995] Wayne Citrin, Michael Doherty, Benjamin Zorn. The Design of a Completely Visual OOP Language, in Visual Object-Oriented Programming, M. Burnett, A. Goldberg, and T. Lewis (eds.), Prentice-Hall, Englewood Cliffs, NJ, 1995, pages 67–93.

[CR 1991] W. Clinger, J. Rees. Revised[4] Report on the Algorithmic Language Scheme. MIT AI Memo No. 848b, Massachusetts Institute of Technology, 1991.

[dHRvE 1995] Walter de Hoon, Luc Rutten, Marko van Eekelen. Implementing a functional spreadsheet in Clean. Journal of Functional Programming. 5:3, July 1995, pages 383–414.

[dSA 1986] Andrea diSessa, Hal Abelson. Boxer: a reconstructible computational medium. Communications of the ACM 29:9, 1986, pages 859–868.

[Draper 1986] Steven Draper. Display managers as the basis for user-machine communication. In Donald Norman, Steven Draper (eds.), User Centered System Design: New Perspectives on Human-Computer Interaction. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, page 339.

[Eisenberg 1995] Michael Eisenberg. Programmable Applications: Interpreter Meets Interface. SIGCHI Bulletin 27, 2 (April 1995).

[ERT 1987] Michael Eisenberg, Mitchell Resnick, Franklyn Turbak. Understanding Procedures as Objects. In Empirical Studies of Programmers: Second Workshop. Ablex, 1987, pages 14–32.

[FPJ 1995] Sigbjörn Finne, Simon L Peyton Jones. Composing Haggis. In Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics, Maastrict, Netherlands, September 1995.

[FMB 1990] Bjorn Freeman-Benson, John Maloney, Alan Borning. An incremental constraint solver. Communications of the ACM, 33:1, January 1990, pp. 54–63.

[SMFB 1993] Michael Sannella, John Maloney, Bjorn Freeman-Benson, Alan Borning. Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. Technical Report 92-07-05a, Department of Computer Science and Engineering, University of Washington, May 1993. (also appears in Software — Practice and Experience)

[GR 1991] Emden W. Gansner, John H. Reppy. eXene. In Proceedings of the 1991 CMU Workshop on SML, 1991.

[GR 1983] A Goldberg, D Robson. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA.

[Halverson 1998] Microsoft Press, Michael Halvorson. Microsoft Visual Basic 6.0: Deluxe Learning. Microsoft Press, 1998.

[Hill 1992] Ralph D Hill. The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications. In Proceedings of CHI'92, ACM Press, May 1992, pages 335–342.

[HGSG 1990] J-M Hoc, T R G Green, R Samuray, D J Gilmore. Psychology of Programming. London, Academic Press, 1990.

[Hübscher 1995] Roland Hübscher. Visual programming with temporal constraints in a subsumption-like architecture. Technical Report CU-CS-778-95, Department of Computer Science, University of Colorado, 1995.

[Hudak 1989] Paul Hudak. Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, 21:3, September 1989, pages 359–411.

[HS 1988] Paul Hudak, R S Sundaresh. On the expressiveness of purely functional I/O systems. Yale University Research Report YALEU/DCS/RR-665, Department of Computer Science, 1988.

[HPJW+ 1992] Paul Hudak (ed.), Simon Peyton Jones (ed.), Philip Wadler (ed.), Brian Boutel, Jon Fairbairn, Joseph Fasel, Mara M. Guzmn, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, John Peterson. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language, Version 1.2. ACM SIGPLAN Notices 27:5, May 1992.

[HPF 1997] Paul Hudak, John Peterson, Joseph Fasel. A gentle introduction to Haskell, version 1.4. textttthttp://haskell.org/tutorial.

[Hughes 1990] John Hughes. Why functional programming matters. In David A. Turner, editor, Research Topics in Functional Programming, Addison Wesley, 1990, pages 17–42.

[HHN 1986] E.L. Hutchins, John Hollan, Donald Norman. Direct Manipulation Interfaces. In Donald Norman and Stephen Draper (eds.), User Centered System Design, New Perspectives on Human-Computer Interaction, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 87–124.

[Jones 1991] Mark P. Jones. Gofer version 2.30. Yale University, 1991.

[LPJ 1994] John Launchbury, Simon L Peyton Jones. Lazy functional state threads. In Proceedings of the ACM Conference on Programming Language Design and Implementation, June 1994.

[Lewis 1989] Clayton Lewis. New Approaches to Programming. Technical Report CU-CS-429-89, Department of Computer Science, University of Colorado, 1989.

[LVC 1989] Mark Linton, John Vlissides, Paul Calder. Composing user interfaces with Interviews. IEEE Computer, 22:2, February 1989, pages 8–22.

[Liskov 1996] Barbara Liskov. A history of CLU. In History of programming languages II, ACM Press, 1996, pp. 471–497.

[LO 1987] Clayton Lewis, Gary M Olson. Can Principles of Cognition Lower the Barriers to Programming? In Empirical Studies of Programmers: Second Workshop. Ablex, 1987, pages 248–263.

[LRB 1991] Clayton Lewis, John Riemann, Brigham Bell. Problem-centered design for expressiveness and facility in a graphical programming system. Human-Computer Interaction 6, 1991, pages 319–355.

[McPhee 1992] Nic McPhee. The Importance of Lazy Evaluation in Search. Unpublished paper, University of Texas at Austin, 1992.

[Microsoft 1994] Microsoft Corporation. Visual Basic User's Guide [for Microsoft Excel]. Microsoft Corporation, 1994.

[Myers 1989] Brad Myers. Encapsulating Interactive Behaviors. In Proceedings of CHI'89, ACM Press, May 1989, pages 319–324.

[Nardi 1993] Nardi, B. A Small Matter of Programming:: Perspectives on End User Computing. MIT Press, 1993.

[Neuwirth 1995a] Erich Neuwirth. Spreadsheet structures as a model for proving combinatorial identities. Journal of Computers in Mathematics and Science Teaching, 14:3, pp. 419–434, 1995.

[Neuwirth 1995b] Erich Neuwirth. Visualizing Formal and Structural Relationships with Spreadsheets. In A. DiSessa, C. Hoyles, and R. Noss, Computers and Exploratory Learning, Springer-Verlag, 1995.

[Neuwirth 1998] Spreadsheets, Mathematics, Science, and Statistics Education. texttthttp://sunsite.univie.ac.at/Spreadsite/

[NR 1994] Rob Noble, Colin Runciman. Functional languages and graphical user interfaces: a review and a case study. Technical Report 94-223, Department of Computer Science, University of York, February 1994.

[NR 1995] Rob Noble, Colin Runciman. Gadgets: lazy functional components for graphical user interfaces. In Manuel Hermenegildo and S. Doaitse Swierstra, editors, PLILP'95: Seventh International Symposium on Programming Languages, Implementations, Logics, and Programs, volume 982 of Lecture Notes in Computer Science, Springer-Verlag, 1995, pages 321–340.

[Olsen 1989] Dan R Olsen Jr. A Programming Language Basis for User Interface Management. In Proceedings of CHI 89, ACM Press, May 1989, pages 171–176.

[Ott 1984] Lyman Ott. An introduction to statistical methods and data analysis (second edition). PWS Publishers, 1984.

[Ousterhout 1994] John Ousterhout. Tcl and the Tk Toolkit. Reading, MA, Addison-Wesley, 1994.

[PF 1994] David Peak, Michael Frame. Chaos under control: the art and science of complexity. W.H. Freeman and Company, 1994.

[PJGF 1996] Simon L Peyton Jones, Andrew Gordon, Sigbjörn Finne. Concurrent Haskell. In Proceedings of the 23rd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, pages 295–308, January 1996.

[PJW 1993] Simon L. Peyton Jones, Philip Wadler. Imperative functional programming. In ACM Conference on Principles of Programming Languages. ACM Press, January 1993, pages 71–84.

[Pike 1994] Rob Pike. Acme: a user interface for programmers, in Proceedings of the Winter 1994 USENIX Conference, pages 223–234, 1994.

[PLRW 1992] Peter Polson, Clayton Lewis, John Rieman, Cathleen Wharton. Cognitive walkthroughs: a method for theory-based evaluation of user interfaces. International Journal of Man-Machine Studies, 36, pages 741–773.

[PVM 1994] Jorg Poswig, Guido Vrankar, Claudio Morara. VisaVis: A Higher-Order Functional Visual Programming Language. Journal of Visual Languages and Computing 5, 1994, pages 83–111.

[RS 1995] Alexander Repenning, Tamara Sumner. Agentsheets: a medium for creating domain-oriented visual languages. IEEE Computer 28:3, March 1995, pages 17–25.

[Serrarens 1995] Pascal Serrarens. BriX: A Deterministic Concurrent Functional X Windows System. Department of Computer Science, University of Bristol, June 1995.

[Shneiderman 1983] Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. IEEE Computer 16, 8 (1983), pp. 57–69.

[SB 1989] Michael Spenke, Christian Beilken. A spreadsheet interface for logic programming. In Proceedings of CHI'89, ACM Press, May 1989, pages 75–80.

[Thompson 1990] Simon Thompson. Interactive functional programs: a method and a formal semantics. In David Turner (ed), Research Topics in Functional Programming, pages 249–285, Addison-Wesley, 1990.

[Took 1990] Roger Took. Surface interaction: a paradigm and model for separating application and interface, in Proceedings of CHI'90, pages 35–42, April 1990.

[Travers 1994] Michael Travers. Recursive interfaces for reactive objects, in Proceedings of CHI'94, pages 379–385, April 1994.

[Turner 1986] An Overview of Miranda. SIGPLAN Notices, December 1986.

[Wadler 1998a] Philip Wadler. An angry half-dozen. SIGPLAN Notices, 33:2, February 1998, pp. 25–30.

[Wadler 1998b] Philip Wadler. Why no one uses functional languages. SIGPLAN Notices, 33:8, August 1998, pp. 23–27.

[Wadler 1998c] Philip Wadler. Functional Programming in the Real World. `http://www.cs.bell-labs.com/~wadler/realworld/index.html`.

[Wilde 1994] Nicholas Wilde. DataSheets: designing an end-user programming environment to support a specific domain. Technical Report CU-CS-733-94, Department of Computer Science, University of Colorado, 1994.

[WL 1990] Nicholas Wilde, Clayton Lewis. Spreadsheet-based interactive graphics: from prototype to tool. In Proceedings of CHI'90, pages 153–159, 1990.

[Wirth 1989] Niklaus Wirth, J. Gutknecht. The Oberon System, Software Practice and Experience 19:9, September 1989, pages 857–894.

[Wolfram 1994] Stephen Wolfram. Cellular automata and complexity: collected papers. Reading, MA, Addison-Wesley, 1994.

Appendix A

Temperature Conversion Walkthrough and Examples in Haggis

A screen image of the final line-fitting example in Haggis appears in figure 2.1.

A.1 Temperature Conversion Walkthrough in Haggis

```
Step 1:
  choose TwoBars as starting example

  Guiding Knowledge:
    creating a custom component is much more work
      than modifying an existing component
    the TwoBars example looks a lot like the desired example
    the TwoBars interaction shows that the bars are linked
    no other example looks helpful
    the TwoBars example runs and behaves similarly to temperature conversion

  Obstacles:
    TwoBars is horizontal, TC is vertical

Step 2:
  infer that ["*title: Bars"] specifies that the window title is Bars
  change title to Temperature Conversion

  Guiding Knowledge:
    the example's window title is Bars
    Temperature Conversion is a more meaningful and appropriate title

  Obstacles:
    ["*title: Temperature Conversion"] is cluttered
      and modifying it is thus likely to be error-prone

Step 3:
  infer that lev1, lev2, are the two bars
    that dh1, dh2 are the two display handles
    that ["*title: Bars"] specifies that the window title is Bars

  Guiding Knowledge:
    dh is a "standard" abbreviation for display handle
    lev suggests level
    an expression:   componentCreator ... >>= \(c,dh) ->
      creates a component c and its display handle dh
    the example's window title is Bars

  Obstacles:
    understanding (c,dh) pairs is central to using Haggis
    understanding >>= \ ... -> (at least as a template) is also critical

'Do' Step 3:
  infer that lev1, lev2, are the two bars
    that dh1, dh2 are the two display handles
    that ["*title: Bars"] specifies that the window title is Bars

  Guiding Knowledge:
    dh is a "standard" abbreviation for display handle
    lev suggests level
    an expression:   (c,dh) <- componentCreator ...
      in a 'do' context creates a component c and its display handle dh
    the example's window title is Bars
```

```
  Obstacles:
    understanding (c,dh) pairs is central to using Haggis
    understanding do ... <- ... (at least as a template) is also critical
    do x <- ... syntax makes x look undeclared, because it binds a new x
      this is far from obvious and violates programmer intuition
      (but then, programmers have trouble including declarations anyway,
         so maybe this isn't likely to confuse)

Step 4:
  rename lev1, lev2, dh1, dh2 to fBar, cBar, fBarDH, cBarDH
  change title to Temperature Conversion

  Guiding Knowledge:
    names that reflect the purpose of an entity are the least confusing
    programming languages allow renaming of variables and parameters
    program still runs after names are changed
      (not a guarantee, but a positive sign)
    the scope of the lambda expressions extends all the way
      to the end of the indentation level (here, the end of the program)
    indentation level can indicate scope and subexpression grouping

  Obstacles:
    scope may be challenging, especially since:
      >>= (and >> as well) associate to the right
      lambdas are at the end of the line, apparently disembodied
      succeeding lines are not intended to show additional scope level
      indentation level as a scope indicator may be unclear

'Do' Step 4:
  rename lev1, lev2, dh1, dh2 to fBar, cBar, fBarDH, cBarDH
  change title to Temperature Conversion

  Guiding Knowledge:
    names that reflect the purpose of an entity are the least confusing
    programming languages allow renaming of variables and parameters
    program still runs after names are changed
      (not a guarantee, but a positive sign)
    the scope of an x introduced by x <- in a do context extends all the way
      to the end of the 'do' indentation level (here, the end of the program)

  Obstacles:
    scope may be challenging, especially since:
      succeeding lines are not intended to show additional scope level
      indentation level as a scope indicator may be unclear
      in "do x <- ...", x looks undeclared (see other version of this step)

Step 5:
  make bars vertical by changing XAxis to YAxis

  Guiding Knowledge:
    problem statement and domain knowledge indicate vertical bars
    XAxis/YAxis make bars horizontal/vertical
    program will show vertical bars

  Obstacles:
    simply need to know this

Step 6:
  place bars side-by-side by changing vbox to hbox

  Guiding Knowledge:
    problem statement and domain knowledge specify side-by-side bars
    program shows stacked vertical bars
    vbox/hbox arrange their arguments (DHs) from top to bottom / left to right
    DHs (display handles) are the graphical depictions of the bars
    to make the bars vertical and still side-by-side,
      both orientation and boxing must change

  Obstacles:
    some understanding (unclear how much) of vbox, hbox, DH is necessary

Step 7:
  infer that initial heights of bars are 0 for F and 100 for C

  Guiding Knowledge:
    "mkBar range start orient dc" makes a bar of height "start"

  Obstacles:
```

```
    potential confusion between range (low,high) and starting value

Step 8:
  change initial values of bars to 32 for F and 0 for C

  Guiding Knowledge:
    problem statement specifies that bar heights should be proportionate
    32 Fahrenheit is 0 Celsius
    these values are a reasonable starting temperature (domain knowledge)

  Obstacles:
    see step 7

Step 9:
  decide that next step is to change the link connecting the bars
    to an F<-->C conversion

  Guiding Knowledge:
    the TwoBars interaction shows that the two bars are linked
    the problem statement specifies that the thermometers be linked
      by F<-->C conversions

  Obstacles:

Step 10:
  infer that "track" connects the two Bars

  Guiding Knowledge:
    "track src dst" suggests that track updates dst from src
    track is passed fBar and cBar in the forkIO lines

  Obstacles:
    understanding event handlers is central to using Haggis successfully
      it's not clear how deep the understanding must be
    might not need to understand forkIO here, but will need to later

Step 11:
  infer that waitGauge (getLevelGauge src)
    waits for the src's value to change

  Guiding Knowledge:
    Bar is just Level with some additional features
    getLevelGauge returns the value of the Level (src) as a Gauge
    a Gauge is a stateful storage component
    waitGauge waits for the value of the Gauge to change

  Obstacles:
    stateful storage is perhaps more problematic than interface components
      because interface components have visible state change
    all nontrivial Haggis programs use Gauges or similar entities
    waiting for an event is a typical Haggis feature
    understanding that waiting on the Gauge is the same as
      waiting on the level is potentially problematic

Step 12:
  infer that waitGauge (getLevelGauge src) >>= \v ->
    waits for src's value to change, then passes it on as v

  Guiding Knowledge:
    >>= \x -> executes its left side, then passes the result as x
      because
        >>= executes its left side
          then passes the result to the lambda on its right side
        \x -> begins a function with one parameter, x

  Obstacles:
    understanding >>= \ ... -> well enough
    there are two features combined here (in waitGauge, actually)
        waiting for the Gauge to change
        getting the Gauge's value
      and understanding them in combination is significantly more difficult

'Do' Step 12:
  infer that (within a 'do' context) v <- waitGauge (getLevelGauge src)
    waits for src's value to change, then sets v to that value

  Guiding Knowledge:
    in a 'do' context, x <- executes its left side,
      then sets x to the result
```

```
    in a 'do' context, statements are evaluated sequentially
    a 'do' context consists of all statements between a 'do'
      and the next statement indented no farther than the 'do'

  Obstacles:
    understanding do ... <- ... well enough
    see other version of this step for discussion of waitGauge
    imperative syntax is at variance with the functional model

Step 13:
  conclude that the first line of track sets v to the height of the src Bar
    immediately after each interaction with the src Bar

  Guiding Knowledge:
    waitGauge waits for a change in value, then returns the value
    interactions with the Bar change the value
    the value of the Bar is its height and vice versa

  Obstacles:
    knowing that the value of the Bar is its height
      and not some more complicated value

Step 14:
  infer that setLevel dst (100-v)
    sets the value of the dst Bar to be 100-v,
      where v is the value of the src Bar

  Guiding Knowledge:
    setLevel dst x sets the value of the level dst to the new value x
    Bar is just Level with some additional features
    (the (100-v) also suggests that the Bar's value is numeric)

  Obstacles:
    usual caveats about >>= \x ->  or  do x <-

Step 15:
  conclude that track repeatedly waits for the src Bar to change,
    then sets the dst Bar to 100 minus the new value of the src Bar

  Guiding Knowledge:
    the body of track waits and sets
    track calls itself recursively

  Obstacles:
    the fact that track is bottomless recursion might be disturbing

Step 16:
  infer that two "track" processes are created:
    one waiting for changes to fBar and setting cBar, and
    one waiting for changes to cBar and setting fBar

  Guiding Knowledge:
    forkIO takes a expression and creates a process
      that evaluates that expression
    track is called as "track fBar cBar" in one forkIO call, and
    track is called as "track cBar fBar" in the other

  Obstacles:
    not clear how much understanding of forkIO is required
    separately threaded event handlers is a significant concept

Step 17:
  infer that the two processes are different
    because their parameters are different

  Guiding Knowledge:
    "track" is defined only once
    the two track processes are passed parameters differently
    parameterizing functions is a common in functional programming especially
    the two bars need separate track processes
      because they have different behavior
      and because each active component has its own track process

  Obstacles:
    understanding that there are two processes and why they differ

Step 18:
  infer that the fBar track process must set the cBar to 5/9 (F-32)
  infer that the cBar track process must set the fBar to 9/5 C + 32
```

```
  Guiding Knowledge:
    These are the formulas for f->c and c->f
    when one bar changes, the other must change accordingly (problem stmt)

  Obstacles:
    some possibility of source/destination confusion

Step 19:
  infer that (100-v) in "track" must be replaced
    by the appropriate function of v in each case

  Guiding Knowledge:
    100-v computes the value of the dst bar from the value of the src bar
    interaction with the bars shows that the bar heights sum to 100
    the f->c and c->f conversion functions will compute the value of one bar
      from the value of the other

  Obstacles:
    none

Step 20:
  consider copying "track" and replacing 100-v with
    (5/9*(v-32))  and   (9/5*v + 32)  in the two copies

  Guiding Knowledge:
    copying the function creates two versions, one for each process

  Obstacles:
    none

Step 21:
  instead of copying, decide to add a function parameter "convert" to track
    and replace (100-v) with (convert v)

  Guiding Knowledge:
    duplication of expressions can be avoided by parameterization
    the two track processes already have different parameters

  Obstacles:
    copying the function seems more "obvious"
      perhaps even to a functional programmer (certainly to the author)
    the HU might copy the function as in the previous step,
      then notice (or not) that the differences can be extracted
      into the single parameter "convert"

Step 22:
  modify track to take a parameter "convert"
    in definition and recursive call [and type signature! (not present)]
  note that other calls to track must also have a parameter added

  Guiding Knowledge:
    adding a parameter must happen in
      the type signature
      the definition
      recursive calls
      calls in other expressions

  Obstacles:
    standard programming problem of needing to make a change
      in several places to maintain consistency

Step 23:
  in the where clause, create functions convertFtoC and convertCtoF

  Guiding Knowledge:
    the functions need to go in a let or where clause
      in the same scope as track

  Obstacles:
    scope knowledge (including the use of "where")
      in a >>= or do context

Step 24:
  write     convertFtoC f = (5/9)*(f-32)
  and write convertCtoF c = (9/5)*c + 32

  Guiding Knowledge:
    these are the conversion formulas for F<-->C (domain knowledge)
```

```
  Obstacles:
    opportunity for source/destination confusion

Step 25:
  add convertFtoC to "track fBar cBar" to get "track convertFtoC fBar cBar"
  add convertCtoF to "track cBar fBar" to get "track convertCtoF cBar fBar"

  Guiding Knowledge:
    a call to track (now) has the form "track convert src dst"
    adding a parameter to the definition of track requires
      passing in a corresponding value in all calls to track
    passing in the conversion function requires adding it to the calls

  Obstacles:
    need to make changes in multiple places consistently


At this point, the program looks like this:

module Main(main) where

import Concurrent
import Haggis

main =
 mkDC ["*title: TemperatureConversion"] >>= \dc ->
 mkBar (0,100) 32 YAxis dc              >>= \(fBar,fBarDH) ->
 mkBar (0,100) 0  YAxis dc              >>= \(cBar,cBarDH) ->
 realiseDH dc (hbox [fBarDH,cBarDH])    >>
 forkIO (track convertFtoC fBar cBar)   >>
 forkIO (track convertCtoF cBar fBar)   >>
 return ()
 where
  convertFtoC f = (5/9)*(f-32)
  convertCtoF c = (9/5)*c + 32
  track convert src dst =
   waitGauge (getLevelGauge src)        >>= \v ->
   setLevel dst (convert v)             >>
   track convert src dst

or, in the 'do' version:

module Main(main) where

import Concurrent
import Haggis

main =
 do
   dc            <- mkDC ["*title: TemperatureConversion"]
   (fBar,fBarDH) <- mkBar (0,100) 32 YAxis dc
   (cBar,cBarDH) <- mkBar (0,100) 0  YAxis dc
   realiseDH dc (hbox [fBarDH,cBarDH])
   forkIO (track convertFtoC fBar cBar)
   forkIO (track convertCtoF cBar fBar)
   return ()
 where
   convertFtoC f = (5/9)*(f-32)
   convertCtoF c = (9/5)*c + 32
   track convert src dst =
    do
      v <- waitGauge (getLevelGauge src)
      setLevel dst (convert v)
      track convert src dst


Step 26:
  decide that next step is to add numeric labels

  Guiding Knowledge:
    problem statement specifies showing temperatures in numerals
    all other parts of the problem statement are complete

  Obstacles:
    none

Step 27:
  divide adding labels into three steps
```

```
    specifying how the labels will be created
    specifying how the labels will be displayed
    specifying how the labels will change

  Guiding Knowledge:
    components must be created and displayed
    and their interactive behavior and relationship(s) to other components
      must be specified

  Obstacles:
    none

Step 28:
  decide to add label creators first

  Guiding Knowledge:
    labels must be created, so might as well start there
    where labels are created can affect how they are used in the program

  Obstacles:
    none

Step 29:
  infer that creators may be placed anywhere between mkDC and realiseDH

  Guiding Knowledge:
    dc needed from mkDC to pass to creators
    DH needed from creators to pass to realiseDH
    label and bar creators are not otherwise dependent on each other
      because they don't share parameters/return values

  Obstacles:
    unclear how necessary this understanding of dc and dh is
      HU might succeed by sheer imitation

Step 30:
  decide to add label creators together immediately after Bar creators

  Guiding Knowledge:
    the location is between mkDC and realiseDH
    picking a clear convention is useful when no other guidelines exist
    might as well keep them together

  Obstacles:
    none

Step 31:
  add Fahrenheit label creator and Celsius label creator
    label "32" dc >>= \(fLabel,fLabelDH) ->
    label "0"  dc >>= \(cLabel,cLabelDH) ->

  Guiding Knowledge:
    labels are created with "label initialValueString dc"
    "label" creates aLabel and aLabelDH
      where aLabel is the label
        and aLabelDH is its graphical depiction
    the label creator chains with >>= \
    each Bar and its numeric display need to be consistent
    the initial values of the Bars are 32 (Fahrenheit) and 0 (Celsius)
    labels display strings
    "32" is the initial value of the Fahrenheit Bar as a string
    "0"  is the initial value of the Celsius Bar as a string

  Obstacles:
    the >>= \(c,dh) -> contains a lot of syntactic clutter -- including ()
    remembering to use strings for initial values is forgettable
    'dc' is also a bit cluttery -- unmemorable and system-oriented

'Do' Step 31:
  add Fahrenheit label creator and Celsius label creator
    (fLabel,fLabelDH) <- label "32" dc
    (cLabel,cLabelDH) <- label "0"  dc

  Guiding Knowledge:
    labels are created with "label initialValueString dc"
    the label creator creates aLabel and aLabelDH
      where aLabel is the label
        and aLabelDH is its graphical depiction
    the label creator creates/sets them using <-
```

```
    each Bar and its numeric display need to be consistent
    the initial values of the Bars are 32 (Fahrenheit) and 0 (Celsius)
    labels display strings
    "32" is the initial value of the Fahrenheit Bar as a string
    "0"  is the initial value of the Celsius Bar as a string

  Obstacles:
    <- is less cluttery than >>= \(c,dh) ->
      but it does produced the appearance of an undeclared variable
    remembering to use strings for initial values is forgettable
    'dc' is also a bit cluttery -- unmemorable and system-oriented

Step 32:
  decide to display labels under bars, resulting in a 2x2 array

  Guiding Knowledge:
    numbers (being small) will fit best above or below bars (which are narrow)
    since temperatures increase upwards, visual anchor of bar is at bottom
    since numbers won't grow or shrink, best to put them in an anchored place

  Obstacles:
    knowing how to choose best-looking option requires layout experience
      for example, knowing about visual anchors
    knowing what the available layout options are in Haggis
      requires a relatively deep understanding of hbox and vbox

Step 33:
  infer that putting numbers under bars requires changing hbox [fBarDH, cBarDH]

  Guiding Knowledge:
    DHs are graphical depictions of components
    hbox arranges components horizontally in a row (not in a table or grid)
    fLabelDH and cLabelDH need to be passed to realiseDH
      in order for the labels to appear on the display

  Obstacles:
    not clear how much of hbox's layout adjustments need to be understood

Step 34:
  infer that hbox and vbox can be nested

  Guiding Knowledge:
    hbox, vbox :: [DH] -> DH

  Obstacles:
    understanding that hbox and vbox fold [DH] into DH
      is critical to understanding the nesting

Step 35:
  notice that there are two choices for nesting which produce a 2x2 array
    hbox [vbox [fBarDH, fLabelDH], vbox [cBarDH, cLabelDH]]
    vbox [hbox [fBarDH, cBarDH], hbox [fLabelDH, cLabelDH]]

  Guiding Knowledge:
    can group a 2x2 array by columns first, or by rows first

  Obstacles:
    understanding the boxes-and-glue approach is critical to
      understanding the visual difference between these two options
    neither option is the same as a 2x2 grid
    grouping by columns or rows is a system-imposed idiosyncrasy

Step 36:
  decide to choose the first nesting and change hbox [fBarDH,cBarDH] to
    hbox [vbox [fBarDH, fLabelDH], vbox [cBarDH, cLabelDH]]

  Guiding Knowledge:
    we don't know!  either should work
    second choice might result in numbers and bars being different widths
    first choice might result in numbers and bars being different heights
    since heights of numbers won't change (but widths might), go with first

  Obstacles:
    HU might not know difference between choices,
    nor that widths of numbers might change while heights won't
    formatting this long line readably on two lines is a (small) task

Step 37:
  infer that labels (unlike bars) do not need track processes
```

```
  Guiding Knowledge:
    labels will not be changed by direct interactions
    labels will be changed due to interactions with bars
    track processes are needed only for components with direct interactions

  Obstacles:
    since labels must be updated,
      it is far from obvious that they don't need track processes
    HU might end up writing track processes for labels and then discarding them

Step 38:
  conclude that track processes for bars must update labels

  Guiding Knowledge:
    bars will change due to interactions
    labels will change when bars change
    track processes run when bars change
    labels have no track processes of their own

  Obstacles:
    understanding the dynamics of propagating the results of interactions

Step 39:
  infer that a bar's track process only needs to update its own label
    in addition to updating the other level (which it already does)

  Guiding Knowledge:
    waitGauge (in the other bar's track process) waits for a change in value
      regardless of what changes the value (an interaction or another function)
    setLevel (in this bar's track process) sets that Gauge's value
      so, when this bar's process sets the other bar
        the waitGauge for the other bar will return a value
        and the setLabell in for that bar will update its label

  Obstacles:
    the connections here are complex
      even though they are well-designed and consistent
    the HU might well call setLabel for the other label anyway
      not realizing that that is unnecessary

Step 40:
  infer that setLabel is the function to use

  Guiding Knowledge:
    setLabel :: Label -> String -> IO ()
    "setLabel theLabel theString" sets theLabel to display theString

  Obstacles:
    simply must know that setLabel is the right function
      fortunately, this is reasonable
    type signatures are risky because
      understanding how the types match on >>= and >> is harder than average

Step 41:
  infer that the label that setLabel sets must be passed in as a parameter
    to the track function/process

  Guiding Knowledge:
    parameters to track are a function and two Bars
    names occurring in track must be parameters or in a 'let' or 'where'

  Obstacles:
    knowing whether a forked process has access to the globals in its
      defined context is tricky (I don't know)
    in this case, the parameters to the two incarnations of track
      are different, so scope doesn't matter -- they must be passed in

Step 42:
  add srcLabel as the last parameter to track
    in the definition, in the recursive call [and in the type signature]

  Guiding Knowledge:
    parameters must be added in
      definitions
      recursive calls
      type signatures
    order of parameters is not important if the function won't be curried
    when order of parameters isn't dictated by semantics,
```

```
      group parameters logically

  Obstacles:
    standard programming problem of needing to make a change
      in several places to maintain consistency (see step 22)

Step 43:
  change the two track calls from
    forkIO (track convertFtoC fBar cBar)   >>
    forkIO (track convertCtoF cBar fBar)   >>
  to
    forkIO (track convertFtoC fBar cBar fLabel)   >>
    forkIO (track convertCtoF cBar fBar cLabel)   >>

  Guiding Knowledge:
    changing a function's parameter list requires
      changing all calls of that function
    the label to be passed is the one belonging to the source Bar
    fLabel is the label belonging to fBar
    cLabel is the label belonging to cBar

  Obstacles:
    opportunity to confuse source and destination
    possibility of adding labels outside parentheses
    >> is syntactic clutter

'Do' Step 43:
  change the two track calls from
    forkIO (track convertFtoC fBar cBar)
    forkIO (track convertCtoF cBar fBar)
  to
    forkIO (track convertFtoC fBar cBar fLabel)
    forkIO (track convertCtoF cBar fBar cLabel)

  Guiding Knowledge:
    changing a function's parameter list requires
      changing all calls of that function
    the label to be passed is the one belonging to the source Bar
    fLabel is the label belonging to fBar
    cLabel is the label belonging to cBar

  Obstacles:
    opportunity to confuse source and destination
    possibility of adding labels outside parentheses
      (seems less likely here due to absence of clutter)

Step 44:
  infer that (in track)
    setLabel must go between waitGauge and the recursive call
    setLabel can go before or after setLevel

  Guiding Knowledge:
    recursive call must be last
    setLabel needs v, and so must come after waitGauge
    setLabel and setLevel both chain with >> -- or, equivalently, return IO (),
      so order is unimportant

  Obstacles:
    reasonable, given even the minimum of understanding of scope and >>
      that is required by previous steps

Step 45:
  decide to put setLabel before setLevel

  Guiding Knowledge:
    we don't know!
    put "local" changes before changes to other components

  Obstacles:
    if the system is slow, the choice is between
      having the bars consistent and the labels lag
      having each bar consistent with its label, but the pairs out of sync
    it almost certainly doesn't matter,
      which means there's no guideline for what should be first
      (and which is one of the reasons why functional programming
        eliminated sequencing)
      (in imperative languages, some sequencing is arbitrary, and some isn't)

Step 46:
```

```
  infer that setLabel must be passed (show v), not v

  Guiding Knowledge:
    srcLabel needs to display the same value that src (the Bar) does
    v is a number and is the most recent value of the src Bar
    labels display strings, not numbers
    (show v) converts the value v to a string

  Obstacles:
    the need for this conversion is easily forgotten
    possible source-destination confusion is already taken care of,
      because only one label has been passed in to track,
      and only one bar's value is available

Step 47:
  add "setLabel srcLabel (show v)  >>" before the setLevel function

  Guiding Knowledge:
    setLabel is an interaction function that returns IO (),
      so it chains with >>

  Obstacles:
    probably ok given previous steps
    understanding how IO types work with >> and >>=
    >> is syntactic clutter

'Do' Step 47:
  add "setLabel srcLabel (show v)" before the setLevel function

  Guiding Knowledge:
    setLabel is an interaction function that returns IO (),
      so it must be placed in a 'do' context
    a 'do' context consists of all statements between a 'do'
      and the next statement indented no farther than the 'do'

  Obstacles:
    probably ok given previous steps
    understanding 'do' contexts is required
    in this case, 'do' seems to be less confusing

Step 48:
  conclude that the program is finished

  Guiding Knowledge:
    all of the conditions of the problem statement have been met

  Obstacles:
    knowing that program is finished might take a little checking,
      both mental and visual


Hmm, now the program looks like:

module Main(main) where

import Concurrent
import Haggis

main =
 mkDC ["*title: TemperatureConversion"]        >>= \dc ->
 mkBar (0,100) 32 YAxis dc                     >>= \(fBar,fBarDH) ->
 mkBar (0,100) 0  YAxis dc                     >>= \(cBar,cBarDH) ->
 label "32" dc                                 >>= \(fLabel,fLabelDH) ->
 label "0"  dc                                 >>= \(cLabel,cLabelDH) ->
 realiseDH dc (hbox [vbox [fBarDH, fLabelDH],
                     vbox [cBarDH, cLabelDH]]) >>
 forkIO (track convertFtoC fBar cBar fLabel)   >>
 forkIO (track convertCtoF cBar fBar cLabel)   >>
 return ()
 where
  convertFtoC f = (5/9)*(f-32)
  convertCtoF c = (9/5)*c + 32
  track convert src dst srcLabel =
   waitGauge (getLevelGauge src)               >>= \v ->
   setLabel srcLabel (show v)                  >>
   setLevel dst (convert v)                    >>
   track convert src dst srcLabel

or, in the 'do' version:
```

```
module Main(main) where

import Concurrent
import Haggis

main =
 do
   dc               <- mkDC ["*title: TemperatureConversion"]
   (fBar,fBarDH)    <- mkBar (0,100) 32 YAxis dc
   (cBar,cBarDH)    <- mkBar (0,100) 0  YAxis dc
   (fLabel,fLabelDH) <- label "32" dc
   (cLabel,cLabelDH) <- label "0"  dc
   realiseDH dc (hbox [vbox [fBarDH, fLabelDH],
                       vbox [cBarDH, cLabelDH]])
   forkIO (track convertFtoC fBar cBar fLabel)
   forkIO (track convertCtoF cBar fBar cLabel)
   return ()
 where
   convertFtoC f = (5/9)*(f-32)
   convertCtoF c = (9/5)*c + 32
   track convert src dst srcLabel =
    do
      v <- waitGauge (getLevelGauge src)
      setLabel srcLabel (show v)
      setLevel dst (convert v)
      track convert src dst srcLabel
```

A.2 Starting Examples for Temperature Conversion Walkthrough in Haggis

A.2.1 Combinator Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  mkDC ["*title: Bars"]           >>= \dc ->
  mkBar (0,100) 0   XAxis dc      >>= \(lev1,dh1) ->
  mkBar (0,100) 100 XAxis dc      >>= \(lev2,dh2) ->
  realiseDH dc (vbox [dh1,dh2])   >>
  let
    g1 = getLevelGauge lev1
    g2 = getLevelGauge lev2
  in
  forkIO (track lev1 lev2)  >>
  forkIO (track lev2 lev1)  >>
  return ()
  where
    track src dst =
      waitGauge (getLevelGauge src) >>= \v ->
      setLevel dst (100-v)          >>
      track src dst
```

A.2.2 do Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  do
    dc         <- mkDC ["*title: Bars"]
    (lev1,dh1) <- mkBar (0,100) 0   XAxis dc
    (lev2,dh2) <- mkBar (0,100) 100 XAxis dc
    realiseDH dc (vbox [dh1,dh2])
    let
      g1 = getLevelGauge lev1
      g2 = getLevelGauge lev2
    in
```

```
    forkIO (track lev1 lev2)
    forkIO (track lev2 lev1)
    return ()
    where
      track src dst =
        do
          v <- waitGauge (getLevelGauge src)
          setLevel dst (100-v)
          track src dst
```

A.3 Final Temperature Conversion Examples in Haggis

A.3.1 Combinator Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  mkDC ["*title: TemperatureConversion"]        >>= \dc ->
  mkBar (0,100) 32 YAxis dc                     >>= \(fBar,fBarDH) ->
  mkBar (0,100) 0  YAxis dc                     >>= \(cBar,cBarDH) ->
  label "32" dc                                 >>= \(fLabel,fLabelDH) ->
  label "0"  dc                                 >>= \(cLabel,cLabelDH) ->
  realiseDH dc (hbox [vbox [fBarDH, fLabelDH],
                      vbox [cBarDH, cLabelDH]]) >>
  forkIO (track convertFtoC fBar cBar fLabel)   >>
  forkIO (track convertCtoF cBar fBar cLabel)   >>
  return ()
  where
    convertFtoC f = (5/9)*(f-32)
    convertCtoF c = (9/5)*c + 32
    track convert src dst srcLabel =
      waitGauge (getLevelGauge src)         >>= \v ->
      setLabel srcLabel (show v)            >>
      setLevel dst (convert v)              >>
      track convert src dst srcLabel
```

A.3.2 do Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  do
    dc                <- mkDC ["*title: TemperatureConversion"]
    (fBar,fBarDH)     <- mkBar (0,100) 32 YAxis dc
    (cBar,cBarDH)     <- mkBar (0,100) 0  YAxis dc
    (fLabel,fLabelDH) <- label "32" dc
    (cLabel,cLabelDH) <- label "0"  dc
    realiseDH dc (hbox [vbox [fBarDH, fLabelDH],
                        vbox [cBarDH, cLabelDH]])
    forkIO (track convertFtoC fBar cBar fLabel)
    forkIO (track convertCtoF cBar fBar cLabel)
    return ()
    where
    convertFtoC f = (5/9)*(f-32)
    convertCtoF c = (9/5)*c + 32
    track convert src dst srcLabel =
      do
        v <- waitGauge (getLevelGauge src)
        setLabel srcLabel (show v)
        setLevel dst (convert v)
        track convert src dst srcLabel
```

Appendix B

Line-Fitting Walkthrough and Examples in Haggis

A screen image of the final line-fitting example in Haggis appears in figure 2.1.

B.1 Line-Fitting Walkthrough in Haggis

```
Step 1:
  choose Canvas as a starting example

  Guiding Knowledge:
    creating a custom component is much more work
      than modifying an existing component
    the Canvas example looks a lot like the desired example
    the DrawArea example also looks plausible
      but the items created on the Canvas can be dragged around
      whereas the items in the DrawArea example are not draggable

  Obstacles:
    choosing DrawArea would be a big mistake
      because finding which object was clicked
        would involve implementing the equivalent of hitCompositeContainerElt

Step 2:
  infer that ["*title: Canvas"] specifies that the window title is Canvas
  change title to Line-Fitting

  Guiding Knowledge:
    the example's window title is Canvas
    Line-Fitting is a more meaningful and appropriate title

  Obstacles:
    ["*title: Line-Fitting"] is cluttered
      and modifying it is thus likely to be error-prone
    this is not likely to be a problematic step
      and it's not central to the task anyway

Step 3:
  infer that compositeContainer creates a canvas (CompositeContainer)
    that 'canvas' is the canvas
    that 'dh' is its display handle

  Guiding Knowledge:
    dh is a "standard" abbreviation for display handle
    'canvas' suggests a canvas
    an expression:   componentCreator ... >>= \(c,dh) ->
      creates a component c and its display handle dh
    the example's window title is Canvas

  Obstacles:
    HU must have some idea of what a "canvas" is
    understanding  (c,dh) pairs is central to using Haggis
    understanding >>= \ ... -> (at least as a template) is also critical

'Do' Step 3:
  infer that compositeContainer creates a canvas (CompositeContainer)
    that 'canvas' is the canvas
    that 'dh' is its display handle

  Guiding Knowledge:
```

```
    dh is a "standard" abbreviation for display handle
    'canvas' suggests a canvas
    an expression:   (c,dh) <- componentCreator ...
      in a 'do' context creates a component c and its display handle dh
    the example's window title is Canvas

  Obstacles:
    HU must have some idea of what a "canvas" is
    understanding (c,dh) pairs is central to using Haggis
    understanding do ... <- ... (at least as a template) is also critical
    do x <- ... syntax makes x look undeclared, because it binds a new x
      this is far from obvious and violates programmer intuition
      (but then, programmers have trouble including declarations anyway,
         so maybe this isn't likely to confuse)

Step 4:
  infer that catchMouseEv does something with the mouse
    and that 'mouse' is somehow the mouse

  Guiding Knowledge:
    'mouse' suggests the mouse

  Obstacles:
    it's not clear how deeply the HU must understand catchMouseEv
      does the HU need to understand events and event dispatching?
      clearly, this example does not bring up issues of conflicts
        between the contained and container widgets, etc., etc.
    the use of dh as parameter and "return value"
      is potentially quite confusing

Step 5:
  infer that 2 -> describes what happens when mouse button 2 is clicked

  Guiding Knowledge:
    button 2 creates circles in the window
    'case whichButton' suggests that 2 and 1 are (some kind of) buttons
    in 2 ->
      newElement suggests the creation of a new something
      'placeTransparent canvas' suggests placing the something on the canvas

  Obstacles:
    HU must know that mouse buttons are numbered, and how they are numbered
    it's helpful if HU notices that placeTransparent gets newElement's dh

Step 6:
  infer that trackCtrl may be where the mouse behavior is defined

  Guiding Knowledge:
    trackCtrl is passed mouse and canvas in the forkIO line
    the 2 -> case matches the behavior of mouse button 2
    trackCtrl's first line involves getMouseDown and 'MouseButton Down'
      which suggest interactions with the mouse
    dragger also takes mouse and canvas
      but it's called only by trackCtrl
      and it also has cce and offset as parameters -- a less clear match

  Obstacles:
    this is a critical step, and trackCtrl is a substantial function
    this step simply must happen for progress to occur
    some parts are hard to decipher (eg sendMouseEv)

Step 7:
  infer that 1 -> describes what happens when mouse button 1 is clicked

  Guiding Knowledge:
    button 1 allows the items to be dragged
    the 1 -> case calls a function called 'dragger'
    hitCompositeContainerElt and elementHit suggest (weakly) selection

  Obstacles:
    HU must know mouse button numbering (see step 5)
    HU must be familiar with the term "dragging"
    the 1-> case is harder to decipher than the 2 -> case
    there's a lot of other stuff (eg getCompositeContainerEltCoord)
      that is less obvious

Step 8:
  infer that dragger somehow implements dragging the circles in the window
```

```
  Guiding Knowledge:
    the name 'dragger' suggests dragging
    dragger takes the canvas and mouse as parameters (among others)
    one of the cases for whichEvent is 'Motion'

  Obstacles:
    HU must be familiar with the term "dragging"
    it would be helpful if the HU recognized 'cce' as the circle being dragged

Step 9:
  infer that dragger will not be needed

  Guiding Knowledge:
    points in the line-fitting example do not need to be draggable
      (in fact, it doesn't make much sense for them to be draggable)
    dragger implements dragging

  Obstacles:
    this is quite a reasonable step,
    and it's useful, because the HU need not understand how dragger works

Step 10:
  decide to plot the points first,
    then get them to change color when clicked

  Guiding Knowledge:
    the points must be displayed (problem statement)
    the points must be displayed to be clicked on
    the 1 -> case implements (somehow) behavior in response to clicking

  Obstacles:
    this step seems unproblematic

Step 11:
  add the list of points to the program, with the name "problems",
    as a list of pairs

  Guiding Knowledge:
    the list of points must be defined to be plotted
    pairs are a common representation for points
    placeTransparent, hitCompositeContainerElt, and dragger all use pairs
    a list is the canonical homogeneous compound data structure
    data values are defined by name = value-expression
    "problems" is the name of the source data set

  Obstacles:
    if the points are Floats, substantial conversion complications ensue

Step 12:
  add the type signature  problems :: [Coord]  to the program

  Guiding Knowledge:
    type signatures help make programs more readable and errors more detectable
    'Coord' is defined as (Int,Int)
    the points are each of type (Int,Int)
    the type of a list of Coord is [Coord]

  Obstacles:
    the type of Coord is an important detail of Haggis
    knowing that Coord is the right type to use
      is relatively deep (but higher-level) Haggis knowledge

Step 13:
  decide that the point-plotting function (plotPoints)
    should go between the compositeContainer and return () lines

  Guiding Knowledge:
    the compositeContainer line creates the canvas
    plotPoints needs the canvas to plot the points
    the return () line ends the function

  Obstacles:
    plotPoints can go after forkIO
      but it's almost inconceivable that the HU would conclude/infer that
      and data dependencies will later force plotPoints to come before forkIO
    the most "natural" place to put plotPoints
      is between compositeContainer and catchMouseEv
      but this appears to be only stylistic
```

```
Step 14:
  decide to put plotPoints
    between the compositeContainer and catchMouseEv lines

  Guiding Knowledge:
    this seems to be the "natural" place to put it
    (other examples show catchMouseEv and forkIO consistently last

  Obstacles:
    the HU might think that putting plotPoints
      before catchMouseEv and forkIO is necessary, although it is not
    however, assuming that plotPoints must come before catchMouseEv
      has no foreseeable negative consequences

Step 15:
  infer that plotPoints needs two parameters
    the canvas ('canvas')
    the list of points ('problems')

  Guiding Knowledge:
    plotPoints needs the points to be able to plot them
    plotPoints needs the canvas to be able to plot the points on it

  Obstacles:
    seems ok

Step 16:
  decide that (for now at least) plotPoints won't return a useful value

  Guiding Knowledge:
    since the HU is only plotting the points, no information is needed

  Obstacles:
    plotPoints will need to return a useful value later,
      but it's not clear how much the HU should think about that now

Step 17:
  write 'plotPoints canvas problems >>' after the compositeContainer line

  Guiding Knowledge:
    >> is used to chain when no value is passed along

  Obstacles:
    probably ok
    need to understand >>, at least as a template
    >> is not as complex as >> \x ->
      but it's still significant
    knowing the connection between "returning no value" and >>
      requires more than pattern-following knowledge of monadic I/O

'Do' Step 17:
  write 'plotPoints canvas problems' after the compositeContainer line

  Guiding Knowledge:
    no <- is needed when the "statement" doesn't return a value

  Obstacles:
    probably ok
    need to understand how to write statements in a 'do' context
    knowing the connection between "returning no value"
      and a statement without <-
      may require more than pattern-following knowledge of monadic I/O

Step 18:
  write the type signature of plotPoints
    plotPoints :: CompositeContainer -> [Coord] -> IO ??

  Guiding Knowledge:
    plotPoints takes canvas and problems and "doesn't return a value"
    canvas is of type CompositeContainer
    problems is of type [Coord]
    IO functions that "don't return values" return IO a
      where 'a' can be anything

  Obstacles:
    knowing that IO <anything> can be the return type
        for functions that chain with >>
        or for statements in a 'do' context that don't use <-
      requires understanding the type structure of monadic IO, not just >>
```

```
    knowing that the type is not necessarily IO ()
      is significant and subtle, too

Step 19:
  decide to proceed with the function definition
    even though the type signature is incomplete

  Guiding Knowledge:
    if part of the type signature is unconstrained
      see if the function implementation suggests a type

  Obstacles:
    this step should be ok

Step 20:
  decide that a mapping function would be appropriate for plotting the points

  Guiding Knowledge:
    mapping functions apply a function to a list of items
    the points are a list
    plotting a single point is an operation that can be expressed as a function
    there is no other obvious way to do this

  Obstacles:
    mapping would probably occur to the HU
    folding might also, and it's not clear how the HU would know to avoid it
    the apparent conflict between sequentially-evaluated monads
      and purely-functional mapping could be quite problematic

Step 21:
  consider using map to plot the points

  Guiding Knowledge:
    map is the canonical mapping function

  Obstacles:
    HU might be unsure if map is the appropriate function (and rightly so)

Step 22:
  infer that the mapped function must be applied to [Coord]
    rather than CompositeContainer and [Coord]

  Guiding Knowledge:
    map :: (a->b) -> [a] -> [b]
    that is, map takes a function and a list, and returns a list
    the mapped function (whatever it is) will be the function
    [Coord] will be the list

  Obstacles:
    the extra parameter (CompositeContainer)
      could make this inference tricky
    presumably, a functional programmer would successfully make the inference

Step 23:
  infer that the mapped function must have type Coord -> ??

  Guiding Knowledge:
    the mapped function is applied to [Coord]
      and so its parameter must be Coord
    the result type of the mapping doesn't matter,
      so the return type of the mapping is undetermined
    the mapped function must have type a->b not a->b->c
    the mapped function might or might not return IO a
      it's not clear at this point

  Obstacles:
    this follows pretty straightforwardly from step 22
    however, it involves reasoning about partial information
      (in the absence of a return type)
      which is potentially more confusing and error-prone
      than the analogous case with total information
    also, knowing that the return type won't necessarily be IO a
      is a significant, if subtle, issue

Step 24:
  infer that mapped function will plot a single point

  Guiding Knowledge:
    plotPoints (the result of the mapping) will plot all of the points
```

```
    a mapping takes a function which computes/does a single thing
      applies it to a list
      and in that way computes/does a bunch of things

  Obstacles:
    this seems reasonable given steps 20 through 23
    the critical issue is why the HU would notice this now
      noticing this the key to step 26
    the alternative is that the HU wouldn't notice
      until she/he wrote the mapped function
      but that makes for a messier walkthrough

Step 25:
  infer that, to plot the point, the mapped function will need two parameters
    the canvas (type CompositeContainer)
    the point to be plotted (type Coord)

  Guiding Knowledge:
    the mapped function needs the point to be able to plot it
    the mapped function needs the canvas to be able to plot the point on it
    the canvas and point have types CompositeContainer and Coord respectively

  Obstacles:
    seems ok (compare step 15)

Step 26:
  notice that
    to plot the point, the mapped function needs exactly two parameters
      (CompositeContainer and Coord)
    to be mapped, the mapped function needs exactly one parameter
      (Coord)

  Guiding Knowledge:
    steps 23 and 25

  Obstacles:
    the critical issue is in step 24

Step 27:
  notice that the CompositeContainer used will be the same for each point

  Guiding Knowledge:
    all the points will be plotted in the same canvas (CompositeContainer)

  Obstacles:
    the critical issue here is why the HU would notice this
      (because it resolves the extra parameter "problem" in step 26)
    to notice it, the HU probably must also make the inference in step 25

Step 28:
  infer that the mapped function
    need not take CompositeContainer as a parameter
    but can reference it in some other way (currying, scope)

  Guiding Knowledge:
    the CompositeContainer will be the same for each point (step 27)
    specialized functions can be made from general-purpose functions
      by currying
      by returning a lambda expression
    or, specialized functions can be defined in a where (or let) clause
      (within the scope of the general function)

  Obstacles:
    making specialized function by currying or by returning a lambda expression
      is one of the programming "plans" or "games" in functional programming
    knowing about where and let clauses and scope is an important FP feature

Step 29:
  notice that
    if the mapped function doesn't take CompositeContainer as a parameter
    then it will have type Coord -> ??
      which is the type it must have

  Guiding Knowledge:
    the mapped function, to be mapped, must have type Coord -> ?? (steps 23,26)
      to plot the point, it needs CompositeContainer (step 25)
        which it need not take as a parameter (steps 27 and 28)
    the result type of the mapping doesn't matter,
      so the return type of the mapping is unknown
```

```
  Obstacles:
    this follows from steps 23 through 28
      which is quite a bit to put together
    (see also the partial reasoning problem in step 23)

Step 30:
  infer that the mapped function can be written
    as a curried version of a function that takes CompositeContainer
    as a lambda expression returned from such a function
  or in a where (or let) clause inside the scope of plotPoints

  Guiding Knowledge:
    the mapped function needs CompositeContainer (step 25)
      which it need not take as a parameter (steps 27 and 28)
    the ways of referencing something which isn't a parameter are
      currying, lambda expressions, and where (or let) clauses

  Obstacles:
    see step 28

Step 31:
  decide to write the mapped function
    as a curried version of a function which takes CompositeContainer

  Guiding Knowledge:
    a curried version is easier and less messy
      than returning a lambda expression
    a function which takes CompositeContainer seems more generic

  Obstacles:
    there's really no reason to choose currying over 'where'
      in fact, 'where' is the simpler choice
    however, choosing currying now simplifies the walkthrough later

Step 32:
  infer that the uncurried function (call it plotPoint) would have type
    plotPoint :: CompositeContainer -> Coord -> ??

  Guiding Knowledge:
    the name 'plotPoint' is descriptive and analogous to plotPoints
    plotPoint needs CompositeContainer and Coord as parameters (step 26)
    plotPoint's return type is unknown
      because the mapped function's return type is unknown (step 23)

  Obstacles:
    this is pretty reasonable given steps 23 and 26
    it still requires knowledge about currying
      note that producing a function to be curried
        is more complicated conceptually
        than noticing that an existing function can be curried

Step 33:
  infer that what plotPoint should do is what the 2 -> case does

  Guiding Knowledge:
    the 2 -> case corresponds to the user pressing mouse button 2
    pressing mouse button 2 creates a circle on the canvas
    plotPoint should create a circle on the canvas

  Obstacles:
    hopefully, the HU hasn't forgotten by now about 2 ->

Step 34:
  notice that the body of the 2 -> case refers to names outside of it
    canvas
    x
    y

  Guiding Knowledge:
    a name that appears in an expression but is not defined there
      is defined outside of it

  Obstacles:
    this is reasonable for a functional programmer
    the 'canvas' reference would just work, anyway, even if it were copied
    the x and y references would be revealed by the compiler
    of course, newElement and placeTransparent also refer to the outside scope
      but they are functions, and the HU might think of them differently
```

```
        (in spite of the fact that there is no difference in FP)
      and newElement is defined earlier in the let
      and placeTransparent in the Haggis libraries

Step 35:
  infer that (in the placeTransparent line)
    canvas is of type CompositeContainer
    (x-r,y-r) is of type Coord

  Guiding Knowledge:
    the type signature of placeTransparent is:
      placeTransparent :: CompositeContainer
                       -> Coord
                       -> DisplayHandle
                       -> IO CompositeContainerElt
    the line is
      placeTransparent canvas (x-r,y-r) dh
    the parameters to a function must have types that are the same
      as the types in the corresponding positions in the type signature

  Obstacles:
    this should be a piece of cake for a functional programmer

Step 36:
  infer that the plotPoint definition should begin
    plotPoint canvas (x,y) =

  Guiding Knowledge:
    plotPoint takes two parameters, the canvas and the point to be plotted
    (x,y) is a logical name choice for the point to be plotted
      since it is used in the 2 -> case
      and x and y are needed individually in (x-r,y-r)
    r (the offset used for centering) is defined elsewhere in the let
      and so only x and y need to be passed in
    a (non-curried) function definition begins with the name of the function
      followed by its parameters (in the correct order, of course)

  Obstacles:
    this should be straightforward for a functional programmer
    the HU (probably) needs to understand
      that r is being subtracted to center the dot

Step 37:
  decide that canvas and (x-r,y-r) in the body of 2 ->
    can be preserved in the body of plotPoint

  Guiding Knowledge:
    plotPoint should do exactly what the 2 -> case does
    canvas, x, and y are the "parameters" (outside references) of the 2 -> case
    canvas and (x,y) are the corresponding parameters of plotPoint
    r will be in scope for plotPoint just as it was in the 2 -> case

  Obstacles:
    the key here (and in step 33) is understanding that an expression
      can be parameterized and made into a function
   it's not clear how large a conceptual hurdle this is

Step 38:
  write plotPoint (except for the type signature)
    plotPoint canvas (x,y) =
      newElement                              >>= \ dh ->
      placeTransparent canvas (x-r,y-r) dh    >>
      return ()

  Guiding Knowledge:
    this is the body of 2 ->
      written as a function named plotPoint with parameters canvas and (x,y)

  Obstacles:
    this should be trivial for a functional programmer given steps 36 and 37
    the issues around >>= and >> should not come up
      since this is just wrapping up an expression as a function

'Do' Step 38:
  write plotPoint (except for the type signature)
    plotPoint canvas (x,y) =
      do
        dh <- newElement
        placeTransparent canvas (x-r,y-r) dh
```

```
      return ()

  Guiding Knowledge:
    this is the body of 2 ->
      written as a function named plotPoint with parameters canvas and (x,y)

  Obstacles:
    this should be trivial for a functional programmer given steps 36 and 37
    the issues around 'do' and <- should not come up
      since this is just wrapping up an expression as a function

Step 39:
  infer that, given the definition in step 38
    the type signature of plotPoint is
    plotPoint :: CompositeContainer -> Coord -> IO ()

  Guiding Knowledge:
    the parameters of plotPoint have types CompositeContainer and Coord
    the body of plotPoint is a chain using >>= and >>
    the return type of a chain is the return type of the last line of the chain
    return () returns a value of type IO ()

  Obstacles:
    this is actually much more complicated just under the surface
      the return type of plotPoint is the return type of the chain
        but the simple rule 'return type of the last element of of the chain'
        is a consequence of the way the type signatures and associativity
          of >>= and >> work
      and the HU has to know when that rule works
        (only when there are no parentheses, for instance)
        the case expression in trackCtrl is a complicated example
      if the HU doesn't know this rule (or is not convinced)
        the HU needs to understand how to derive type signatures
        working through these type signatures is a lengthy, tedious derivation
        involving the types of >>= and >> and all of the functions used
          and the associativity of >>= and >>
          and the scope of lambda expressions (all the way to the right)

'Do' Step 39:
  infer that, given the definition in step 38
    the type signature of plotPoint is
    plotPoint :: CompositeContainer -> Coord -> IO ()

  Guiding Knowledge:
    the parameters of plotPoint have types CompositeContainer and Coord
    the body of plotPoint is a chain in a 'do' context
    the return type of a chain is the return type of the last line of the chain
    return () returns a value of type IO ()

  Obstacles:
    this is actually much more complicated just under the surface
      the return type of plotPoint is the return type of the chain
        but the simple rule 'return type of the last element of of the chain'
        is a consequence of the way the type signatures and associativity
          of >>= and >> work
        and >>= and >> are invisible in the 'do' syntax
      and the HU has to know when that rule works
        (only when there are no parentheses, for instance)
        the case expression in trackCtrl is a complicated example
      if the HU doesn't know this rule (or is not convinced)
        the HU needs to understand how to derive type signatures
        working through these type signatures is a lengthy, tedious derivation
        involving the types of >>= and >> and all of the functions used
          and the associativity of >>= and >>
          and the scope of lambda expressions (all the way to the right)
        the HU also needs to be able to translate back and forth between
          the 'do' syntax and the >>= and >> syntax

Step 40:
  infer that, given plotPoint, the mapped function is (plotPoint canvas)
    which has type Coord -> IO ()

  Guiding Knowledge:
    the mapped function will be plotPoint
      curried with a CompositeContainer (step 31)
    (plotPoint canvas) is plotPoint curried with a CompositeContainer
    plotPoint has type CompositeContainer -> Coord -> IO ()
      so (plotPoint canvas) has type Coord -> IO ()
```

```
  Obstacles:
    this requires an understanding of currying
      and probably of function application associating to the left
        and type signatures associating to the right
    but is otherwise unproblematic

Step 41:
  conclude that
    map (plotPoint canvas) problems
  should plot the points

  Guiding Knowledge:
    mapping a point-plotting function onto the list of points
      should plot the points
    (plotPoint canvas) is a point-plotting function
    problems is the list of points

  Obstacles:
    this seems pretty straightforward given step 40

Step 42:
  write plotPoints
    plotPoints canvas points =
      map (plotPoint canvas) points

  Guiding Knowledge:
    'plotPoints canvas problems' is how plotPoints is called in tracker
    'points' is an appropriate name for the list of points
    'canvas' is a reasonable name for the canvas

  Obstacles:
    potential confusion of
      'points' (formal argument) versus 'problems' (actual argument)
    and of
      'canvas' (same name for both formal and actual arguments)
    the HU might notice
      that a curried definition could omit 'points' altogther
      that is,
        plotPoints canvas = map (plotPoint canvas)
      or even
        plotPoints canvas = map $ plotPoint canvas

Step 43:
  infer from the type signatures of plotPoint and map
    that the type signature of plotPoints is
      plotPoints :: CompositeContainer -> [Coord] -> [IO ()]

  Guiding Knowledge:
    this follows from the types of 'map' and 'plotPoint'

  Obstacles:
    the HU needs to understand how to derive type signatures

Step 44:
  infer from the plotPoints call in tracker
    that the type signature of plotPoints is
      plotPoints :: CompositeContainer -> [Coord] -> IO a
    where the type of 'a' is unknown for the moment

  Guiding Knowledge:
    CompositeContainer and [Coord] are the types of 'canvas' and 'problems'
    any function which chains (whether with >> or >>=) returns IO a
      (where a is some type, possibly () in the case of >> chaining)

  Obstacles:
    inferring all but the return type should be trivial at this point
    inferring the return type involves understanding (at least partially)
      the types of >> and >>=, at the very least
      understanding that they must be preceded and followed
        by functions returning IO a (for various a)

    up to this point in the walkthrough,
      type signatures probably have been a slight liability (extra work)
    here is where they become critical, albeit also quite messy

'Do' Step 44:
  infer from the plotPoints call in tracker
    that the type signature of plotPoints is
      plotPoints :: CompositeContainer -> [Coord] -> IO a
```

```
    where the type of 'a' is unknown for the moment

  Guiding Knowledge:
    CompositeContainer and [Coord] are the types of 'canvas' and 'problems'
    any function in a 'do' context (with or without '<-') returns IO a
      (where a is some type, possibly () in the case of >> chaining)

  Obstacles:
    inferring all but the return type should be trivial at this point
    inferring the return type involves understanding (at least partially)
      the types of >> and >>=, at the very least
      even when they are absent in the 'do' context
      or, alternatively, understanding the types of "statement succession"
        (with or without '<-') in a 'do' context
      understanding that they must be preceded and followed
        by functions returning IO a (for various a)

    up to this point in the walkthrough,
      type signatures probably have been a slight liability (extra work)
    here is where they become critical, albeit also quite messy

Step 45:
  notice that the return types of the plotPoints call and definition conflict

  Guiding Knowledge:
    a function's type signature must match its definition and all calls
    IO a cannot match [IO ()], because IO cannot match [] (list)
      (no matter how 'a' is instantiated)

  Obstacles:
    the HU must know how to compare type signatures
      including those that have type variables
        (which means knowing how type variables can be instantiated)
    the HU could find this out from the compiler
      which might produce an error message like
        "Couldn't match the type    IO a
         against                    [IO CompositeContainerElt]"
    (the actual error message from GHC is
        "Couldn't match the type
         `_State _RealWorld -> (Either IOError13 a._136, _State _RealWorld)'
         against `[IO CompositeContainerElt]'.
      which is much less lucid)
    but the real questions are why? and what to do about it?

Step 46:
  infer that the plotPoints definition must change to have return type IO a

  Guiding Knowledge:
    the definition returns a list because map does
    the call needs IO a because >> and >>= do
      (the call is constrained by the sequential IO "statement" format)
    the definition can be changed (added to)

  Obstacles:
    the HU should see without much difficulty that the call cannot change
    the HU would reasonably conclude that the definition must change
    the catch is how to change the definition

'Do' Step 46:
  infer that the plotPoints definition must change to have return type IO a

  Guiding Knowledge:
    the definition returns a list because map does
    the call needs IO a because it is in a 'do' context
      (the call is constrained by the sequential IO "statement" format)
    the definition can be changed (added to)

  Obstacles:
    the HU should see without much difficulty that the call cannot change
    the HU would reasonably conclude that the definition must change
    the catch is how to change the definition

Step 47:
  infer that either
    something needs to be added to the plotPoints definition
      to change the return type to IO a
    or map is the wrong mapping function to use

  Guiding Knowledge:
```

```
    sometimes a 'wrapper' around an expression
      can convert the return value to the desired type
    'map' was considered tentatively (steps 20 and 21) and might be wrong
    'map' is used with pure functions, and might not work with monadic ones

  Obstacles:
    the 'wrapper' idea is another functional programming plan/game
    this is a lot of steps to take given that the HU is using map provisionally
      the HU might want to know earlier that mapping is the way to go
        before investing all of this effort
      the HU might want to know about mapIO before going this far
    but the HU probably wouldn't see another way of plotting the points
      except to write something recursive from scratch
      and that something would need to look a lot like mapIO
      (for difficulty in writing mapIO, see obstacles for step 50)

Step 48:
  conclude that adding a wrapper won't solve the problem

  Guiding Knowledge:
    adding 'return' around the body of plotPoints
      makes the return type IO [IO CompositeContainerElt]
        which looks odd and strange and isn't used anywhere
        and is actually a monad of unevaluated closures
          which means that the points never get plotted
            as the HU will see if she/he compiles and runs the program
    taking the head of the list of IO CompositeContainerElt's
      makes the return type IO CompositeContainerElt
      and it seems strange to make the first point 'more important'
      and it also means only the head of the list is evaluated
       which means only the first point is plotted (as the HU will see)
    trying to append '>> return ()' to the body of plotPoints
      won't work because the type returned by map isn't IO a
      which is the same problem that the plotPoints definition is posing

  Obstacles:
    for the HU to reject the 'return' idea purely on aesthetics is farfetched
    for the HU to understand the effect of lazy evaluation
      in conjunction with the strict evaluation of monadic I/O
        requires sophisticated knowledge
        of the mechanics of functional languages and of the IO monad
      (much more so than understanding 'head $ map square [2..]', for example)
    so the most likely outcome is that the HU will run the program
      and wonder why it doesn't work
    the 'head' idea is unlikely to begin with
    the chaining idea seems logical, but quickly falls flat
      the HU might not notice that it's just moving the problem
    eventually (when the values need to be used)
      something like mapIO will be required
        to extract all the values and put them in one monad
        that is, to convert [IO a] to IO [a]

'Do' Step 48:
  conclude that adding a wrapper won't solve the problem

  Guiding Knowledge:
    adding 'return' around the body of plotPoints
      makes the return type IO [IO CompositeContainerElt]
        which looks odd and strange and isn't used anywhere
        and is actually a monad of unevaluated closures
          which means that the points never get plotted
            as the HU will see if she/he compiles and runs the program
    taking the head of the list of IO CompositeContainerElt's
      makes the return type IO CompositeContainerElt
      and it seems strange to make the first point 'more important'
      and it also means only the head of the list is evaluated
       which means only the first point is plotted (as the HU will see)
    wrapping the body with 'do ... return ()'
      won't work because the type returned by map isn't IO a
      which is the same problem that the plotPoints definition is posing

  Obstacles:
    for the HU to reject the 'return' idea purely on aesthetics is farfetched
    for the HU to understand the effect of lazy evaluation
      in conjunction with the strict evaluation of monadic I/O
        requires sophisticated knowledge
        of the mechanics of functional languages and of the IO monad
      (much more so than understanding 'head $ map square [2..]', for example)
    so the most likely outcome is that the HU will run the program
```

```
      and wonder why it doesn't work
    the 'head' idea is unlikely to begin with
    the chaining idea seems logical, but quickly falls flat
      the HU might not notice that it's just moving the problem
    eventually (when the values need to be used)
      something like mapIO will be required
        to extract all the values and put them in one monad
        that is, to convert [IO a] to IO [a]

Step 49:
  conclude that map is the wrong function
    and a monadic version of map is needed

  Guiding Knowledge:
    the 'wrapper' idea won't work
    no other ideas seem plausible
    pure functions like map may not always work with monads
      because monads must be evaluated sequentially
      and pure functions like map aren't sequential

  Obstacles:
    knowing that a monadic version of map is needed is hard to justify
    the 'sequential' argument rests on the HU having a reasonably clear idea
      of how the 'sequential' monadic world and the purely functional world
      are joined

Step 50:
  infer that mapIO :: (a -> IO b) -> [a] -> IO [b]
    is the right function

  Guiding Knowledge:
    it is like map
      but the mapping function must return a monad
      and the result of the mapping is a monad containing a list
    plotPoint (the mapping function) returns a monad (IO ())
    plotPoints needs to return a monad of something
    plotPoints currently returns [IO ()]
      which seems close to IO [()]

  Obstacles:
    knowing that mapIO will work depends on
      understanding how to compare two type-variable-filled type signatures
    the HU simply must know about mapIO, or guess its existence and find it
    if the HU concludes mapping is inappropriate
      she/he could be lost for a long time
    the alternative is to write mapIO, which could be quite a challenge
      involving substantial knowledge of how IO monads work
      including monadic functions (>>=) and their types
      the correct definition is probably something like:
        mapIO :: (a -> IO b) -> [a] -> IO [b]
        mapIO f []     = return []
        mapIO f (x:xs) = f x        >>= \ h ->
                         mapIO f xs >>= \ t ->
                         return h:t

'Do' Step 50:
  infer that mapIO :: (a -> IO b) -> [a] -> IO [b]
    is the right function

  Guiding Knowledge:
    it is like map
      but the mapping function must return a monad
      and the result of the mapping is a monad containing a list
    plotPoint (the mapping function) returns a monad (IO ())
    plotPoints needs to return a monad of something
    plotPoints currently returns [IO ()]
      which seems close to IO [()]

  Obstacles:
    knowing that mapIO will work depends on
      understanding how to compare two type-variable-filled type signatures
    the HU simply must know about mapIO, or guess its existence and find it
    if the HU concludes mapping is inappropriate
      she/he could be lost for a long time
    the alternative is to write mapIO, which could be quite a challenge
      involving substantial knowledge of how IO monads work
      including monadic functions (>>=) and their types
        which is arguably harder in the 'do' context
        where the functions are missing ("statement succession")
```

```
      the correct definition is probably something like:
        mapIO :: (a -> IO b) -> [a] -> IO [b]
        mapIO f []     = return []
        mapIO f (x:xs) = do
                           h <- f x
                           t <- mapIO f xs
                           return h:t

Step 51:
  change 'map' to 'mapIO' in the definition of plotPoints
    and add 'import MiscIO' to the top of the program

  Guiding Knowledge:
    'mapIO' can be applied to same arguments as 'map' was
      (the return type will be different)
    'mapIO' is in the MiscIO library, which must be imported

  Obstacles:
    knowing that (plotPoint canvas) will work for mapIO
      depends on knowing how to manipulate type signatures
    the HU simply must know that mapIO is in the MiscIO library
      and how to import libraries (Concurrent and Haggis offer templates)

Step 52:
  infer that the type signature of plotPoints is now
    plotPoints :: CompositeContainer -> [Coord] -> IO [()]

  Guiding Knowledge:
    this follows from the types of 'mapIO' and 'plotPoint'
    (compare step 43)

  Obstacles:
    the HU needs to understand how to derive type signatures
    (compare step 43)

Step 53:
  conclude that the program will now plot the points correctly

  Guiding Knowledge:
    the type issue with plotPoints has been resolved
    the program runs and plots the points

  Obstacles:
    this should be no problem if the HU still has her/his sanity

Step 54:
  delete case 2 in trackCtrl

  Guiding Knowledge:
    since the program now plots the points
    and since adding new points is not part of the problem statement
      case 2 in trackCtrl is no longer needed

  Obstacles:
    there's no compelling reason to delete this now
      but it will simplify future modifications to newElement
      and thus the walkthrough is simpler if this step occurs here
    even if the HU kept case 2 around indefinitely
      the only obstacles would be propagating the modifications to newElement

Step 55:
  infer that r (in 'r = 10') is the radius
    and change it to 5

  Guiding Knowledge:
    the existing circles look a bit big for points
    r is a common abbreviation for radius
    r is the argument to 'circle'
    'circle' takes a single parameter, the radius, and returns a Picture
    r is defined to be 5 by 'r = 5'

  Obstacles:
    this might be a bit hard to figure out
      but let's face it, it's not critical to interactivity
    the HU could finish the example just fine with big, blobby points

Step 56:
  decide that the next step is to get the dots to change color
    (as decided in step 10)
```

```
  Guiding Knowledge:
    the points need to change color when clicked
    the 1 -> case implements (somehow) behavior in response to clicking
    the points are displayed now, and thus can be clicked on
    the line will need to change as the points are clicked on,
      so drawing it now seems premature

  Obstacles:
    this step seems reasonable and unproblematic
    (note:  inferences about case 1 -> and dragger are in steps 7,8,9)

Step 57:
  infer that case 1 -> finds which dot was clicked
    and that cce is that dot

  Guiding Knowledge:
    hitCompositeContainerElt returns (via chaining) elementHit
    the Nothing ->, Just cce -> is a Maybe construct
    the cce thus extracted is passed to dragger

  Obstacles:
    the HU must understand how >>= binds return values (elementHit)
    the HU must understand hitCompositeContainerElt
    the HU must understand cases and the Maybe construct
    it doesn't seem to be critical to understand (x,y) yet
      nor that getMouseDown ... x y binds x and y to the mouse position
    in steps 76 and 77 understanding (x,y) as coords will matter
      possible confusion with (x,y) in case 2 ->
        we've already concluded that (x,y) in case 2 is of type Coord (step 35)
        but in case 2, (x,y) is both the mouse and the cce coord

'Do' Step 57:
  infer that case 1 -> finds which dot was clicked
    and that cce is that dot

  Guiding Knowledge:
    hitCompositeContainerElt returns (via '<-') elementHit
    the Nothing ->, Just cce -> is a Maybe construct
    the cce thus extracted is passed to dragger

  Obstacles:
    the HU must understand how <- binds return values (elementHit)
    the HU must understand hitCompositeContainerElt
    the HU must understand cases and the Maybe construct
    it doesn't seem to be critical to understand (x,y) yet
      nor that getMouseDown ... x y binds x and y to the mouse position
    in steps 76 and 77 understanding (x,y) as coords will matter
      possible confusion with (x,y) in case 2 ->
        we've already concluded that (x,y) in case 2 is of type Coord (step 35)
        but in case 2, (x,y) is both the mouse and the cce coord

Step 58:
  infer that replacing dragger with one or more statements
    that change the dot's color
    will make the program change the dots' colors when they are clicked

  Guiding Knowledge:
    when dots are clicked, they start being draggable
    dragger is the function that makes them draggable

  Obstacles:
    the HU must understand that dragger implements dragging
      even if she/he doesn't understand how
    the HU must understand the modularity/separability
      of selection and dragging to know that she/he can replace dragging

Step 59:
  infer that there is no way to change the color of a CompositeContainerElt

  Guiding Knowledge:
    CompositeContainerElts are primitive datatypes, not complex data structures
    Haggis provides no functions resembling
      getCompositeContainerEltObject :: CompositeContainerElt -> a

  Obstacles:
    the HU might well assume that CompositeContainerElts can be modified, etc
    especially since the name suggests an element, not an index
    finding out that there is no such functionality
```

```
      could be frustrating and time-consuming
    (the reason it is absent has to do with strong typing and polymorphism
      either not all CompositeContainerElts would be the same type,
      or a CompositeContainer could only hold one kind of object)

Step 60:
  infer that modifying the dot would mean
    keeping track of which dot goes with which CompositeContainerElt
    and modifying the dot when the CompositeContainerElt is clicked

  Guiding Knowledge:
    the dot is created at some point (in newElement)
    to change the dot, there must be a function
      that maps CompositeContainerElts to dots

  Obstacles:
    the HU might not realize that the dot has an "application" handle
      as well as a display handle
      especially given how simple the dots are
    the HU might not know exactly where the dot is created
      even though transparentGlyph's return value
      is analogous to that of other component creators
    keeping track of objects and their indexes is a common programming concern

Step 61:
  decide to delete and recreate the dot rather than store and modify it

  Guiding Knowledge:
    the HU knows how to create it, having done so in plotPoint(s)
    there is a function deleteCompositeContainerElt
      which only needs the CompositeContainerElt (ie not the dot)
    keeping track of the dots and the CompositeContainerElts
      is complicated, more lengthy, and error-prone
      because it involves storage, access, and update

  Obstacles:
    the HU might not think of this
      but it's a pretty reasonable step
    the HU must know about deleteCompositeContainerElt
      but this is reasonable given the number of other CompositeContainer
      functions that the HU must know

Step 62:
  replace dragger call with
    deleteCompositeContainerElt cce

  Guiding Knowledge:
    replacing dragger with deleteCompositeContainerElt
      will delete the dot instead of dragging it
    dragger returns IO ()
      and so does deleteCompositeContainerElt
      so the types are unaffected

  Obstacles:
    the HU might replace the dragger call with
        deleteCompositeContainerElt cce >>
        return ()
      which is redundant but still ok
    the HU might also choose to wait and replace the dragger call
      with the delete-recreate sequence
      and that's ok too
    the HU (upon compile/run) will see that mouse-1 clicks delete the points

'Do' Step 62:
  replace dragger call with
    deleteCompositeContainerElt cce

  Guiding Knowledge:
    replacing dragger with deleteCompositeContainerElt
      will delete the dot instead of dragging it
    dragger returns IO ()
      and so does deleteCompositeContainerElt
      so the types are unaffected

  Obstacles:
    seeing that the types are unaffected
      might be more challenging in a 'do' context
    the HU might replace the dragger call with
        deleteCompositeContainerElt cce
```

```
        return ()
      which is redundant but still ok
    the HU might also choose to wait and replace the dragger call
      with the delete-recreate sequence
      and that's ok too
    the HU (upon compile/run) will see that mouse-1 clicks delete the points

Step 63:
  infer that plotPoint is the right function to recreate the dots

  Guiding Knowledge:
    plotPoint plots a single point
    plotPoint is the function used (by plotPoints) to plot the points initially

  Obstacles:
    if the HU doesn't realize this immediately
      she/he will realize it after writing something resembling plotPoint

Step 64:
  infer that 'withColour blue' in newElement makes the dots blue

  Guiding Knowledge:
    plotPoint plots a single point
    plotPoint calls newElement
    newElement contains 'withColour blue'
    the points plotted are blue

  Obstacles:
    this should be unproblematic
      although the transparentGlyph call has quite a bit of other stuff in it

Step 65:
  consider copying newElement and replacing 'blue' with 'yellow'

  Guiding Knowledge:
    copying the function creates two versions, one for each color of dot
    the dot needs to be recreated in a different color

  Obstacles:
    none

Step 66:
  instead of copying, add a function parameter 'c' to newElement
    and replace 'blue' with 'c'

  Guiding Knowledge:
    adding a parameter avoids duplication of the whole function
    one function is easier to understand and maintain
      than two similar functions are

  Obstacles:
    this is probably quite easy
    however, some of the 'rules' of functional programming
      are suspended or different in the context of monadic I/O
      and this might result in the HU having different ideas/intuitions
      especially since the function (in this case) has no parameters

Step 67:
  infer that the new type signature of newElement is (and write it as)
    newElement :: Colour -> IO DisplayHandle

  Guiding Knowledge:
    blue is a color
    Colour is the type for colors
    adding a parameter involves adding a type and a -> to the type signature

  Obstacles:
    this should be easy
    adding to a no-parameter type signature might also seem a tiny bit strange
      (see obstacles for step 66)

Step 68:
  infer that the definition of plotPoint must also change
    to take the color as a parameter

  Guiding Knowledge:
    plotPoint will be called to recreate the dot
    the dot must be recreated in a different color
```

```
  Obstacles:
    this should be easy
      especially after the HU makes the changes in steps 66 and 67

Step 69:
  infer that plotPoint's color parameter ('colour') should come before 'coord'

  Guiding Knowledge:
    'colour' is a natural name for the color parameter
    plotPoint is curried to be passed to plotPoints
    the mapped function is plotPoint without its 'coord' parameter
    if 'colour' comes after 'coord', then the currying won't work

  Obstacles:
    this is actually a pretty subtle point
      about currying and adding parameters to a function which is curried
    but the HU could probably figure it out by trial and error, also

Step 70:
  edit plotPoint to take an additional parameter 'colour'
    between canvas and coord

  Guiding Knowledge:
    color seems more specific than canvas
    general-to-specific is a way of ordering otherwise unconstrained parameters
    the Haggis functions that take a CompositeContainer take it first
    consistency with existing functions aids clarity and reduces errors

  Obstacles:
    this is no problem
    the order of canvas and colour is irrelevant (so long as it is consistent)

Step 71:
  edit the body of plotPoint to pass 'colour' to newElement

  Guiding Knowledge:
    newElement has been modified to take the color as a parameter
    plotPoint has 'colour' as its color parameter
    nothing else has changed

  Obstacles:
    this should be no problem
      especially since it makes the call of newElement match its definition

Step 72:
  edit the type signature of plotPoint to include Colour in the proper place

  Guiding Knowledge:
    the type signature must match the parameters in the definition

  Obstacles:
    this should be easy as well

Step 73:
  infer that the color must be added
    to the (curried) call to plotPoint in the plotPoints definition

  Guiding Knowledge:
    adding a parameter to a function means changing
      the function definition
      the type signature
      all calls to the function (including recursive calls)
    (see also step 69)

  Obstacles:
    this should be no problem

Step 74:
  rewrite the plotPoints definition as
    plotPoints canvas points =
      mapIO (plotPoint canvas blue) points

  Guiding Knowledge:
    the color must be passed to plotPoint
    the mapped function must have type Coord -> IO ()
    (plotPoint canvas blue) has type Coord -> IO ()

  Obstacles:
    this should be straightforward
```

```
    a possible typo would be to put 'blue' outside the parentheses
      but that's unlikely
    since this function returning IO a is a one-liner
      presumably the >> vs 'do' issue doesn't come up

Step 75:
  infer that the place to add plotPoint is in the Just cce case in trackCtrl
    after the deleteCompositeContainerElt call

  Guiding Knowledge:
    the deleteCompositeContainerElt call deletes the dot
    the plotPoint call will recreate it
    the dot needs to be deleted and recreated

  Obstacles:
    this is quite easy given step 62
    the plotPoint call could precede the deleteCompositeContainerElt call
      this would work just as well
        and the new dot would cover the old one before it was deleted
        (which hardly matters on a fast system with a buffered display)

Step 76:
  infer that there are two choices for the 'coord' parameter
    to the new plotPoint call
    (x,y) and (a,b)

  Guiding Knowledge:
    hitCompositeContainerElt takes (x,y) as its coord parameter
    getCompositeContainerEltCoord returns (a,b)
    dragger takes (x-a,y-b) as its 'offset' parameter (of type coord)

  Obstacles:
    this step seems reasonable
    the HU might even consider (x-a,y-b) a viable option (it isn't)

Step 77:
  choose (a,b) as the coord parameter for the new plotPoint call

  Guiding Knowledge:
    (a,b) is returned by getCompositeContainerEltCoord
     getCompositeContainerCoord returns the coordinates
       of the CompositeContainerElt

  Obstacles:
    it's not obvious how (x,y) and (a,b) are different
    but they must be different, otherwise (x-a,y-b) would always be (0,0)
    the HU might choose (x,y) because it is used in case 2 ->
    unfortunately, choosing (x,y) makes the element 'jump'
      so that the lower left corner is at the mouse pointer hot spot
      this wouldn't matter much if it were the start of dragging
        but it matters here because it changes where the dot is on the display
        and where the dot is affects whether the fitted line 'looks right'

Step 78:
  in trackCtrl, replace
     deleteCompositeContainerElt cce
  with
     deleteCompositeContainerElt cce    >>
     plotPoint canvas yellow (a,b)

  Guiding Knowledge:
    deleteCompositeContainerElt and plotPoint must be chained in an I/O context
    since deleteCompositeContainerElt doesn't return a useful value
      and plotPoint doesn't need one from it
      they can be chained with >>
    plotPoint returns IO ()
      so it doesn't need to be followed by >> return ()
      in order to match the return type of trackCtrl
    (see step 62)

  Obstacles:
    this should be ok given step 62
      but it does involve the 'rules' of monadic IO chaining
      and may involve understanding the IO types (including for >> and >>=)

'Do' Step 78:
  in trackCtrl, replace
     deleteCompositeContainerElt cce
  with
```

```
    deleteCompositeContainerElt cce
    plotPoint canvas yellow (a,b)

  Guiding Knowledge:
    deleteCompositeContainerElt and plotPoint must be chained in an I/O context
    since deleteCompositeContainerElt doesn't return a useful value
      and plotPoint doesn't need one from it
      deleteCompositeContainerElt doesn't need a <- to its left
    plotPoint returns IO ()
      so it doesn't need to be followed by return ()
      in order to match the return type of trackCtrl
    (see step 62)

  Obstacles:
    this should be ok given step 62
      but it does involve the 'rules' of monadic IO "statement succession"
      and may involve understanding the IO types (including for >> and >>=)
        which might involve translating 'do' syntax back to >> and >>= syntax

Step 79:
  infer that the program will now change a clicked dot to yellow
    but that the colors don't toggle
    and there is no computation in the program

  Guiding Knowledge:
    the program runs
      and clicking mouse-1 on a dot changes it to yellow
    clicking again does (appears to do) nothing
    the program is supposed to calculate the fitted line and correlation

  Obstacles:
    this should be no problem

Step 80:
  infer that dragger is no longer needed
    and delete it

  Guiding Knowledge:
    the call to dragger in trackCtrl has been deleted (step 62)
    there are no other calls to dragger (except the recursive call)
    a function which is never called can be deleted

  Obstacles:
    the HU might not delete this yet
      but it's not terribly important
    the HU could have deleted it after step 62
      but would probably concentrate on creating functionality first

Step 81:
  infer that information about the points must be stored somehow

  Guiding Knowledge:
    of course, each point has coordinates
    each point has a status (inlier or outlier)
      that determines its color on the display
      that determines whether it participates in the computation
        (of the fitted line and correlation coefficient)
    each point also has an index (a CompositeContainerElt) in the canvas
    the CompositeContainerElt is only an index into the canvas
      it is not linked to the object (or its color) or anything else
    the CompositeContainer has no means for storing application data

  Obstacles:
    recognizing that the Haggis components cannot store the line-fitting data
      and that some other structure must be created
      is one of the most important (groups of) steps in this walkthrough
    the HU must know this (or come to this conclusion) to solve the problem

Step 82:
  infer that the CompositeContainerElt can be used to look up the point data
    in whatever the data structure turns out to be

  Guiding Knowledge:
    the CompositeContainerElt is the first information about the point
      that is available when the point is clicked
    looking up data requires that some sort of key or index be available

  Obstacles:
    this should be pretty obvious
```

```
    it's also important

Step 83:
  infer that getCompositeContainerEltCoord can be used
    to look up the coordinate, given the CompositeContainerElt

  Guiding Knowledge:
    getCompositeContainerEltCoord is used to find
      where to place the replacement point
    the original point is placed with the coordinates from the data set
    the original point and its replacement are in exactly the same spot

  Obstacles:
    this is a little bit subtle
      and so the HU might miss it and store the CompositeContainerElt
      but it saves the HU from storing the CompositeContainerElt
      so it shortens the walkthrough
      and makes the final program a better solution

Step 84:
  infer that the only other piece of data needed is the status
    and thus that it is the only piece of data that needs to be looked up

  Guiding Knowledge:
    the coordinates and the status are needed for the computation
    the index (CompositeContainerElt) is needed by Haggis
    the index is returned when the points are clicked
    the coordinates can be found from the index
    clicking is the only way the user can interact with the program

  Obstacles:
    this is reasonable, especially given step 83

Step 85:
  infer that the coordinates can be used to look up the status

  Guiding Knowledge:
    coordinates can be compared for equality
    coordinates use integers, not floating-point numbers
      and thus can be compared exactly
    the coordinates are available for use as indexes

  Obstacles:
    knowing that pairs can be compared for equality
      and used as keys in a search
      is a little subtle, but perhaps not to a functional programmer
    knowing that Coords are Ints and not Floats requires good Haggis knowledge
    knowing that exact comparisons of Floats are problematic
      involves knowing some details about floating-point numbers on computers

Step 86:
  infer that the index (CompositeContainerElt) need not be stored

  Guiding Knowledge:
    the index is needed only by Haggis
    the index is needed only to create and destroy dots
      and to look up the coordinates of the point
    the coordinates of the point can then be used to look up the status

  Obstacles:
    this is a little subtle (see step 83)

Step 87:
  conclude that the data structure will store the statuses
    and will be indexed by or searchable by the coordinates

  Guiding Knowledge:
    the CompositeContainerElt need not be stored nor used as an index
    the coordinates can be used to look up the status

  Obstacles:
    this is reasonable

Step 88:
  decide to use a list of (coordinate,status) pairs as the data structure
    as opposed to an array indexed by the coordinates and storing the statuses

  Guiding Knowledge:
    an array indexed by the coordinates and storing the statuses
```

```
      would be very sparse
      would be messy to create
    lists are a more common and familiar data structure in functional languages
    a list of pairs is fairly easy to create and filter

  Obstacles:
    the array would be easy to update and fairly easy to filter
    updating the list is messier (see comparison-of-array-and-list)
    [of course, what's really needed here is a dictionary!]

Step 89:
  infer that a Gauge is the appropriate Haggis component
    for storing the list of (coordinate,status) pairs

  Guiding Knowledge:
    Gauge is the Haggis component that
      stores arbitrary data statefully

  Obstacles:
    the HU simply has to know about Gauge; there's no way around it
    Gauge also supports waiting for a change in the stored data
      but the HU probably doesn't see the value of that at this point
      (and might finish the example just fine without ever seeing that value)

Step 90:
  infer that
      gauge (zip problems (repeat True))
    will create a Gauge that stores the list of (coordinate,status) pairs

  Guiding Knowledge:
    'gauge' takes an initial value
      and creates a (monad containing a) Gauge
    all the points are inliers initially
    'zip'-ping the list of points with a list of True
      will produce the correct initial list of (coordinate,status) pairs

  Obstacles:
    the HU must know about the Gauge creator 'gauge' and how it works
    the HU must know how to zip lists (including when one is infinite)
    the HU must know or be able to create this little trick (zip X (repeat XX))

Step 91:
  infer that the type of
      gauge (zip problems (repeat True))
    (from step 90)
    is IO (Gauge [(Coord,Bool)])

  Guiding Knowledge:
    knowing and/or deriving the types of zipped lists
    gauge returns a monad of a Gauge of its argument
    type constructors associate to the right

  Obstacles:
    this is actually a relatively simple type derivation
    the Gauge and IO types are the new parts
      the novelty might be a little (but not hugely) confusing

Step 92:
  infer that
      gauge (zip problems (repeat True))
    should chain with >>= \ x ->
    and that the 'x' will be a Gauge
      containing the list of (coordinate,status) pairs

  Guiding Knowledge:
    this gauge creation call returns a useful value
      and therefore it should chain with >>=, not >>
    the useful value is the Gauge
    >>= "strips off" the IO monad part

  Obstacles:
    this is the first time in this walkthrough
      that the HU is writing a function (call) that chains with >>=
    the HU must understand >>= and how it extracts values from IO monads
      (at least as well as described in the Guiding Knowledge)

'Do' Step 92:
  infer that
      gauge (zip problems (repeat True))
```

```
    should have x <- on its left
    and that the 'x' will be a Gauge
      containing the list of (coordinate,status) pairs

  Guiding Knowledge:
    this gauge creation call returns a useful value
      and therefore it needs a <-
    the useful value is the Gauge
    <- "strips off" the IO monad part

  Obstacles:
    this is the first time in this walkthrough
      that the HU is writing a function (call) that uses <-
    the HU must understand <- and how it extracts values from IO monads
      (at least as well as described in the Guiding Knowledge)
    the 'do' version may be more complicated than the >>= version
      and the HU might have to translate to >>= to understand

Step 93:
  infer that
      gauge (zip problems (repeat True))  >>= ...
    could be put anywhere in tracker
    but that it should probably go before forkIO

  Guiding Knowledge:
    none of its parameters are the results of other functions in tracker
    its return value is not currently needed by any of the functions in tracker
    trackCtrl will need it to choose the new color of a clicked dot

  Obstacles:
    this is another case of (largely) unconstrained ordering
    the HU might have to move the 'gauge' line up due to later constraints

'Do' Step 93:
  infer that
      ... <- gauge (zip problems (repeat True))
    could be put anywhere in tracker
    but that it should probably go before forkIO

  Guiding Knowledge:
    none of its parameters are the results of other functions in tracker
    its return value (to the left of the <-)
      is not currently needed by any of the functions in tracker
    trackCtrl will need it to choose the new color of a clicked dot

  Obstacles:
    the use of <- is odd
      because it places a return value to the left of the function
    this is another case of (largely) unconstrained ordering
    the HU might have to move the 'gauge' line up due to later constraints

Step 94:
  put the line
      gauge (zip problems (repeat True))  >>= \ theDataGauge ->
    at the very beginning of tracker

  Guiding Knowledge:
    other functions might need to use it
    it's clear that it won't need other function's return values
    'theDataGauge' is a descriptive name
      for the gauge containing the coordinate and status data

  Obstacles:
    this should be ok given steps 92 and 93

'Do' Step 94:
  put the line
      theDataGauge <- gauge (zip problems (repeat True))
    at the very beginning of tracker

  Guiding Knowledge:
    other functions might need to use it
    it's clear that it won't need other function's return values
    'theDataGauge' is a descriptive name
      for the gauge containing the coordinate and status data

  Obstacles:
    this should be ok given steps 92 and 93
```

```
Step 95:
  add a parameter theDataGauge (as the last parameter)
    to the call to trackCtrl in the forkIO line
    to the definition of trackCtrl
    to the recursive call at the end of trackCtrl
  modify the type signature to include Gauge [(Coord,Bool)]

  Guiding Knowledge:
    trackCtrl needs to reference the Gauge containing the point data
    the possible choices are parameters, globals, and where or let clauses
    globals and where or let clauses won't be in the same scope
      as the scope of theDataGauge in tracker
    adding a parameter consists of changing
      the type signature
      the definition
      all calls (including recursive calls)

  Obstacles:
    the only significant possibility for confusion
      concerns the scope issues
        of forked processes
        of values encapsulated in IO monads
    it seems that a forked process "takes with it" its (static) scope
    but it's not clear whether monads can be defined in such a way
      that the forked process can see them
      (except by taking them as parameters)
    the potential for confusion isn't so much about how to do the right thing
      as it's about why all of this works the way it does
    there's also a small possible confusion that the type signature
      should have IO (Gauge [(Coord,Bool)]) instead of Gauge [(Coord,Bool)]

Step 96:
  add a line
      getGauge theDataGauge >>= \ thePointData ->
    between the getCompositeContainerEltCoord
    and deleteCompositeContainerElt lines in trackCtrl

  Guiding Knowledge:
    the status of the point is needed to know what color to make the dot
    the coordinates of the point are needed to look up the status
    the list of (coordinate,status) pairs is needed
      because it is where the coordinates can be used to look up the status
    getGauge gets the list of (coordinate,status) pairs from the Gauge
    thePointData is a somewhat descriptive name for that list
    the getGauge call could come anywhere before the placeElement call
      (because the placeElement call is where the status is needed)
      but it makes sense to put it near the getCompositeContainerEltCoord call
        because the getCompositeContainerEltCoord call
        is where the coordinate is found from the CompositeContainerElt

  Obstacles:
    the HU must know about getGauge; it's part of knowing about Gauge
    the dependency issues are not too hard
      and they would get sorted out later if not anticipated now
    the arbitrariness of statement ordering crops up again here

'Do' Step 96:
  add a line
      thePointData <- getGauge theDataGauge
    between the getCompositeContainerEltCoord
    and deleteCompositeContainerElt lines in trackCtrl

  Guiding Knowledge:
    the status of the point is needed to know what color to make the dot
    the coordinates of the point are needed to look up the status
    the list of (coordinate,status) pairs is needed
      because it is where the coordinates can be used to look up the status
    getGauge gets the list of (coordinate,status) pairs from the Gauge
    thePointData is a somewhat descriptive name for that list
    the getGauge call could come anywhere before the placeElement call
      (because the placeElement call is where the status is needed)
      but it makes sense to put it near the getCompositeContainerEltCoord call
        because the getCompositeContainerEltCoord call
        is where the coordinate is found from the CompositeContainerElt

  Obstacles:
    the scope understanding issues might be harder or easier for <- than >>=
      because the newly introduced value is on the left
    the HU must know about getGauge; it's part of knowing about Gauge
```

```
    the dependency issues are not too hard
      and they would get sorted out later if not anticipated now
    the arbitrariness of statement ordering crops up again here

Step 97:
  decide to use 'partition' to find the point in the list

  Guiding Knowledge:
    'partition' partitions a list according to a predicate
    the predicate can test for the desired coordinate
      and then 'partition' will return
        a list of the point with the desired coordinate
        and a list of the other points
    after the point is found, it needs to be modified (status changed)
    "modification" in a functional language means returning a new value
    returning a new list consists of
      adding the new point to the original list of points minus the old point
    the new list will be new point
      added to the list (returned by 'partition') of the other points

  Obstacles:
    the HU must do sophisticated reasoning and know some standard list tricks

Step 98:
  infer that 'partition' will best be written in a 'where' or 'let' clause

  Guiding Knowledge:
    'partition' returns a pair
      here, the pair consists of the point to change and the other points
    both members of the pair are needed for different purposes
      so the 'partition' expression can either be duplicated
      or it can be put in a where or let
    duplication of any non-small expression is potentially harder to understand

  Obstacles:
    the HU must have some way of dealing with this complicated set of issues

Step 99:
  infer that a 'let' is necessary

  Guiding Knowledge:
    'partition' (more specifically, the predicate passed to it)
      needs (a,b) to compare with the coordinate
      of each (coordinate,status) pair
    (a,b) is bound in a lambda expression in the function (trackCtrl)
      and the scope of the lambda expression does not extend
      into the where clause for the function

  Obstacles:
    this is a subtle point
      the scope of a lambda extends to the end of the sequence of >> and >>=
      but does not extend into a where clause
      which is not obvious
        in part because the associativity of >> and >>= avoids parentheses
        (which helps declutter, but in this case helps ambiguate scope)

Step 100:
  write a 'let ... in' clause for trackCtrl immediately after the getGauge line
    let
      ([thePoint],otherPoints) =
        partition (\ (coord,_) -> coord = (a,b)) thePointData
    in
  indent the succeeding lines (through 'return ()') in one more level

  Guiding Knowledge:
    \ (coord,_) -> coord = (a,b)
      returns True only for the point with coordinates (a,b)
    thePointData is the list of (coordinate,status) pairs
    (a,b) is the coordinate of the clicked point
    [thePoint] can be pattern-matched as a singleton list
      because there should only be one point with coordinates (a,b)
    thePoint and otherPoints are descriptive names
      for the point and the list of other points respectively
    the 'let ... in' clause must go after the getGauge line
      because thePointData is needed to compute the values in the 'let'
    immediately after the getGauge line is a good place
      because later lines will use these values
    indenting subsequent lines helps clarify scope
```

```
  Obstacles:
    the HU must have knowledge of partition, writing small lambdas,
      pattern-matching, and descriptive naming
    the HU must know how to put in a let clause including how far to indent it
    the indentation of lets mixed with >> and >>= sometimes looks strange
    not indenting subsequent lines appears to be standard in Haggis
    ((=(a,b)) . fst) would also work instead of the lambda
      but it is arguably more obscure, if more elegant
    there's really no need to bind a and b separately
      (since the dragger call is gone)
      but it's not worth making (and justifying) that change

'Do' Step 100:
  write a 'let ... in' clause for trackCtrl immediately after the getGauge line
    let
      ([thePoint],otherPoints) =
        partition (\ (coord,_) -> coord = (a,b)) thePointData
    in
  indent the succeeding lines (through 'return ()') in one more level

  Guiding Knowledge:
    \ (coord,_) -> coord = (a,b)
      returns True only for the point with coordinates (a,b)
    thePointData is the list of (coordinate,status) pairs
    (a,b) is the coordinate of the clicked point
    [thePoint] can be pattern-matched as a singleton list
      because there should only be one point with coordinates (a,b)
    thePoint and otherPoints are descriptive names
      for the point and the list of other points respectively
    the 'let ... in' clause must go after the getGauge line
      because thePointData is needed to compute the values in the 'let'
    immediately after the getGauge line is a good place
      because later lines will use these values
    indenting subsequent lines helps clarify scope

  Obstacles:
    the HU must have knowledge of partition, writing small lambdas,
      pattern-matching, and descriptive naming
    the HU must know how to put in a let clause including how far to indent it
    the indentation of lets in a 'do' context sometimes looks strange
    not indenting subsequent lines appears to be standard in Haggis
    ((=(a,b)) . fst) would also work instead of the lambda
      but it is arguably more obscure, if more elegant
    there's really no need to bind a and b separately
      (since the dragger call is gone)
      but it's not worth making (and justifying) that change

Step 101:
  add to the 'let ... in' clause in trackCtrl
    theNewStatus = not (snd thePoint)

  Guiding Knowledge:
    thePoint is a (coordinate,status) pair
    'snd' selects the second member from the pair (in this case, status)
    status is a Bool
    'not' inverts Bool values
    theNewStatus should be the Boolean inverse of the current status
      (that's what toggling is, after all)
    order doesn't matter in a 'let ... in' clause

  Obstacles:
    this should be no problem
    the HU must know about component projections for pairs
      and inverting Boolean values
    the HU must know that ordering doesn't matter in 'let' clauses

Step 102:
  change 'yellow' to '(if theNewStatus then blue else yellow)'
    in the plotPoint call in trackCtrl

  Guiding Knowledge:
    the plotPoint call recreates the clicked point in yellow
    the plotPoint call needs to recreate the clicked point
      in blue if theNewStatus is True
      in yellow if theNewStatus is False

  Obstacles:
    the HU must know how to use if-then-else expressions
    the HU must know to parenthesize the expression
```

```
    the HU could also use case (modeled on the case expressions in the program)
    all of this is typical functional programming

Step 103:
  add to the 'let ... in' clause in trackCtrl
    theNewPointData = ((a,b),theNewStatus):otherPoints

  Guiding Knowledge:
    the new list of (coordinate,status) pairs is a list containing
      a pair consisting of the coordinates and new status of the clicked point
      the (coordinate,status) pairs for the other points
    ':' is the operator to use to combine an item and a list
    order doesn't matter in a 'let ... in' clause

  Obstacles:
    this is (very) basic functional programming
    the nested tuples are a little messy
    the HU should have no problem with this

Step 104:
  add (after the 'let ... in' lines in trackCtrl)
    setGauge theDataGauge theNewPointData >>

  Guiding Knowledge:
    the data structure 'theDataGauge' must be updated
      so that it matches the display
      so that clicking changes the status of the point in the data structure
    theNewPointData is the correct new value for theDataGauge
    setGauge doesn't return a value, so >> is the correct chaining operator
    setGauge must go after the 'let' expression
      because the 'let' expression computes theNewPointData
    setGauge could go anywhere after the 'let' expression
    putting setGauge near getGauge might help program readability (chunking)
      but perhaps not noticably

  Obstacles:
    the HU must know about setGauge; it's part of knowing about Gauge
    the arbitrariness of statement ordering crops up again here

'Do' Step 104:
  add (after the 'let ... in' lines in trackCtrl)
    setGauge theDataGauge theNewPointData

  Guiding Knowledge:
    the data structure 'theDataGauge' must be updated
      so that it matches the display
      so that clicking changes the status of the point in the data structure
    theNewPointData is the correct new value for theDataGauge
    setGauge doesn't return a value, so it doesn't use <-
    setGauge must go after the 'let' expression
      because the 'let' expression computes theNewPointData
    setGauge could go anywhere after the 'let' expression
    putting setGauge near getGauge might help program readability (chunking)
      but perhaps not noticably

  Obstacles:
    the HU must know about setGauge; it's part of knowing about Gauge
    the arbitrariness of statement ordering crops up again here

Step 105:
  infer that the fitted line
    must be drawn initially
    must be deleted and redrawn each time a point is clicked

  Guiding Knowledge:
    the user needs to see the fitted line upon running the program
    the fitted line (and correlation) will change when a point's status does

  Obstacles:
    this seems reasonable
    there is some question as to why the HU would infer this now

Step 106:
  infer that the line-drawing function (call it drawLine)
    will take two endpoints as parameters
    and draw a line on the canvas

  Guiding Knowledge:
    the endpoints will specify where the line is
```

```
    the 'line' function in Haggis takes the endpoints as parameters
    the line needs to be drawn on the canvas (CompositeContainer)

  Obstacles:
    the HU must know about the 'line' function

Step 107:
  infer that drawLine can be modeled on newElement combined with plotPoint

  Guiding Knowledge:
    newElement creates a dot (given a color)
    plotPoint calls newElement, then places the dot on the canvas

  Obstacles:
    it's reasonable for the HU to look at plotPoint for inspiration
    modeling a function on two functions in combination is a little complicated

Step 108:
  infer that 'circle r' in newElement
    could be replaced by 'line (x1,y1) (x2,y2)'

  Guiding Knowledge:
    'circle r' creates a circle
    'line (x1,y1) (x2,y2)' creates a line
    both the circle and the line are of type Picture

  Obstacles:
    the HU has to know that 'circle' and 'line' both create Pictures
      because there is no reason she/he would assume
      that they return the same type

Step 109:
  infer that fillSolid can be discarded

  Guiding Knowledge:
    lines don't have interiors which can be filled or not filled
    fillSolid has type Picture -> Picture
      and so the types are unaffected by its removal

  Obstacles:
    this step seems reasonable
    it's possible that fillSolid could be meaningful for a line
    the HU needs to know the type of fillSolid

Step 110:
  write the first part of drawLine:
  drawLine canvas ((x1,y1),(x2,y2)) =
    transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc >>= \ (_,dh) ->

  Guiding Knowledge:
    this is the first line of newElement with
      the color changed to red
      fillSolid omitted
      'circle' changed to 'line' as described in step 108
    ((x1,y1),(x2,y2)) needs to be a parameter of drawLine
    'canvas' needs to be a parameter of drawLine
    'canvas' comes before 'coord' in plotPoint (and in many Haggis functions)
      so for consistency it should come first here also
    (_,dh) was used in newElement, so it should work here, too

  Obstacles:
    this seems reasonable given steps 107 through 109
    there's a lot of stuff just copied without full comprehension
      (transparentGlyph, dc, and >>= \ (_,dh) ->)

'Do' Step 110:
  write the first part of drawLine:
  drawLine canvas ((x1,y1),(x2,y2)) =
    (_,dh) <- transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc

  Guiding Knowledge:
    this is the first line of newElement with
      the color changed to red
      fillSolid omitted
      'circle' changed to 'line' as described in step 108
    ((x1,y1),(x2,y2)) needs to be a parameter of drawLine
    'canvas' needs to be a parameter of drawLine
    'canvas' comes before 'coord' in plotPoint (and in many Haggis functions)
      so for consistency it should come first here also
```

```
    (_,dh) was used in newElement, so it should work here, too

  Obstacles:
    this seems reasonable given steps 107 through 109
    there's a lot of stuff just copied without full comprehension
      (transparentGlyph, dc, and '(_,dh) <-')

Step 111:
  infer that the 'return dh' line of newElement
  and the 'newElement' line of plotPoint
    can be eliminated

  Guiding Knowledge:
    'return dh' creates a return value from the dh in the previous line (_,dh)
    'newElement colour >>= \ dh ->' just calls newElement and extracts the dh

  Obstacles:
    part of this is functional programming (and generic programming)
      knowledge about converting functions to expressions (and back)
    the HU must know about how the IO monad complicates this conversion

'Do' Step 111:
  infer that the 'return dh' line of newElement
  and the 'newElement' line of plotPoint
    can be eliminated

  Guiding Knowledge:
    'return dh' creates a return value from the dh in the previous line (_,dh)
    'dh <- newElement colour' just calls newElement and extracts the dh

  Obstacles:
    part of this is functional programming (and generic programming)
      knowledge about converting functions to expressions (and back)
    the HU must know about how the IO monad complicates this conversion
    is the 'do' syntax harder or easier here?

Step 112:
  write the second line of the body of drawLine:
    placeTransparent canvas (0,0) dh

  Guiding Knowledge:
    placeTransparent places the (DisplayHandle of the) object on the canvas
    the line's origin is the origin of the canvas
      and so (0,0) is where the line Glyph should be placed

  Obstacles:
    this is simple copying except that
    knowing that (0,0) is the correct Coord to pass in is quite tricky
      the HU might expect that one of the endpoints would be the correct Coord
    this is just something that the HU must know

Step 113:
  infer that drawLine needs to put the line at the back of the display list
    and that toBackCompositeContainerElt will accomplish this
    and that toBackCompositeContainerElt needs the CompositeContainerElt
      for the line

  Guiding Knowledge:
    the line may be clicked on
    the line may cover one or more points and prevent them from being clicked
    putting the line at the back keeps it from covering the points

  Obstacles:
    this is another thing that the HU must know
      from programming interaction with multiple overlapping objects
    the HU must know about display lists

Step 114:
   add to the end of the second line of the body of drawLine:
     >>= \ lineCCE ->

  Guiding Knowledge:
    placeTransparent returns (in a monad) a CompositeContainerElt
    lineCCE is a descriptive name for the line's CompositeContainerElt
    lineCCE is needed by toBackCompositeContainerElt

  Obstacles:
    the HU must know (as in many other steps) how to use >>= \ x ->
      to extract the result of a function of type IO x
```

```
'Do' Step 114:
   add to the beginning of the second line of the body of drawLine:
     lineCCE <-

  Guiding Knowledge:
    placeTransparent returns (in a monad) a CompositeContainerElt
    lineCCE is a descriptive name for the line's CompositeContainerElt
    lineCCE is needed by toBackCompositeContainerElt

  Obstacles:
    the HU must know (as in many other steps) how to use x <-
      to extract the result of a function of type IO x

Step 115:
  write the third line of the body of drawLine:
    toBackCompositeContainerElt lineCCE >>

  Guiding Knowledge:
   toBackCompositeContainerElt puts the lineCCE at the back of the display list
   toBackCompositeContainerElt doesn't return a useful values
   I/O functions that don't return useful values chain with >>

  Obstacles:
    this is probably ok given step 113

'Do' Step 115:
  write the third line of the body of drawLine:
    toBackCompositeContainerElt lineCCE

  Guiding Knowledge:
   toBackCompositeContainerElt puts the lineCCE at the back of the display list
   toBackCompositeContainerElt doesn't return a useful values
   I/O functions that don't return useful values don't use <-

  Obstacles:
    this is probably ok given step 113

Step 116:
  infer that drawLine needs to return the line's CompositeContainerElt

  Guiding Knowledge:
    the line may be clicked on
    clicking on the line should have no effect
    the only way to detect clicking on the line
      is to compare the CompositeContainerElt clicked
      with the line's CompositeContainerElt

  Obstacles:
    the HU has to foresee this problem or deal with it when it comes up
      (assuming that the HU forsees the problem simplifies the walkthrough)
    forseeing this problem involves experience
       with programming interaction and handling events

Step 117:
  write the fourth and final line of the body of drawLine:
    return lineCCE

  Guiding Knowledge:
    drawLine needs to return the line's CompositeContainerElt
    drawLine is an I/O function, and so it needs to return a monad
    'return lineCCE' returns the line's CompositeContainerElt in a monad

  Obstacles:
    this seems fine given the extensive knowledge other steps require about I/O

Step 118:
  write the type signature for drawLine
    drawLine :: CompositeContainer -> (Coord,Coord) -> IO CompositeContainerElt

  Guiding Knowledge:
    the parameters are 'canvas' and '((x1,y1),(x2,y2))'
      which have types CompositeContainer and (Coord,Coord) respectively
    the return value is a monad of a CompositeContainerElt
      and therefore has type IO CompositeContainerElt

  Obstacles:
    the HU must know how to construct type signatures
    the HU must know that a monad of x has type IO x
```

```
Step 119:
  infer that
    a drawLine call in tracker will draw the initial line
    a drawLine call in trackCtrl will redraw the line as needed

  Guiding Knowledge:
    the line needs to be drawn initially and after each interaction (step 105)
    drawLine is the function that will draw the line
    tracker is the function that sets up the initial interface
    trackCtrl (specifically the 'Just cce' case of case '1')
      is the function in which the interaction's changes take place

  Obstacles:
    this is important, but doesn't seem difficult

Step 120:
  infer that
    the initial line will be calculated from 'problems' and ((0,0),(100,100))
    redrawn lines will be calculated from theNewPointData and ((0,0),(100,100))
    both lines will be calculated by calling fittedLineAndCorrelation

  Guiding Knowledge:
    fittedLineAndCorrelation calculates the fitted line (and correlation)
      from a list of points and a clipping window
    the initial list of points is 'problems'
    the list of points for the redrawn line is part of 'theNewPointData'

  Obstacles:
    this seems ok
    knowing that theNewPointData is not (quite) the right value
      is a small chance for (brief) confusion
    knowing to use theNewPointData
      rather than thePointData (which is out-of-date) could be tricky
      (this is an imperative trap!)

Step 121:
  infer that the list of inliers in theNewPointData is
    filter snd theNewPointData

  Guiding Knowledge:
    filter takes a predicate and a list and returns the elements of the list
      for which the predicate is True
    snd selects the second component of a pair
    the second element of each pair in theNewPointData is a Boolean
      indicating whether the point is an inlier
    the desired lists consists only of the inliers

  Obstacles:
    the HU might write the equivalent (but slightly more complicated)
      filter (\ (_,b) -> b == True) theNewPointData
      filter (\ (_,b) -> b) theNewPointData
    because using 'snd' depends on the combination of 'snd'
      with a list of (_,Bool) pairs
      where the Bool is the filtering criterion

Step 122:
  infer that the desired list of coords for calculating the redrawn line is
    map fst (filter snd theNewPointData)

  Guiding Knowledge:
    (filter snd theNewPointData) is still a list of pairs
    map applies a function to each element of a list
    fst selects the first component of a pair

  Obstacles:
    the HU might miss this (the author did initially)
    it requires thinking clearly about types (formally or informally)
    the HU is unlikely to write a lambda in place of fst (compare step 121)
      because the interaction of functionality in that case is absent here

Step 123:
  in the 'where' clause at the end of the main program, add
    theClippingWindow = ((0,0),(100,100))

  Guiding Knowledge:
    the clipping window will be the same for both the initial and redrawn lines
      so it can be defined in a 'where' clause accessible to both
    definitions in a function's 'where' clause
```

```
      are accessible throughout the entire function
    the definition of theClippingWindow needs to be accessible
      in both tracker and trackCtrl

  Obstacles:
    this seems ok
    indentation could trip the HU up briefly

Step 124:
  infer that (in tracker) drawLine can go anywhere after compositeContainer

  Guiding Knowledge:
    drawLine needs the canvas to draw on it
    compositeContainer creates the canvas

  Obstacles:
    this seems reasonable
    the HU must know about dependencies
    compare steps 13,93,100,104

Step 125:
  add drawLine canvas theLineCoords >>= \ lineCCE ->
    after plotPoints in tracker

  Guiding Knowledge:
    drawLine can go anywhere after compositeContainer
    it seems natural to plot the points first, then draw the line
    drawLine takes the canvas and the line coordinates as parameters
    the line coordinates can be a single name (theLineCoords)
      because they are not needed individually
    drawLine returns a monad of a line CompositeContainerElt
    the drawLine call is being placed in a monadic I/O context
    drawLine chains with >>= in a monadic I/O context

  Obstacles:
    here is another case of arbitrary statement ordering
    as with steps 92,93,94 the HU is inserting into a monadic chain
      a function that returns a useful value and chains with >>=
    knowing that theLineCoords will do instead of ((x1,y1),(x2,y2))
      is subtle, but luckily not particularly critical

'Do' Step 125:
  add lineCCE <- drawLine canvas theLineCoords
    after plotPoints in tracker

  Guiding Knowledge:
    drawLine can go anywhere after compositeContainer
    it seems natural to plot the points first, then draw the line
    drawLine takes the canvas and the line coordinates as parameters
    the line coordinates can be a single name (theLineCoords)
      because they are not needed individually
    drawLine returns a monad of a line CompositeContainerElt
    the drawLine call is being placed in a monadic I/O context
    drawLine uses '<-' in a 'do' context

  Obstacles:
    here is another case of arbitrary statement ordering
    as with steps 92,93,94 the HU is inserting into a statement sequence
      a function that returns a useful value and uses <-
    knowing that theLineCoords will do instead of ((x1,y1),(x2,y2))
      is subtle, but luckily not particularly critical

Step 126:
  add a 'where' expression at the beginning of tracker
    where
      (theLineCoords,correlation) =
        fittedLineAndCorrelation problems theClippingWindow

  Guiding Knowledge:
    fittedLineAndCorrelation computes the endpoints for the initial fitted line
      from the initial set of problems and the 100x100 clipping window
    the 'where' clause defines 'theLineCoords' and 'correlation' locally
      to the tracker function

  Obstacles:
    the 'where' looks a little odd hanging off by itself at the end of tracker
      and the indentation might be a little tricky
    these are aspects of functional programs that have numerous nested clauses
      and are not peculiar to interactive programs
```

```
Step 127:
  add to the 'let' expression in trackCtrl
    (theNewLineCoords,theNewCorrelation) =
      fittedLineAndCorrelation (map fst (filter snd theNewPointData))
                               theClippingWindow

  Guiding Knowledge:
    the definition of (theNewLineCoords,theNewCorrelation)
      must be in the 'let' expression
      because it needs theNewPointData
      which would not be accessible from a 'where' clause (see step 99)
    also, the 'let' has been written
      and putting all definitions together is typically clearer
    this definition combines the results of steps 122 and 123

  Obstacles:
    this seems ok
    since order of definitions in a let is irrelevant
      the placement of this definition is left to style and aesthetics

Step 128:
  after plotPoint in trackCtrl, add
    drawLine canvas theNewLineCoords >>= \ theNewLineCCE ->

  Guiding Knowledge:
    the whole point of calculating theNewLineCoords was for drawing the line
    the CompositeContainerElt of the new line must be kept track of
      so that clicking on it can be ignored (see step 116)
      and so that it can be deleted (step 129)
    it seems appropriate to (re)draw the line after (re)plotting the points

  Obstacles:
    the HU has to know that a special case must be written
      so that clicking the line will be ignored
    this is, perhaps, nonintuitive

'Do' Step 128:
  after plotPoint in trackCtrl, add
    theNewLineCCE <- drawLine canvas theNewLineCoords

  Guiding Knowledge:
    the whole point of calculating theNewLineCoords was for drawing the line
    the CompositeContainerElt of the new line must be kept track of
      so that clicking on it can be ignored (see step 116)
      and so that it can be deleted (step 129)
    it seems appropriate to (re)draw the line after (re)plotting the points

  Obstacles:
    the HU has to know that a special case must be written
      so that clicking the line will be ignored
    this is, perhaps, nonintuitive

Step 129:
  infer that, to delete the old line, the line CompositeContainerElt is needed

  Guiding Knowledge:
    deleteCompositeContainerElt requires a CompositeContainerElt parameter
    the old point was deleted by passing its CompositeContainerElt
      to deleteCompositeContainerElt

  Obstacles:
    this seems to be ok
    the HU might forget to delete it
      but would realize the need to do so
      when the window became cluttered with lines

Step 130:
  after 'deleteCompositeContainerElt cce >>' in trackCtrl, add
    deleteCompositeContainerElt theLineCCE >>

  Guiding Knowledge:
    conceptually, it makes some sense to group the deletions together
    theLineCCE is a descriptive name for the line's CompositeContainerElt

  Obstacles:
    this is "working from the inside out"
      which can be confusing
      but does start from the need for the change
```

```
'Do' Step 130:
  after 'deleteCompositeContainerElt cce' in trackCtrl, add
    deleteCompositeContainerElt theLineCCE

  Guiding Knowledge:
    conceptually, it makes some sense to group the deletions together
    theLineCCE is a descriptive name for the line's CompositeContainerElt

  Obstacles:
    this is "working from the inside out"
      which can be confusing
      but does start from the need for the change

Step 131:
  add theLineCCE as a parameter to trackCtrl
    in the definition of trackCtrl
    in the forkIO call in tracker

  Guiding Knowledge:
    'theLineCCE' in tracker needs to be passed to trackCtrl
    'theLineCCE' used to delete the line
      needs to be a parameter in trackCtrl's definition

  Obstacles:
    this seems ok for a programmer who knows how to add parameters to functions

Step 132:
  add theNewLineCCE to the recursive call in trackCtrl

  Guiding Knowledge:
    trackCtrl's recursive call must match its definition
    the next invocation of trackCtrl
      must have the new line CompositeContainerElt

  Obstacles:
    up to this point, the parameters passed to trackCtrl
      and those passed to the recursive call
      have been identical
    this step brings up the (previously dormant) issue
      of storing state in parameters
    in fact, thePointData and theNewPointData could be passed in this fashion
      eliminating the need for Gauge
    (it's not clear that a Gauge-less solution would be simpler)
    see step 134 for how messy getting a return value
      from nested imperative cases can be
    the need for onward-passed parameters to be returned
      from nested imperative cases
      suggests that (especially multiple) onward-passed parameters
        would be fairly complicated

Step 133:
  change 'return ()' at the end of the 'Just cce' case in trackCtrl to
    return theNewLineCCE
  and change the >> at the end of the case expression to
    >>= \ theNewLineCCE ->

  Guiding Knowledge:
    the nested 'Just cce' case didn't return a value ('return ()')
      and the outer 'case' expression chained with >>
    but the 'Just cce' case needs to return theNewLineCCE
      and the outer 'case' expression needs to chain with >>=
      because theNewLineCCE's scope ends
      with the end of the 'Just cce' case

  Obstacles:
    this is where the functional and imperative worlds collide
    the HU might gather that theNewLineCCE's scope is over
      by the change in indentation
    but she/he might be misled by the imperative "until-the-end" rule
    actually modifying the return value and the chaining operator
      requires significant understanding of the way the monadic expressions
      chain with the monadic operators >> and >>=

'Do' Step 133:
  change 'return ()' at the end of the 'Just cce' case in trackCtrl to
    return theNewLineCCE
  and, at the beginning of the outer 'case' expression, add
    theNewLineCCE <-
```

```
  Guiding Knowledge:
    the nested 'Just cce' case didn't return a value ('return ()')
      and thus the outer 'case' expression didn't need <-
    but the 'Just cce' case needs to return theNewLineCCE
      and the outer 'case' expression needs to use <-
      because theNewLineCCE's scope ends
      with the end of the 'Just cce' case

  Obstacles:
    this is where the functional and imperative worlds collide
    the HU might gather that theNewLineCCE's scope is over
      by the change in indentation
    but she/he might be misled by the imperative "until-the-end" rule
    actually modifying the return value and adding the <-
      requires significant understanding of the way the monadic expressions
      are joined by "statement succession"
      (or by >> and >>=, which requires translation from the 'do' syntax)
    putting <- at the beginning of the outer 'case' expression
      seems more error-prone than putting >>= after it

Step 134:
  change 'Nothing -> return ()' in trackCtrl to
    Nothing -> return theLineCCE
  change '_ -> sendMouseEv mouse deviceEvent' to
    _ -> sendMouseEv mouse deviceEvent >>
         return theLineCCE

  Guiding Knowledge:
    all cases of 'case elementHit' must return the same type
    the 'Just cce' case returns (the new line's) CompositeContainerElt
    in the Nothing case and the '_' case, there is no new line
      so the existing line's CompositeContainerElt is the one to return

  Obstacles:
    knowing that the old line CompositeContainerElt is the one to return
      might be tricky
      (especially in the '_ -> sendMouseEv' case
        where the HU might not know what sendMouseEv means)
    the HU must know that sendMouseEv returns IO ()
      in order to know that it can be chained safely using >>
    this step is due to the "unintended consequence of a change" problem
    the HU must know and remember to revise the Nothing case (author didn't)
    the HU must know and remember to revise the '_' case (author didn't)
    this step casts serious doubt on the use of parameters to store state
      (see step 132)
      since bundling the line CompositeContainerElt in with the Gauge
        would be tedious
        but would avoid this problem

'Do' Step 134:
  change 'Nothing -> return ()' in trackCtrl to
    Nothing -> return theLineCCE
  change '_ -> sendMouseEv mouse deviceEvent' to
    _ -> sendMouseEv mouse deviceEvent
         return theLineCCE

  Guiding Knowledge:
    all cases of 'case elementHit' must return the same type
    the 'Just cce' case returns (the new line's) CompositeContainerElt
    in the Nothing case and the '_' case, there is no new line
      so the existing line's CompositeContainerElt is the one to return

  Obstacles:
    knowing that the old line CompositeContainerElt is the one to return
      might be tricky
      (especially in the '_ -> sendMouseEv' case
        where the HU might not know what sendMouseEv means)
    the HU must know that sendMouseEv returns IO ()
      in order to know that it can be followed by another statement
      without putting <- to its left also
    this step is due to the "unintended consequence of a change" problem
    the HU must know and remember to revise the Nothing case (author didn't)
    the HU must know and remember to revise the '_' case (author didn't)
    this step casts serious doubt on the use of parameters to store state
      (see step 132)
      since bundling the line CompositeContainerElt in with the Gauge
        would be tedious
        but would avoid this problem
```

```
Step 135:
  add CompositeContainerElt to the type signature of trackCtrl

  Guiding Knowledge:
    trackCtrl's type signature must match its definition

  Obstacles:
    this is an easy step to forget
    this should be no problem (once remembered)

Step 136:
  infer that the 'Just cce' case needs to split into
    a case for when the line is clicked on
    a case for when one of the points is clicked on

  Guiding Knowledge:
    the line is identified by a CompositeContainerElt
    the 'Just cce' case is where the CompositeContainerElt is detected/bound
    the 'Just cce' case cannot be made into a pattern match

  Obstacles:
    the fact that case expressions can't pattern match could be tricky
      but is not part of programming interaction (nor is it complicated by it)
    the HU might be tempted to write the cases as
      Just theLineCCE -> ...
      Just cce -> ...
    but that won't work

Step 137:
  modify the 'Just cce' case to read
    Just cce
      | cce == theLineCCE ->
        return theLineCCE
      | True ->
  and indent the rest of the lines in the 'Just cce' case one level

  Guiding Knowledge:
    to specify alternate expressions for different values
      use a guarded expression
    if the line is clicked on, return without doing anything
    the case expression must return the CompositeContainerElt
      for the current line

  Obstacles:
    the HU must know how to use guarded expressions
      including how to use True as the default case
    knowing to return theLineCCE is just like returning it in the Nothing case
      so it's reasonable that the HU would do so in this case

Step 138:
  infer that adding the label to display the correlation consists of
    creating the label (displaying the correlation)
    updating the label when a point is clicked

  Guiding Knowledge:
    the label must be created
    the label must initially display the correct correlation
    the label must be updated when the correlation changes
    the correlation changes each time a point is clicked

  Obstacles:
    this should be no problem

Step 139:
  in tracker, after drawLine, add
    label (show correlation) dc >>= \ (theLabel, labelDH) ->

  Guiding Knowledge:
    label creates a label (given a string to display and a DisplayContext)
    (show correlation) is a string consisting of the correlation coefficient
    label returns a label and a DisplayHandle for the label

  Obstacles:
    the HU must know to not name 'labelDH' just 'dh'
      because it will shadow the 'dh' of the canvas
      which is needed for both catchMouseEv and for displaying
    this is pretty straightforward; it requires
      knowing how to create a label
```

```
      knowing how to use show to display a value
      knowing that component creators return a (component,displayhandle) pair

'Do' Step 139:
  in tracker, after drawLine, add
    (theLabel, labelDH) <- label (show correlation) dc

  Guiding Knowledge:
    label creates a label (given a string to display and a DisplayContext)
    (show correlation) is a string consisting of the correlation coefficient
    label returns a label and a DisplayHandle for the label

  Obstacles:
    the HU must know to not name 'labelDH' just 'dh'
      because it will shadow the 'dh' of the canvas
      which is needed for both catchMouseEv and for displaying
    this is pretty straightforward; it requires
      knowing how to create a label
      knowing how to use show to display a value
      knowing that component creators return a (component,displayhandle) pair

Step 140:
  infer that labelDH must be returned with (canvas) dh in order to be displayed

  Guiding Knowledge:
    the DisplayHandle determines the display of the component
    the 'dh' of canvas is passed to realiseDH

  Obstacles:
    knowing this could be tricky
    the HU must know that DisplayHandles determine appearance
    the HU must understand how 'dh'
      is returned by tracker and passed to realiseDH

Step 141:
  in tracker, replace 'return dh' with
    return (vbox [labelDH, dh])

  Guiding Knowledge:
    vbox combines a list of DisplayHandles into a single DisplayHandle
    the return value of tracker needs to be IO DisplayHandle
    vbox stacks its parameters on top of each other
    putting the label on top avoids screwing up the coordinate system

  Obstacles:
    the HU needs to know how to use vbox to combine DisplayHandles
    the HU needs to know to put the label on top
      but this may just be a Haggis bug

Step 142:
  infer that the label needs to be passed to trackCtrl
    for it to be updated

  Guiding Knowledge:
    trackCtrl is where the clicks are detected
      and where the line is updated

  Obstacles:
    this should be no problem given previous analogous steps

Step 143:
  modify
    the trackCtrl call in tracker (forkIO)
    the trackCtrl definition
    the trackCtrl recursive call
  to take theLabel as an additional parameter
  modify the trackCtrl type signature
  to take Label as an additional parameter

  Guiding Knowledge:
    adding a parameter to a function includes changing
      the type signature
      the definition
      all calls of the function (including recursive calls)

  Obstacles:
    this should be no problem

Step 144:
```

```
  in trackCtrl, after setGauge, add
    setLabel theLabel (show theNewCorrelation) >>

  Guiding Knowledge:
    the label must be set to the current correlation
    setLabel sets a label to a value (string)
    (show theNewCorrelation) is a string containing the new correlation
    setLabel doesn't return a value, so it chains with >>

  Obstacles:
    this should be ok given previous steps such as 104 and 139

'Do' Step 144:
  in trackCtrl, after setGauge, add
    setLabel theLabel (show theNewCorrelation)

  Guiding Knowledge:
    the label must be set to the current correlation
    setLabel sets a label to a value (string)
    (show theNewCorrelation) is a string containing the new correlation
    setLabel doesn't return a value, so it doesn't need <-

  Obstacles:
    this should be ok given previous steps such as 104 and 139

Step 145:
  conclude that the Line-Fitting example is complete

  Guiding Knowledge:
    all of the parts of the problem statement have been completed
    the program runs; the line and correlation change when points are clicked

  Obstacles:
    this should be no problem
```

B.2 Starting Examples for Line-Fitting Walkthrough in Haggis

B.2.1 Combinator Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  mkDC ["*title: Canvas"]   >>= \ dc ->
    let
      tracker :: IO DisplayHandle
      tracker =
        compositeContainer (Just (200,200)) dc >>= \ (canvas,dh) ->
        catchMouseEv dh                        >>= \ (mouse,dh) ->
        forkIO (trackCtrl mouse canvas)        >>
        return dh

      r = 10

      newElement :: IO DisplayHandle
      newElement =
        transparentGlyph (withColour blue $ fillSolid $ circle r) dc >>= \ (_,dh) ->
        return (resizer dh)

      trackCtrl :: Mouse -> CompositeContainer -> IO ()
      trackCtrl mouse canvas =
        getMouseDown mouse >>=
                \ deviceEvent@(DevEv _ x y _ (MouseButton Down whichButton)) ->
        (case whichButton of
          2 ->
            newElement                              >>= \ dh ->
            placeTransparent canvas (x,y) dh        >>
            return ()
          1 ->
            hitCompositeContainerElt canvas (x,y)  >>= \ elementHit ->
            case elementHit of
              Nothing  -> return ()
              Just cce ->
                toFrontCompositeContainerElt cce   >>
```

```
            getCompositeContainerEltCoord cce  >>= \ (a,b) ->
            dragger canvas cce mouse (x-a,y-b)
        _ ->
           sendMouseEv mouse deviceEvent
       ) >>      -- end of case expression
       trackCtrl mouse canvas

  dragger :: CompositeContainer -> CompositeContainerElt
                                -> Mouse
                                -> Translation
                                -> IO ()
  dragger canvas cce mouse offset@(offset_x,offset_y) =
    getMouseEv mouse >>= \ deviceEvent@(DevEv _ x y _ whichEvent) ->
    case whichEvent of
      Motion ->
        moveCompositeContainerElt cce (x - offset_x, y - offset_y) >>
        dragger canvas cce mouse offset
      MouseButton Up _ ->
        sendMouseEv mouse deviceEvent
      _ ->
         sendMouseEv mouse deviceEvent >>
         dragger canvas cce mouse offset

in
  tracker            >>= \ dh ->
  realiseDH dc dh    >>
  return ()
```

B.2.2 do Notation

```
module Main(main) where

import Concurrent
import Haggis

main =
  do
    dc <- mkDC ["*title: Canvas"]
    let
      tracker :: IO DisplayHandle
      tracker =
        do
          (canvas,dh) <- compositeContainer (Just (200,200)) dc
          (mouse,dh)  <- catchMouseEv dh
          forkIO (trackCtrl mouse canvas)
          return dh

      r = 10

      newElement :: IO DisplayHandle
      newElement =
        do
          (_,dh) <- transparentGlyph (withColour blue $ fillSolid $ circle r) dc
          return (resizer dh)

      trackCtrl :: Mouse -> CompositeContainer -> IO ()
      trackCtrl mouse canvas =
        do
          deviceEvent@(DevEv _ x y _ (MouseButton Down whichButton)) <- getMouseDown mouse
          (case whichButton of
            2 ->
              do
                dh <- newElement
                placeTransparent canvas (x,y) dh
                return ()
            1 ->
              do
                elementHit <- hitCompositeContainerElt canvas (x,y)
                case elementHit of
                  Nothing  -> return ()
                  Just cce ->
                    do
                      toFrontCompositeContainerElt cce
                      (a,b) <- getCompositeContainerEltCoord cce
                      dragger canvas cce mouse (x-a,y-b)
            _ ->
               sendMouseEv mouse deviceEvent
```

```
        )         -- end of case expression
        trackCtrl mouse canvas

    dragger :: CompositeContainer -> CompositeContainerElt
                                  -> Mouse
                                  -> Translation
                                  -> IO ()
    dragger canvas cce mouse offset@(offset_x,offset_y) =
      do
        deviceEvent@(DevEv _ x y _ whichEvent)  <- getMouseEv mouse
        case whichEvent of
          Motion ->
            do
              moveCompositeContainerElt cce (x - offset_x, y - offset_y)
              dragger canvas cce mouse offset
          MouseButton Up _ ->
            sendMouseEv mouse deviceEvent
          _ ->
            do
              sendMouseEv mouse deviceEvent
              dragger canvas cce mouse offset

  in
    do
      dh <- tracker
      realiseDH dc dh
      return ()
```

B.3 Final Line-Fitting Examples in Haggis

B.3.1 Combinator notation

```
module Main(main) where

import Concurrent
import Haggis
import MiscIO
import FittedLineAndCorrelation

-- fittedLineAndCorrelation :: [(Int,Int)] -> ((Int,Int),(Int,Int))
--                                         -> (((Int,Int),(Int,Int)), Float)

main =
  mkDC ["*title: Line-Fitting"]   >>= \ dc ->
    let
      tracker :: IO DisplayHandle
      tracker =
        gauge (zip problems (repeat True))              >>= \ theDataGauge ->
        compositeContainer (Just (200,200)) dc          >>= \ (canvas,dh) ->
        plotPoints canvas problems                      >>
        drawLine canvas theLineCoords                   >>= \ theLineCCE ->
        label (show correlation) dc                     >>= \ (theLabel, labelDH) ->
        catchMouseEv dh                                 >>= \ (mouse,dh) ->
        forkIO (trackCtrl mouse canvas theDataGauge theLineCCE theLabel) >>
        return (vbox [labelDH,dh])

        where
          (theLineCoords,correlation) =
            fittedLineAndCorrelation problems ((0,0),(100,100))

      r = 5

      newElement :: Colour -> IO DisplayHandle
      newElement c =
        transparentGlyph (withColour c $ fillSolid $ circle r) dc >>= \ (_,dh) ->
        return dh

      plotPoint :: CompositeContainer -> Colour
                                      -> Coord
                                      -> IO CompositeContainerElt
      plotPoint canvas colour coord =
        newElement colour                           >>= \ dh ->
        placeTransparent canvas coord dh

      plotPoints :: CompositeContainer -> [Coord] -> IO [CompositeContainerElt]
      plotPoints canvas = mapIO $ plotPoint canvas blue
```

```
    drawLine :: CompositeContainer -> (Coord,Coord) -> IO CompositeContainerElt
    drawLine canvas ((x1,y1),(x2,y2)) =
      transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc >>= \ (_,dh) ->
      placeTransparent canvas (0,0) dh                           >>= \ lineCCE ->
      toBackCompositeContainerElt lineCCE                        >>
      return lineCCE

    trackCtrl :: Mouse
              -> CompositeContainer
              -> Gauge [(Coord,Bool)]
              -> CompositeContainerElt
              -> Label
              -> IO ()
    trackCtrl mouse canvas theDataGauge theLineCCE theLabel =
      getMouseDown mouse >>=
              \ deviceEvent@(DevEv _ x y _ (MouseButton Down whichButton)) ->
      (case whichButton of
        1 ->
          hitCompositeContainerElt canvas (x,y)  >>= \ elementHit ->
          case elementHit of
            Nothing  -> return theLineCCE
            Just cce
              | cce == theLineCCE ->
                return theLineCCE
              | True ->
                getCompositeContainerEltCoord cce     >>= \ (a,b) ->
                getGauge theDataGauge                 >>= \ thePointData ->
                let
                  (theNewLineCoords,theNewCorrelation) =
                   fittedLineAndCorrelation (map fst (filter snd theNewPointData))
                                            theClippingWindow
                  theNewPointData = ((a,b),theNewStatus):otherPoints
                  theNewStatus = not (snd thePoint)
                  ([thePoint],otherPoints) =
                    partition (\ (coord,_) -> coord == (a,b))
                              thePointData
                in
                  setGauge theDataGauge theNewPointData      >>
                  setLabel theLabel (show theNewCorrelation) >>
                  deleteCompositeContainerElt cce            >>
                  deleteCompositeContainerElt theLineCCE     >>
                  plotPoint canvas (if theNewStatus
                                    then blue
                                    else yellow) (a,b)       >>
                  drawLine canvas theNewLineCoords         >>= \ theNewLineCCE ->
                  return theNewLineCCE
        _ ->
           sendMouseEv mouse deviceEvent >>
           return theLineCCE
      )                                     -- end of case expression
      >>= \ theNewLineCCE ->
      trackCtrl mouse canvas theDataGauge theNewLineCCE theLabel

  in
    tracker           >>= \ dh ->
    realiseDH dc dh   >>
    return ()

where
  problems :: [Coord]
  problems = [ (90, 96),
               (94, 80),
               (40, 50),
               (24, 20),
               (24, 32),
               (30, 42),
               (50, 46),
               (32, 38),
               (60, 70),
               (64, 64)  ]

  theClippingWindow = ((0,0),(100,100))
```

B.3.2 do Notation

```
module Main(main) where
```

```
import Concurrent
import Haggis
import MiscIO
import FittedLineAndCorrelation

-- fittedLineAndCorrelation :: [(Int,Int)] -> ((Int,Int),(Int,Int))
--                                         -> (((Int,Int),(Int,Int)), Float)

main =
  do
    dc <- mkDC ["*title: Line-Fitting"]
    let
      tracker :: IO DisplayHandle
      tracker =
        do
          theDataGauge        <- gauge (zip problems (repeat True))
          (canvas,dh)         <- compositeContainer (Just (200,200)) dc
          plotPoints canvas problems
          theLineCCE          <- drawLine canvas theLineCoords
          (theLabel, labelDH) <- label (show correlation) dc
          (mouse,dh)          <- catchMouseEv dh
          forkIO (trackCtrl mouse canvas theDataGauge theLineCCE theLabel)
          return (vbox [labelDH,dh])

        where
          (theLineCoords,correlation) =
            fittedLineAndCorrelation problems ((0,0),(100,100))

      r = 5

      newElement :: Colour -> IO DisplayHandle
      newElement c =
        do
          (_,dh) <- transparentGlyph (withColour c $ fillSolid $ circle r) d
          return dh

      plotPoint :: CompositeContainer -> Colour
                                      -> Coord
                                      -> IO CompositeContainerElt
      plotPoint canvas colour coord =
        do
          dh <- newElement colour
          placeTransparent canvas coord dh

      plotPoints :: CompositeContainer -> [Coord] -> IO [CompositeContainerElt]
      plotPoints canvas = mapIO $ plotPoint canvas blue

      drawLine :: CompositeContainer -> (Coord,Coord) -> IO CompositeContainerElt
      drawLine canvas ((x1,y1),(x2,y2)) =
        do
          (_,dh)  <- transparentGlyph (withColour red $ line (x1,y1) (x2,y2)) dc
          lineCCE <- placeTransparent canvas (0,0) dh
          toBackCompositeContainerElt lineCCE
          return lineCCE

      trackCtrl :: Mouse
                -> CompositeContainer
                -> Gauge [(Coord,Bool)]
                -> CompositeContainerElt
                -> Label
                -> IO ()
      trackCtrl mouse canvas theDataGauge theLineCCE theLabel =
        do
          deviceEvent@(DevEv _ x y _ (MouseButton Down whichButton)) <- getMouseDown mouse
          theNewLineCCE <- (case whichButton of
            1 ->
              do
                elementHit <- hitCompositeContainerElt canvas (x,y)
                case elementHit of
                  Nothing  -> return theLineCCE
                  Just cce
                    | cce == theLineCCE ->
                      return theLineCCE
                    | True ->
                      do
                        (a,b)        <- getCompositeContainerEltCoord cce
                        thePointData <- getGauge theDataGauge
                        let
```

```
                        (theNewLineCoords,theNewCorrelation) =
                           fittedLineAndCorrelation (map fst (filter snd theNewPointData))
                                                    theClippingWindow
                        theNewPointData = ((a,b),theNewStatus):otherPoints
                        theNewStatus = not (snd thePoint)
                        ([thePoint],otherPoints) =
                          partition (\ (coord,_) -> coord == (a,b))
                                    thePointData
                      in
                        do
                          setGauge theDataGauge theNewPointData
                          setLabel theLabel (show theNewCorrelation)
                          deleteCompositeContainerElt cce
                          deleteCompositeContainerElt theLineCCE
                          plotPoint canvas (if theNewStatus
                                            then blue
                                            else yellow) (a,b)
                          theNewLineCCE <- drawLine canvas theNewLineCoords
                          return theNewLineCCE
                    _ ->
                      do
                        sendMouseEv mouse deviceEvent
                        return theLineCCE
                )                                    -- end of case expression
                trackCtrl mouse canvas theDataGauge theNewLineCCE theLabel

        in
          do
            dh <- tracker
            realiseDH dc dh
            return ()

      where
        problems :: [Coord]
        problems = [ (90, 96),
                     (94, 80),
                     (40, 50),
                     (24, 20),
                     (24, 32),
                     (30, 42),
                     (50, 46),
                     (32, 38),
                     (60, 70),
                     (64, 64)  ]

        theClippingWindow = ((0,0),(100,100))
```

B.3.3 Fitted Line and Correlation Calculation

```
module FittedLine (fittedLineAndCorrelation) where

fittedLineAndCorrelation :: [(Int,Int)] -> ((Int,Int),(Int,Int))
                                        -> (((Int,Int),(Int,Int)), Float)

fittedLineAndCorrelation theCoordinateList theClippingWindow =
  (((round x1, round y1), (round x2, round y2)), correlation)

  where
    ((x1, y1), (x2, y2)) =
       clipToRectangle (slope, intercept)
                       ((\ ((a,b),(c,d)) -> ((fromIntegral a, fromIntegral b),
                                             (fromIntegral c, fromIntegral d)))
                        theClippingWindow)

    (slope, intercept, correlation) =
      slopeInterceptAndCorrelation $
              zip (map (fromIntegral . fst) theCoordinateList)
                  (map (fromIntegral . snd) theCoordinateList)

    slopeInterceptAndCorrelation :: [(Float, Float)] -> (Float, Float, Float)
    slopeInterceptAndCorrelation xys = (slope, intercept, correlation)
      where
        intercept =
          (sum ys) / (fromIntegral n) - slope * (sum xs) / (fromIntegral n)
        slope = sxy / sxx
        correlation = sxy / sqrt (sxx * syy)
        sxx = sum (map square xs) - (square (sum xs)) / (fromIntegral n)
```

```
    syy = sum (map square ys) - (square (sum ys)) / (fromIntegral n)
    sxy = sum (zipWith (*) xs ys) - (sum xs) * (sum ys) / (fromIntegral n)
    square n = n * n
    (xs, ys) = unzip xys
    n = length xys

clipToRectangle ::  (Float, Float) -> ((Float, Float), (Float, Float))
                                   -> ((Float, Float), (Float, Float))
-- line is (slope, intercept)
-- rectangle is ((left, bottom), (right, top))
-- returned line segment is ((x1, y1), (x2, y2))  with  y1 <= y2

clipToRectangle (slope, intercept) ((left, bottom), (right, top))
  {- given that y1 <= y2 -}
  | y1 > top                    = ((0.0, 0.0), (0.0, 0.0))          {- no line -}
  | y1 < bottom && y2 < bottom = ((0.0, 0.0), (0.0, 0.0))          {- no line -}
  | y1 < bottom && y2 > top     = (( (bottom - intercept) / slope, {-  both   -}
                                    bottom),
                                  ( (top - intercept) / slope,
                                    top))

  | y1 < bottom                 = (( (bottom - intercept) / slope, {- y1 only -}
                                    bottom),
                                  ( x2,
                                    y2))

  | y2 > top                    = (( x1,                           {- y2 only -}
                                    y1),
                                  ( (top - intercept) / slope,
                                    top))
  | otherwise                   = ((x1, y1), (x2, y2))             {- neither -}

  where (x1, y1, x2, y2)  | slope >= 0  = (left,
                                           slope * left + intercept,
                                           right,
                                           slope * right + intercept)
                          | otherwise   = (right,
                                           slope * right + intercept,
                                           left,
                                           slope * left + intercept)
```

Appendix C

Temperature Conversion Walkthrough in Excel with Visual Basic

A screen image of the final temperature conversion example in Excel with Visual Basic appears in figure 3.1.

C.1 Temperature Conversion Walkthrough in Microsoft Excel Visual Basic

```
Note: this walkthrough is for a solution using scrollbars.  The
scrollbars, in addition to not looking much like thermometers, show
temperatures increasing downwards.


Step 1:
  decide to use spreadsheet cells for the numeric editable displays

  Guiding Knowledge:
    spreadsheet cells display data and allow it to be edited

  Obstacles:
    hard to imagine for someone with spreadsheet knowledge
    HU might use entry boxes instead,
      which is feasible, but more work

Step 2:
  consider and discard the idea of using formulas to link the
    two editable numeric display cells

  Guiding Knowledge:
    formulas are used to link cells in spreadsheets
    values in cells containing formulas cannot be edited

  Obstacles:
    reasonable
      most spreadsheet users would know the GK
      any attempt to use formulas would quickly fail

Step 3:
  decide to use scrollbars to graphically display and change the temperatures

  Guiding Knowledge:
    scrollbars display and afford manipulation of a value within a range
    scrollbars can be linked to cells, so that
      both display the same value
      changing either changes the other
    no other widgets look reasonable
    graphics (such as rectangles) cannot be dragged
      nor can the location of a click be determined
    a disadvantage:
      (vertical scrollbars' values increase downwards, unfortunately)

  Obstacles:
    scrollbars are not the "obvious" right answer to thermometers
    the weaknesses of graphic objects (dragging, clicking)
      requires deep system knowledge

Step 4:
  divide the problem into three steps
```

```
    choosing the layout of the scrollbars and numeric cells
    displaying the scrollbars
    connecting the scrollbars and cells

  Guiding Knowledge:
    need to choose where cells and scrollbars will go
    need to create scrollbars somehow
    need to connect them to get desired behavior

  Obstacles:
    seems reasonable to create scrollbars before exploring how to link them
    in a direct-manipulation interface, creation includes placement

Step 5:
  further divide the last step above into
    linking each scrollbar to its numeric cell
    connecting the two scrollbar-cell pairs somehow

  Guiding Knowledge:
    linking scrollbars to cells accomplishes two of the necessary connections,
      leaving only one connection to be handled some other way

  Obstacles:
    it's likely that the HU, knowing about scrollbars,
      would know about linked cells
    [the program could be written with no links, simply OnAction and OnEntry
      but that would be much more tedious
      and the notion that the HU would know about OnAction and OnEntry
        and not linked cells is farfetched]

Step 6:
  realize that Visual Basic code will be needed

  Guiding Knowledge:
    linked cells only connect one cell with one scrollbar
    formulas cannot link the cells
    Visual Basic procedures can be triggered when
      a cell is edited
      a scrollbar is manipulated

  Obstacles:
    the realization that Visual Basic code is needed is important
    attempting write the program without Visual Basic code
      could waste considerable time and effort

Step 7:
  summarize steps of the problem solution as
    choosing the layout of the scrollbars and numeric cells
    displaying the scrollbars
    linking each scrollbar to its numeric cell
    connecting the two scrollbar-cell pairs with Visual Basic procedures

  Guiding Knowledge:
    the third original step was divided,
      and its second part requires the use of Visual Basic procedures

  Obstacles:
    none -- it's just a summary of previous steps

Step 8:
  infer that part, but not all, of the problem can be done
    by direct manipulation

  Guiding Knowledge:
    cells can be modified by direct manipulation
    components such as scrollbars can be laid out with direct manipulation
    linking cells to scrollbars can be done with direct manipulation
    connecting the two scrollbar-cell pairs requires Visual Basic

  Obstacles:
    trying to do more than necessary in Visual Basic will take time and effort
    the HU could possibly assume that programming is required for all parts

Step 9:
  notice that having the entire solution in Visual Basic may be simpler

  Guiding Knowledge:
    having all functionality originating in one place is conceptually easier
    Visual Basic functionality must be reestablished when a document is opened
```

```
    additions or changes to the spreadsheet may change references
      in Visual Basic
    having the entire solution in Visual Basic
      puts all of the functionality in one place
      allows recreation of the solution easily
      avoids possibly ambiguous connections with the spreadsheet side

  Obstacles:
    creating the scrollbars by direct manipulation,
      then setting up OnAction calls for them
      is substantially more complicated
      than using Visual Basic to create and bind the scrollbars
      because the newly created scrollbar is also the Selection
      and can be addressed as such, instead of ActiveSheet.Scrollbars(1)
        the latter case runs the risk of the scrollbars getting bound backwards
    however, recognizing these potential complications ahead of time is tricky

Step 10:
  notice that creating part of the solution by direct manipulation is easier

  Guiding Knowledge:
    direct manipulation, when compared with writing programs,
      requires less cognitive effort
      takes fewer steps

  Obstacles:
    presumably, the HU would know that DM is easier than programming
      but she/he might not think about it (see step 8)

Step 11:
  decide to record the direct-manipulation portion of the solution as a macro
    then use the macro recording as a starting point for the full solution

  Guiding Knowledge:
    macro recording creates programs from direct manipulations
    a program created by macro recording can be a template or starting point
    recording a macro and editing it to create a program
      puts all the functionality into the program
      is easier than writing a program from scratch

  Obstacles:
    the HU must know about macro recording, and about editing macros
      otherwise, writing the program will be tremendously more work

Step 12:
  turn on macro recording by:
    selecting Record New Macro... from the Record Macro submenu
      of the Tools menu
    clicking Ok in the dialog box

  Guiding Knowledge:
    macro recording is turned on and off via the Tools menu

  Obstacles:
    should be ok -- just basic (Mac) interface usage

Step 13:
  proceed to the first major step (layout), and
  infer that each number should be closest to its scrollbar/thermometer

  Guiding Knowledge:
    put related or mutually dependent entities near each other for clarity

  Obstacles:
    should be ok

Step 14:
  decide to put the scrollbars in column B, with the numbers in A and C

  Guiding Knowledge:
    the bars should be side by side (problem statement)
      so it makes sense to put them in a column
    each number must go in a cell, and thus in a column
    the numbers should go next to their respective bars

  Obstacles:
    lots of similar solutions should work fine, too

Step 15:
```

```
   decide to create the scrollbars using the Forms toolbar

  Guiding Knowledge:
    a make-scrollbar button is on the Forms toolbar

  Obstacles:
    this is something HU must know
      to create the scrollbars by direct manipulation
      and possibly, to know that scrollbars are available at all

Optional Step (if Forms toolbar is not displayed):
  display the Forms toolbar by
    selecting Toolbars... from the View menu
    clicking the Forms checkbox in the scrollable list
    clicking Ok in the dialog box

  Guiding Knowledge:
    the Toolbars... option on the View menu controls display of toolbars
    clicking the checkbox next to the name of a toolbar will display it

  Obstacles:
    displaying the toolbar should be easy for an experienced interface user
    knowing how to display toolbars is slightly more sophisticated
    most critically, the absence of the Forms toolbar
      (not displayed by default) could contribute to
      the HU not knowing that scrollbars are available
      (although the scrollbar-creator button graphic is hardly lucid)

Step 16:
  create the Fahrenheit scrollbar by
    clicking on the scrollbar button
    dragging from the top of column B to the bottom along the left side

  Guiding Knowledge:
    graphic objects and components are created by
      clicking the appropriate toolbar button and
      dragging out the desired (rectangular) size and shape of the object

  Obstacles:
    the scrollbar-creator button's graphic is minimally clear
    dragging out a scrollbar should be no problem for an experienced XL user

Optional Step (if newly created scrollbar is deselected):
  select the Fahrenheit scrollbar by command-clicking on it

  Guiding Knowledge:
    active objects such as scrollbars must be selected with command-click,
      because they respond to clicks (that do not happen with modifier keys)

  Obstacles:
    one of those things that you just have to know

Step 17:
  link the Fahrenheit scrollbar to cell $A$30, by
    selecting Object... from the Format menu
    selecting the Controls tab
    typing '$A$30' into the Linked Cell box
    clicking the Ok button

  Guiding Knowledge:
    the linked cell of a scrollbar can be specified
      via the Controls tab in the Format Object dialog
    the Format Object dialog is available via Object... on the Format menu
    $A$30 is the absolute reference to cell A30

  Obstacles;
    HU either knows how to link via direct manipulation or doesn't
    linking a cell and scrollbar together in Visual Basic isn't difficult,
      but this action records (in the macro)
        how to use the Selection to set attributes
        and that is a major source of information for the HU
        that can save a major headache
    absolute references are important in some contexts and not others
      knowing what they are and when to use them is important
    typing '=$A$30' works, as do 'Sheet1!A30' and 'Sheet1!$A$30'
      formula line shows (e.g.) =$A$30 not $A$30
    in other words, there are multiple ways to specify the linked cell
      and no clear indication of which is best
      or how they differ in this situation
```

```
Step 18:
  create the Celsius scrollbar in the same fashion as the Fahrenheit scrollbar,
    (as in step 16) but along the right side of column B

  Guiding Knowledge:
    same as in creating the Fahrenheit scrollbar (see step 16)
    the choice of left and right is arbitrary,
      as long as each number cell is next to its scrollbar

  Obstacles:
    see step 16
    the second time should present no difficulty, given the first experience

(Optional Step above (after step 16) is applicable here also)

Step 19:
  link the Celsius scrollbar to cell $C$30
    just as the Fahrenheit scrollbar linked to cell $A$30

  Guiding Knowledge:
    same as with the linking of the Fahrenheit scrollbar to $A$30

  Obstacles:
    see step 17

Step 20:
  turn off macro recording
    by selecting Stop Recording on the Record Macro submenu of the Tools menu
    or by clicking the Stop (square) button on the floating toolbar

  Guiding Knowledge:
    turning off macro recording avoids undesired recording
    macro recording is turned on and off via the Tools menu
      or can be turned off by the Stop button on the floating toolbar
        that appears when macro recording is turned on

  Obstacles:
    remembering to turn off macro recording is tricky
    actually turning it off should be easy
    failing to turn it off means more (mostly mess) to sort through

Step 21:
  look at the macro by (scrolling the tabs and) selecting the Module 1 tab

  Guiding Knowledge:
    a new macro is stored in a newly created Module
    clicking the tab of a sheet or module displays it in the window
    scrolling the line of tabs may be necessary to see all tabs

  Obstacles:
    should be ok for Excel user who knows about sheets and workbooks


The macro is

'
' Macro1 Macro
' Macro recorded 3/21/98
'
'
Sub Macro1()
    ActiveSheet.ScrollBars.Add(74, 2, 15, 649).Select
    With Selection
        .Value = 0
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
        .LinkedCell = "$A$30"
        .Display3DShading = True
    End With
    ActiveSheet.ScrollBars.Add(131, 2, 15, 648).Select
    With Selection
        .Value = 0
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
```

```
        .LinkedCell = "$C$30"
        .Display3DShading = True
    End With
End Sub


(note that macro will have additional statments
  if the various optional steps above are taken
  or if the HU does additional direct manipulations while recording)


Step 22:
  infer that the ActiveSheetScrollBars.Add lines
    match the creation of the scrollbars

  Guiding Knowledge:
    the first line of the macro is one of these,
      and the first action was to draw a scrollbar
    there are two lines, and two scrollbars were created
    'Add' suggests creation
    the documentation says that Scrollbars.Add(x,y,w,h) creates a scrollbar

  Obstacles:
    the .Select is potentially confusing,
      since it specifies an attribute of a newly-created object
    even though the documentation is probably necessary
      for understanding ...ScrollBars.Add()
      the macro helps the HU find the appropriate documentation

Step 23:
  infer that the rest of the macro must record the cell linking

  Guiding Knowledge:
    only four actions were performed when the macro was being recorded
        creating the Fahrenheit scrollbar
        linking it to $A$30 using the Controls tab of the Format Object dialog
        creating the Celsius scrollbar
        linking it to $C$30 using the Controls tab of the Format Object dialog
    the two creation steps correspond to the two .Add statements

  Obstacles:
    the fact that the other attributes are gratuitously set to their defaults
      provides significantly more code to sift through

Step 24:
  infer that the With Selection ... End With sequences
    correspond to the fields on the Control tab
    in the scrollbar's Format Object dialog

  Guiding Knowledge:
    these sequences immediately follow the .Add statements
      and the Format Object dialog was opened for each scrollbar
      immediately after its creation
    the scrollbar creation actions correspond to the .Add statements
    the names
        .Value, .Min, .Max, .SmallChange, .LargeChange, .LinkedCell
      roughly match the entries
        Current Value, Maximum Value, Minimum Value,
        Incremental Change, Page Change, Cell Link
      in the dialog box
      (there's no dialog entry that matches the .Display3DShading name)
    the values in the dialog box (0, 0, 100, 1, 10)
      match those recorded by the macro
    the Cell Link box was modified by direct manipulation to $A$30 ($C$30)
      and the macro matches this exactly
    (of course, there is no value match to the .Display3DShading value)

  Obstacles:
    slight mismatches between the Visual Basic and dialog box names
      make this step a bit less obvious
    also, the macro doesn't record that the dialog box was opened,
      just that the values were changed (even though most weren't)

Step 25:
  infer that With Selection means that the selection's properties
    are being set by the With sequence

  Guiding Knowledge:
    the properties must belong to some object
```

```
    the object is named in the With <object> part of the With statement

  Obstacles:
    although the Selection is a convenient way to set properties
      it may be less obvious as an object
      than the more "persistent" objects such as ActiveSheet.Scrollbars(1)

Step 26:
  infer that .Select in the .Add statement selects the newly created scrollbar

  Guiding Knowledge:
    the scrollbar was selected after it was drawn
    the With Selection ... End With statement appears to change the scrollbar,
      but it changes the selection
      which suggests that the scrollbar is selected

  Obstacles:
    the .Select specifies an action (selection)
      upon a newly created object (scrollbar)
      which could be hard to infer smoothly from the syntax

Step 27:
  conclude that the macro recorded the direct-manipulation actions as:
    the .Add statement created the scrollbar
    the .Select part of that statement selected it
    the With Selection ... End With sequence linked the cell

  Guiding Knowledge:
    this interpretation is highly consistent with the sequence of actions
      and with the sequence of observations of the interface display

  Obstacles:
    this seems reasonable based on previous steps

Step 28:
  decide to set the Fahrenheit scrollbar value to 32 in the macro

  Guiding Knowledge:
    .Value corresponds to the value of the scrollbar
    if the Celsius bar is at 0, the Fahrenheit bar should be at 32
      because 0 C = 32 F

  Obstacles:
    this should be fine based on previous steps

Step 29:
  change the .Value statement in the first With Selection ... End With
    to read .Value = 32

  Guiding Knowledge:
    the first With Selection ... End With corresponds to the first scrollbar
    the first scrollbar created was the Fahrenheit scrollbar

  Obstacles:
    small chance of setting the wrong one
    note that min and max are NOT changed
      this is important, because setting matching ranges
      (for example F in [32..212], C in [0..100])
      will result in both scrollbars looking identical
      so having min and max correspond would be a mistake

Step 30:
  change the name of the macro from Macro1 to SetUpThermometers
    in the first line and in the header
  add a comment 'Draw Fahrenheit Scrollbar before the first .Add statement
  and a comment 'Draw Celsius Scrollbar before the second .Add statement

  Guiding Knowledge:
    the name of a macro should reflect what it does
    the header should contain the name
    the name of the macro appears in the first line Sub <macroname> ()
    the name Macro1 is also in the comment header above the first line
    the comments clarify which statements are for which scrollbars

  Obstacles:
    HU might not bother to do this, but it's all quite reasonable

Step 31:
  conclude that the macro SetUpThermometers will
```

```
    create two scrollbars with values of 32 and 0
    and link them to A30 and C30

  Guiding Knowledge:
    the macro was recorded by creating the scrollbars
      and linking them to the cells
    the only changes the macro have been
      the initial value of the first (Fahrenheit) scrollbar
      renaming the macro and adding comments

  Obstacles:
    should be ok

Step 32:
  recall that the final major step is to connect the two scrollbar-cell pairs
    using Visual Basic procedures

  Guiding Knowledge:
    the macro recorded the first three steps
        laying out the scrollbars and cells
        creating the scrollbars
        linking the scrollbars to their respective cells
      thus only the original fourth step remains
    connecting the two scrollbar-cell pairs can be done only with Visual Basic

  Obstacles:
    should be ok -- presumably HU still has sight of goal

Step 33:
  infer that the other scrollbar-cell pair will need to be updated both
    when the scrollbar moves
    and when the cell is edited

  Guiding Knowledge:
    Visual Basic procedures can only be triggered
      from direct user actions
      from recalculation
      in a few other cases (time, incoming data from another application)
    there is no recalculation in this spreadsheet
    the other cases don't apply
    the two possible actions are
      moving a scrollbar
      editing a cell

  Obstacles:
    HU might think that updating from scrollbar will be enough
      since each scrollbar will change when its cell changes
    knowing the full range of possibilities could be difficult
      because there isn't a simple unified model

Step 34:
  infer that the relevant ways of updating one pair from another are among
    OnAction, OnCalculate, OnData, OnDoubleClick, OnEntry, OnKey, OnRepeat,
    OnSheetActivate, OnSheetDeactivate, OnTime, OnUndo, OnWindow

  Guiding Knowledge:
    these are the available properties
      that can have attached Visual Basic procedures
      that are called when certain events occur

  Obstacles:
    knowing all this constitutes a sophisticated knowledge
      of how Visual Basic handles (responds to) interaction and other events

Step 35:
  infer that OnAction is the only one that can be triggered by a scrollbar

  Guiding Knowledge:
    the only On... property that scrollbars have is OnAction
    for scrollbars, OnAction is triggered whenever the scrollbar is
      moved by the mouse
    the On... properties are triggered only by user interactions
      or by spreadsheet recalculation (caused by user interaction)
      (or external events -- time, data from another application)

  Obstacles:
    again, extensive knowledge of Visual Basic is required
      to be sure that OnAction is the only option
    it is, however, the only On... property listed for scrollbars
```

```
      in the documentation

Step 36:
  decide to use OnAction to update the other pair when a scrollbar is moved

  Guiding Knowledge:
    the scrollbar must update its cell and the other scrollbar and cell
    the scrollbar is linked to its cell and thus need not update it
    updating the other scrollbar and cell must be via Visual Basic
    OnAction is the only scrollbar property
      that can start a Visual Basic procedure

  Obstacles:
    the HU really has no other options
    see previous steps

Step 37:
  infer that the place to set OnAction for a scrollbar is anywhere
    in the With Selection ... End With sequence for that scrollbar

  Guiding Knowledge:
    the With Selection ... End With sequence is
      where other scrollbar properties are set
    the order within the With is unimportant because
      no statement references anything in any other statement

  Obstacles:
    knowing that .OnAction is a property
      on the same level as .Value, etc
      is a significant, but realistic, piece of knowledge

Step 38:
  decide to have the scrollbar's OnAction procedure
    update the other cell, rather than the other scrollbar

  Guiding Knowledge:
    updating either is sufficient, because they are linked
    referring to the other cell is simpler
      than referring to the other scrollbar
      because cell addressing is straightforward and coordinate-based

  Obstacles:
    knowing to make this choice makes the code a little more transparent
      because scrollbar #1 could be anywhere on the sheet,
      but cells(30,1) is labeled on the sheet by being in row 30 and column A

Step 39:
  in the first With Selection ... End With sequence,
    add a line
      .OnAction = "UpdateCelsiusCell"

  Guiding Knowledge:
    .OnAction = "some action" in a With theObject ... End With
      sets the OnAction property of theObject to "some action"
    the value of OnAction needs to be the name of a Visual Basic procedure
    the Visual Basic procedure needs to update the Celsius cell
      so UpdateCelsiusCell is an appropriate name

  Obstacles:
    should be fine based on previous mental operations

Step 40:
  decide that the next step is to write the UpdateCelsiusCell procedure

  Guiding Knowledge:
    it needs to be written
    writing it is part of the task of having the Fahrenheit scrollbar
      update the Celsius cell and scrollbar
    completing a task is cognitively simpler than switching tasks
    there is no reason to switch tasks

  Obstacles:
    the HU might do step 45 now, which would also be fine
    doing this somehow seems to be better task focus (but maybe it's not)

Step 41:
  write a header and Sub UpdateCelsiusCell and End Sub

  Guiding Knowledge:
```

```
    the SetUpThermometers procedure is identical except for the name change

  Obstacles:
    copying a procedure structure
      should be easy for even an almost-novice programmer
      but might be tricky for a spreadsheet user writing her/his first program

Step 42:
  decide that the body of the procedure will be
      C = 5/9 * (F -32)
    where C is the Celsius cell and F is the Fahrenheit cell

  Guiding Knowledge:
    the Celsius cell needs to be updated
    the Celsius cell's value is computed from the Fahrenheit cell's value
      by the formula C = 5/9 * (F -32)
    a scrollbar's OnAction property is called after the linked cell is updated

  Obstacles:
    knowing that the linked cell is updated first is absolutely critical
      otherwise, the HU would have to get the value from the scrollbar
      which is not tremendously messy,
        but using the cells leverages the HU's knowledge of spreadsheets
    also, perhaps, the cell "feels" more like a variable than the scrollbar

Step 43:
  write the body of UpdateCelsiusCell:
    Cells(30, 3) = 5 / 9 * (Cells(30, 1) - 32)

  Guiding Knowledge:
    Cells(R, C) references the contents of the cell at row R and column C
    the Celsius cell is C30, that is, row 30 and column 3
    the Fahrenheit cell is A30, that is, row 30 and column 1

  Obstacles:
    HU could use Range($A$30) and Range($C$30)
      (here the absolute references are absolutely critical!!)
    knowing how Cells(row,col) works is core Visual Basic knowledge
    the (row, column) instead of (x,y) addressing could trip the HU up
      (even though it does match with A1-style addressing)

Step 44:
  write an analogous UpdateFahrenheitCell procedure
    with a body
      Cells(30, 1) = 9 / 5 * Cells(30, 3) + 32

  Guiding Knowledge:
    analogous to previous three steps
    the Fahrenheit-from-Celsius formula is
      F = 9/5*C + 32

  Obstacles:
    should be no problem based on previous two steps
    small chance of transposing the cells
      since Cells(30,1) and Cells(30,3) look a lot like each other
        and are not named mnemonically

Step 45:
  in the second With Selection ... End With sequence,
    add a line
      .OnAction = "UpdateFahrenheitCell"

  Guiding Knowledge:
    analogous to that for the addition of .OnAction to the first With

  Obstacles:
    should be no problem based on step 39

Step 46:
  decide that the next and final step is to find a means for cell editing
    to trigger update of other scrollbar-cell pair

  Guiding Knowledge:
    this is the only remaining functionality to implement

  Obstacles:
    none, although see step 33

Step 47:
```

```
  decide that OnEntry is the appropriate property to use

  Guiding Knowledge:
  the choices are
    OnAction, OnCalculate, OnData, OnDoubleClick, OnEntry, OnKey, OnRepeat,
    OnSheetActivate, OnSheetDeactivate, OnTime, OnUndo, OnWindow
  OnAction and OnKey are too general
  OnCalculate is irrelevant because the sheet has no formulas
  OnData is irrelevant because no data is linked to another application
  OnDoubleClick would require double-clicking, which is arbitrary
  OnRepeat and OnUndo implement menu options
  OnSheetActivate, OnSheetDeactivate, and OnWindow are irrelevant
    because they have to do with switching sheets and windows
  OnTime could work, but would not provide immediate feedback

  Obstacles:
    as in steps 34 and 35, knowing what is not appropriate
      requires extensive knowledge of Visual Basic's interaction primitives

Step 48:
  infer that the OnEntry property cannot be set for a single cell or range

  Guiding Knowledge:
    OnEntry is only available for Applications and Worksheets

  Obstacles:
    since a user typically edits one cell at a time
      (except in block pastes or array formulas)
      to have OnEntry apply only to worksheets and applications
        is a cognitive mismatch
    also, Application.Caller must be used (see step 51)

Step 49:
  conclude that the procedure to which OnEntry is set must
    determine which cell was edited
    update the other cell appropriately

  Guiding Knowledge:
    OnEntry property can't be set for a single cell
    updating the other cell is simpler than updating the other scrollbar
      (see similar step above)

  Obstacles:
    HU could get really stuck on the apparent paradox
      between the OnEntry function called with no parameters by the worksheet
      and the obvious need to determine which cell was edited
    this seeming impossibility offers no clues about how to proceed

Step 50:
  suspect that there must be some way to find out which cell was edited

  Guiding Knowledge:
    which cell was edited is central, a critical piece of information

  Obstacles:
    hopefully, the HU would pursue this suspicion long enough to succeed

Step 51:
  decide to use Application.Caller to find out which cell was changed

  Guiding Knowledge:
    the Caller property is used to find out how Visual Basic was called
    when Visual Basic is called by OnEntry, Caller holds the cell reference
    only the Application has the Caller property, hence Application.Caller

  Obstacles:
    Application.Caller is documented and reasonably easy to find
    its role, and the language design around it,
      is opaque and poorly motivated

Step 52:
  decide that the body of the OnEntry procedure will have the structure
    if <FahrenheitCellChanged> then
      ChangeCelsiusCell
    else if <CelsiusCellChanged> then
      ChangeFahrenheitCell

  Guiding Knowledge:
    if-then-else statements are used for choosing alternatives
```

```
  Obstacles:
    this seems reasonable knowledge for most programmers
      (including spreadsheet users who use IF(,,) formulas)
    integrating this with the Application.Caller incantation
      might be less obvious
    Note: this step is here rather than earlier because
      if OnEntry could be specified per cell,
      the existing UpdateCelsiusCell and UpdateFahrenheitCell could be used

Step 53:
  conclude that the OnEntry procedure will
    use Application.Caller to determine which cell was edited
    use if-then-else (see previous step) to update the other cell

  Guiding Knowledge:
    Application.Caller holds the reference of the cell that changed
    if-then-else will select the appropriate statement

  Obstacles:
    Application.Caller is strange enough that using it this way is nonobvious

Step 54:
  infer that <FahrenheitCellChanged> in the next-but-last step
    can be replaced by Application.Caller = Cells(30, 1)

  Guiding Knowledge:
    if the Fahrenheit cell changes,
      then its reference will be stored in Application.Caller
    the reference of the Fahrenheit cell is Cells(30, 1)

  Obstacles:
    knowing to use this substitution for <FahrenheitCellChanged>
      is potentially problematic

Step 55:
  infer that <CelsiusCellChanged> in the same step
    can be replaced by Application.Caller = Cells(30, 3)

  Guiding Knowledge:
    analogous to previous step

  Obstacles:
    should be no problem given previous step

Step 56:
  infer that ChangeCelsiusCell and ChangeFahrenheitCell in that step
    can be replaced by UpdateCelsiusCell and UpdateFahrenheitCell

  Guiding Knowledge:
    UpdateCelsiusCell and UpdateFahrenheitCell have already been written
      and they do just what ChangeCelsiusCell and ChangeFahrenheitCell specify

  Obstacles:
    the HU might duplicate the code and then notice (or not) the simplification

Step 57:
  conclude that the body of the OnEntry procedure will be
    If Application.Caller = Cells(30, 1) Then
      UpdateCelsiusCell
    Else
      If Application.Caller = Cells(30, 3) Then
        UpdateFahrenheitCell
      End If
    End If

  Guiding Knowledge:
    compiled from previous three steps
    this is the exact syntax of If Then Else in Visual Basic

  Obstacles:
    should be no problem to substitute inferences from steps 54,55,56
    If Then Else syntax in Visual Basic should pose no major problem
      especially given the editor's (minimal, but useful) feedback

Step 58:
  in SetUpThermometers, add a line at the end
    ActiveSheet.OnEntry = "UpdateCells"
```

```
  Guiding Knowledge:
    OnEntry can be set for the Application or a Worksheet
      we don't know which is best; ActiveSheet is where the scrollbars are
      (alternatively, Application may be the easiest to figure out)
    the OnEntry property needs to be set up initially
    SetUpThermometers does all the rest of the setting up,
      including setting up the OnAction properties
    statement ordering is unimportant
      conceptually, it seems natural to set up the On... properties last
    since the procedure called when a cell is entered can update either cell,
      UpdateCells seems like a reasonable name

  Obstacles:
    this should be ok given practice with setting .OnAction properties
      but it's still weird enough that it might be error-prone
    some possibility of setting .OnEntry to the wrong procedure,
      especially since only a body for UpdateCells exists

Step 59:
  write a header and Sub UpdateCells and End Sub
    around the body written two steps above

  Guiding Knowledge:
    framework is identical to the framework for the other Update... procedures

  Obstacles:
    none

Step 60:
  conclude that program is complete

  Guiding Knowledge:
    the OnEntry property will be set to a procedure (UpdateCells)
      (that will update one cell when the other changes)
    changing the other scrollbar-cell pair when a cell is edited
      was the last bit of functionality that the program needed

  Obstacles:
    knowing that program is complete might require a mental and visual check
```

C.2 Final Temperature Conversion Example in Excel Visual Basic

```
'
' SetUpThermometers
'
Sub SetUpThermometers()
    ActiveSheet.ScrollBars.Add(74, 2, 15, 649).Select
    With Selection
        .Value = 32
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
        .LinkedCell = "$A$30"
        .Display3DShading = True
        .OnAction = "UpdateCelsiusCell"
    End With
    ActiveSheet.ScrollBars.Add(131, 2, 15, 648).Select
    With Selection
        .Value = 0
        .Min = 0
        .Max = 100
        .SmallChange = 1
        .LargeChange = 10
        .LinkedCell = "$C$30"
        .Display3DShading = True
        .OnAction = "UpdateFahrenheitCell"
    End With
    ActiveSheet.OnEntry = "UpdateCells"
End Sub
'
'
' UpdateCelsiusCell
'
Sub UpdateCelsiusCell()
      Cells(30, 3) = 5 / 9 * (Cells(30, 1) - 32)
```

```
End Sub
'
'
' UpdateFahrenheitCell
'
Sub UpdateFahrenheitCell()
      Cells(30, 1) = 9 / 5 * Cells(30, 3) + 32
End Sub
'
'
' UpdateCells
'
Sub UpdateCells()
  If Application.Caller = Cells(30, 1) Then
    UpdateCelsiusCell
  Else
    If Application.Caller = Cells(30, 3) Then
      UpdateFahrenheitCell
    End If
  End If
End Sub
```

Appendix D

Line-Fitting Walkthrough in Excel Visual Basic

A screen image of the final line-fitting example in Excel with Visual Basic appears in figure 3.2.

D.1 Line-Fitting Walkthrough in Microsoft Excel Visual Basic

```
Step 1:
  list the x and y coordinates in A5:A14 and B5:B14 respectively
  put headers "x" and "y" in A4 and B4 respectively

  Guiding Knowledge:
    spreadsheet cells display data and allow it to be edited
    spreadsheet cells also are accessible to formulae
    first two columns are a reasonable location
    first few rows may be needed for headers
    everything can be moved later if need be

  Obstacles:
    spreadsheet is natural place to store data
    HU might not know that cells can be linked to objects
      and thus might think that cells won't be useful

Step 2:
  decide to store inlier/outlier status as a column of TRUE/FALSE cells
    in column C

  Guiding Knowledge:
    cells can contain Boolean values
    data in the spreadsheet can be used
      easily and effectively by spreadsheet functions
      less conveniently (but nevertheless effectively) by Visual Basic procedures

  Obstacles:
    this step and the next few have to occur all together

Step 3:
  put TRUE in each of the cells C5:C14

  Guiding Knowledge:
    initially, all points are inliers
    TRUE and FALSE are the Boolean values in Microsoft Excel

  Obstacles:
    trivial given step 2

Step 4:
  decide to compute (and "store") the inliers using =IF(<inlier?>,coord,"")

  Guiding Knowledge:
    this formula will leave the cell blank if the point is an outlier

  Obstacles:
    the HU simply has to have this insight
      and to know that the slope/intercept/correlation formula
        will span successfully over blank cells
      (the chart range won't!)
```

```
Step 5:
  place the formula =IF(C5,A5,"") in D5
    and the formula =IF(C5,B5,"") in E5
  and copy them down to D6:E14

  Guiding Knowledge:
    these formulas implement step 5
    formulas can be copied to ranges, and will keep relative references

  Obstacles:
    this is standard spreadsheet manipulation
    the formulas are a little unusual -- see step 4

Step 6:
  place the formula =SLOPE(E5:E14,D5:D14) in C17
    and the formula =INTERCEPT(E5:E14,D5:D14) in C18
    and the formula =CORRELATION(E5:E14,D5:D14) in C19
    and put the labels "Slope:", "Intercept:", and "Correlation:"
      in A17, A18, and A19 respectively
    define the names "slope" and "intercept" to be C17 and C18 respectively

  Guiding Knowledge:
    the slope and intercept are needed to calculate the fitted line
    the correlation needs to be displayed per the problem statement
    Excel provides functions for all three
    the functions take Y's first, then X's
    labels are always a good idea (sort of imitation naming)
    defining names is useful when formulas get complicated
    the bounding box formulas are complicatec

  Obstacles:
    confusing Y's and X's
    failing to define names (need arises in step 10)

Step 7:
  change C6 to FALSE to see if cell blanking and formulas work (they do)

  Guiding Knowledge:
    it's a good idea to check that ideas work as expected

  Obstacles:
    none

Step 8:
  decide to put the plot to the right of the data

  Guiding Knowledge:
    the plot is important, and so should be near the top
    the plot goes with the data, and should be near it

  Obstacles:
    none

Step 9:
  put 0, 0, 100, 100 (the x1, y1, x2, y2 coordinates for the bounding box)
    in G21:J21 (under the space for the plot)
  put the label "Bounding Box:" in A21
  define names "xmin", "ymin", "xmax", "ymax" for G21, H21, I21, J21 respectively

  Guiding Knowledge:
    the bounding box is needed to calculate the fitted line
      from the slope and intercept
    labels are good (see step 6)
    names are really useful sometimes (see step 6)

  Obstacles:
    this is all straightforward spreadsheet use
      except defining names, which is a more sophisticated feature
    (but all the name-defining is for the sake of the computation,
      not the interaction)

Step 10:
  place the formulas =xmin, =slope*xmin+intercept, =xmax, =slope*xmax+intercept
    in G22:J22
  put the label "Vertical Intercepts:" in A22
  define names "x_1", "y_2", "x_2", "y_2" for G22, H22, I22, J22

  Guiding Knowledge:
```

```
    these formulas calculate the vertical intercepts of the line
    labels are good (see step 6)
    the names are needed for the next set of formulas

  Obstacles:
    the names are internal and thus somewhat arbitrary
    (these formulas are part of the computation and thus are assumed given)

Step 11:
  place the formulas
    =IF(AND(ymin<=y_1,y_1<=ymax),x_1,IF(ymin>y_1,(ymin-intercept)/slope,
                                                 (ymax-intercept)/slope))
    =IF(AND(ymin<=y_1,y_1<=ymax),y_1,IF(ymin>y_1,ymin,ymax))
    =IF(AND(ymin<=y_2,y_2<=ymax),x_2,IF(ymin>y_2,(ymin-intercept)/slope,
                                                 (ymax-intercept)/slope))
    =IF(AND(ymin<=y_2,y_2<=ymax),y_2,IF(ymin>y_2,ymin,ymax))
    in G23:J23
  put the label "Fitted Line:" in A23

  Guiding Knowledge:
    these are the (hideous) formulas for clipping a line
      from a bounding box and vertical intercepts
    labels are good (see step 6)

  Obstacles:
    (these formulas are part of the computation and thus are assumed given)

Step 12:
  infer that none of the components provided by Visual Basic are appropriate

  Guiding Knowledge:
    the available components are
      label, edit box, group box, button, check box, option button,
      list box, drop-down box,
      combination list-edit box, combination drop-down edit box
      scrollbar, spinner
    only the button comes close
      buttons are square, have text labels, and each has its own macro

  Obstacles:
    aside from the usual difficulty of determining an absence of functionality
      this should be no problem

Step 13:
  conclude that graphics with linked behavior is the way to go

  Guiding Knowledge:
    none of the standard components are appropriate
    Excel provides facilities for linking graphics to macros

  Obstacles:
    the HU must know this to proceed
    this is one of the major appeals and selling points of Visual Basic

Step 14:
  decide to have a Visual Basic procedure draw the graphics

  Guiding Knowledge:
    Visual Basic functionality must be reestablished when a document is opened
    spreadsheet additions or changes may change Visual Basic references
    having all of the functionality originating in one place
      is conceptually easier
      allows easy recreation of the solution
      avoids possibly ambiguous connections with the spreadsheet and graphics

  Obstacles:
    recognizing the potential complications ahead of time is tricky
      perhaps less so for a veteran programmer, mistrustful of direct manipulation
    using Visual Basic to create and set up OnAction calls is easier
      because the newly drawn graphic is also the selection

Step 15:
  decide to record a macro to use as a starting point for the Visual Basic procedure

  Guiding Knowledge:
    drawing the graphics by direct manipulation is easy
    recording a macro is easy
    figuring out how to use the language with the macro as an example
      is drastically easier than figuring it out from scratch
```

```
  Obstacles:
    the HU must know about macro recording and editing macros
    this is pretty central to Visual Basic

Step 16:
  turn on macro recording by
    selecting Record New Macro... from the Record Macro submenu of the Tools menu
    clicking OK in the dialog box

  Guiding Knowledge:
    this is how to turn on macro recording

  Obstacles:
    ok -- basic Mac interface knowledge
    must know where to find command

Step 17:
  draw a solid rectangle for the background of the plot
    as a square above the bounding box calculation
    and to the right of the data
  leave it white

  Guiding Knowledge:
    the plot will look better and be easier to distinguish if it has a background
    a solid rectangle is needed
    white is an appropriate color

  Obstacles:
     the HU might choose the outline rectangle first
     the HU might choose drawing the dots first
       drawing the rectangle afterward will cover them
     in any case, the main issue is that the macro will be more complicated
     and the main remedy will be to re-record it with the correct actions only

Step 18:
  draw a solid oval for the dot (as a small circle)
  pull down the color palette and select blue
  pull down the color palette and select yellow

  Guiding Knowledge:
    the dots need to be plotted
    blue is a good color
    they'll all be plotted the same way except for coordinates
    the dots will be changed to yellow, so set them to yellow also

  Obstacles:
    should be no problem
    HU probably won't draw outlined circles
    see also step 17 obstacles
    EXCEPT that the HU really can't be expected to anticipate
      needing to know how to change dots to yellow at this early stage
      (the author just assumed it would be .SomeProperty = "yellow")
        (see obstacles to steps 23 and 24 for more on this)
      but it simplifies the walkthrough without loss of insight into the process

Step 19:
  draw a line (for the fitted line)
  pull down the color palette and select red

  Guiding Knowledge:
    the fitted line will need to be drawn
    red is a good color

  Obstacles:
    no problem, see also step 17 obstacles

Step 20:
  turn off macro recording
    by selecting Stop Recording on the Record Macro submenu of the Tools menu
    or by clicking the Stop (black square) button on the floating toolbar

  Guiding Knowledge:
    this is how to turn off macro recording
    turning off macro recording avoids unnecessary recorded clutter
    attaching actions to the graphic objects might be too complicated for now

  Obstacles:
    remembering to turn off macro recording is tricky
```

```
    actually turning it off should be easy
    failing to turn it off means extra stuff at the end of the macro

Step 21:
  look at the macro by (scrolling the tabs and) selecting the (new) Module 1 tab

  Guiding Knowledge:
    a new macro is stored in a Module (possibly newly created)
    clicking the tab of a sheet or module displays it in the window
    scrolling the line of tabs may be necessary to see all tabs

  Obstacles:
    should be ok for an Excel 5 user
    one who hasn't used macros could fail to notice new module
    alternative:  select Macro... from the Tools menu, select Macro1, click Edit

The macro is:


Sub Macro1()
    ActiveSheet.Rectangles.Add(449, 27, 402, 369).Select
    Selection.Interior.ColorIndex = 2
    ActiveSheet.Ovals.Add(797, 54, 16, 14).Select
    Selection.Interior.ColorIndex = 2
    Selection.Interior.ColorIndex = 5
    Selection.Interior.ColorIndex = 6
    ActiveSheet.Lines.Add(483, 346, 808, 78).Select
    Selection.Border.ColorIndex = 3
End Sub


Step 22:
  infer that the .Add lines create the rectangle, oval, and line

  Guiding Knowledge:
    a rectangle, circle, and oval were drawn while the macro was running
    'Add' suggests creation (although not as well as 'Create' or 'Draw' would)
    the lines contain 'Rectangles', 'Ovals', or 'Lines' respectively
    the other lines do not suggest creating anything, nor do they suggest shapes

  Obstacles:
    this seems reasonable
      although (as partially noted), 'Add' doesn't suggest drawing very well
           and the plural (Rectangles, etc) is potentially confusing
             since there are no rectangles, etc before these commands execute
               and there's only one of each afterwards
      we're also ignoring the .Select for now (see step 24)

Step 23:
  infer that the Selection...ColorIndex lines set the color of the selection
    from some sort of table

  Guiding Knowledge:
    'Selection' and 'Color' are part of the statement
    there's no other object named in the statement
    the value is a number and the name includes 'Index'
      which suggests a table
      (rather than setting the color directly)

  Obstacles:
    this notation also seems a bit removed from the user's model
    what's the point of indexing the color
      (why can't the macro just set the color?)
    the HU might also wonder why the rectangle's color is set
      and why the oval's color is set twice
    we're assuming that the HU ignores the Interior and Border parts

Step 24:
  infer that the .Select in the .Add lines selects the newly created object

  Guiding Knowledge:
    each object was selected immediately after it was drawn
    the .ColorIndex line immediately following each object changes it
      but refers to the selection

  Obstacles:
    the .ColorIndex lines make this step a lot less obvious
      than analogous .Color lines would
    the .Select specifies an action (selection)
```

```
      on a newly created object (rectangle, oval, line)
      which could be hard to infer smoothly from the syntax

Step 25:
  conclude that the macro recorded the direct-manipulation actions as:
    the .Add statements created their respective objects
    the .Select parts of those statements selected the objects
    the Selection...ColorIndex statements set the colors

  Guiding Knowledge:
    the order of the statements and their apparent function
      match the order and behavior of the direct-manipulation actions performed

  Obstacles:
    this should be ok given previous steps

Step 26:
  infer that, in the statement creating the oval, the four numbers are
    x, y,  width, height

  Guiding Knowledge:
    the x and y distances match the distance of the oval from left and top edges
    width and height are about the same and are small
    x (likewise width) almost always comes before y (likewise height)

  Obstacles:
    the HU must successfully guess that the y coordinates increase downwards
    the HU might assume x1,y1,x2,y2 in spite of the above knowledge

Step 27:
  infer that the statement creating the square also uses x,y,width,height

  Guiding Knowledge:
    the oval used it, so probably the square does, too
    the square's second two numbers are about the same size

  Obstacles:
    assuming step 26 goes ok, this one should, too

Step 28:
  infer that the statement creating the line uses x1,y1,x2,y2

  Guiding Knowledge:
    the second two numbers are a factor of ten apart
      and are reasonable coordinates for the second endpoint of the line
      and are NOT a reasonable width and height for its bounding box
    thinking of a line in terms of width and height is unusual

  Obstacles:
    in this case, there's a choice of incongruities
      inconsistent notation between solid shapes and lines
      or counterintuitive use of "width" and "height" for a line

Step 29:
  decide that the square drawn by the procedure should be 100 by 100

  Guiding Knowledge:
    the data fall nicely in the range 0..100 on both axes

  Obstacles:
    this is an important graphics choice

Step 30:
  decide to place the square at (450,30)

  Guiding Knowledge:
    the direct-manipulation-drawn square was at (449,27)
    round numbers are easier

  Obstacles:
    this should be no problem and is not critical anyway
    it does touch on the origin issue
    (using Range().Left seems to complicated for this walkthrough)
      although it may be more robust)

Step 31:
  edit the .Rectangles.Add line to read
    ActiveSheet.Rectangles.Add(450, 30, 100, 100).Select
```

```
  Guiding Knowledge:
    the edited statement will draw a side-100 square at (450,30)

  Obstacles:
    should be ok given steps 29 and 30

Step 32:
  add 'End Sub' after the .ColorIndex line after the .Rectangles.Add line
    and change the name of the macro from Macro1 to DrawBorderRectangle

  Guiding Knowledge:
    we don't know how we're going to draw the ovals yet
      and the line must be redrawn repeatedly and thus must be in a separate macro

  Obstacles:
    this seems reasonable
      but dividing functionality into procedures
      can be hard to do well


The DrawBorderRectangle macro/procedure is thus:

Sub DrawBorderRectangle()
    ActiveSheet.Rectangles.Add(450, 30, 100, 100).Select
    Selection.Interior.ColorIndex = 2
End Sub


Step 33:
  recognize that some sort of iteration (For ... To ... Next)
    will be needed to plot the points

  Guiding Knowledge:
    imperative languages have iteration constructs
      most have a For-like iteration construct
    the points could be plotted by individual statements
      but that would tie the code to the number of points (bad design)

  Obstacles:
    this is a crucial step, and one of the most programming-esque
      partly because there is no help whatsoever available from macro recording

Step 34:
  recognize that the iteration construct must have access to the number of points

  Guiding Knowledge:
    a fixed iteration needs an upper bound
    the iteration must happen as many times as there are points

  Obstacles:
    this is also important, but probably not problematic

Step 35:
  decide to calculate the number of points with the COUNT spreadsheet function

  Guiding Knowledge:
    the COUNT function returns the number of numeric cells in a range
    Visual Basic will need to get this value somehow
      but the alternative is to get ALL of the values somehow, then count them

  Obstacles:
    knowing to use the spreadsheet whenever possible is an important simplifier
    the definition of "numeric" above is somewhat complicated
      but we ignore that here

Step 36:
  put the formula =COUNT(A5:A14) into cell D1
  put the label "Number of Points:" into cell A1

  Guiding Knowledge:
    COUNT(A5:A14) counts the number of points
      we're not worrying about points with only one coordinate present
      we're not worrying about making formula work when points are added
    labels are good

  Obstacles:
    this should be no problem
    anticipating the potential ways it could break
      is a whole 'nother area of spreadsheet use and analysis
```

```
Step 37:
  decide to use Cells(1,4).Value in Visual Basic for the COUNT result in D1

  Guiding Knowledge:
    Cells(1,4) is the way to refer to D1 in Visual Basic
      row 1 is the 1, column D is the 4
      the "coordinates" are (row,column), not (x,y)
    the .Value property is necessary to obtain the value of the cell

  Obstacles:
    there's a lot here that simply has to be learned
      and the only way to do so is through the (very good on-line) documentation
    (no macro recordings will show this)
    having the cell coordinates be (row,column) is inconsistent with both
      the (x,y) graphics coordinates
      the A1-style cell notation

Step 38:
  write Sub PlotPoints()
  write End Sub on the next line

  Guiding Knowledge:
    this is the way to begin a macro/procedure
    PlotPoints is an appropriate name
    since plotting the points is a self-contained activity
      putting it in a macro/procedure of its own makes sense for good organization

  Obstacles:
    the HU might not yet decide at this point to create a procedure
    or might just write this code and stick it in a bigger procedure

Step 39:
  write For PointNumber = 1 To Cells(1,4).Value
    (between the Sub and End Sub lines)
    and write Next PointNumber on the next line
    (indented appropriately -- possibly by the editor?)

  Guiding Knowledge:
    this is how the For ... To ... Next iteration construct is laid out
    PointNumber is a good name for the point to be plotted
    running the iteration from 1 to the number of points seems simplest
    Cells(1,4).Value refers to the number of points

  Obstacles:
    the HU must know that the For ... To ... Next construct exists
    the HU must know that the For ... To ... Next construct is the best to use here
    the HU must know how to use the For ... To ... Next iteration construct
      (see also step 33)

Step 40:
  copy the next three lines (.Ovals.Add and both Selection...ColorIndex lines)
    from the remnants of Macro1
    and put them between the For line and the Next line

  Guiding Knowledge:
    these lines draw an oval on the screen and color it blue
    the iteration needs to draw a blue oval for each point coordinates
    editing a macro is faster and easier than writing from scratch

  Obstacles:
    the HU might just look at the lines and think about changing them
      (but not copy them)

Step 41:
  decide that the first Selection...ColorIndex line is unnecessary and delete it

  Guiding Knowledge:
    each Selection...ColorIndex line changes the color of the oval
    the second line destroys the changes made by the first line
      so the first line has no effect on the final outcome and can be deleted
    the second and third Selection...ColorIndex lines
      correspond to setting the color to blue and yellow respectively

  Obstacles:
    this is reasoning about imperative programs
      which can sometimes be problematic
      and in any case, must be learned
    expecting the HU to know that the last two lines correspond to her/his actions
```

```
      is a little farfetched
      but expanding this out into several steps
        does not add any new insights to the walkthrough


The PlotPoints macro now looks like:

Sub PlotPoints()
    For PointNumber = 1 To Cells(1,4).Value
        ActiveSheet.Ovals.Add(797, 54, 16, 14).Select
        Selection.Interior.ColorIndex = 5
        Selection.Interior.ColorIndex = 6
    Next PointNumber
End Sub


Step 42:
  replace the two Selection.Interior.ColorIndex lines with
    If Cells(PointNumber + 4, 3).Value Then
      Selection.Interior.ColorIndex = 5
    Else
      Selection.Interior.ColorIndex = 6
    End If

  Guiding Knowledge:
    the macro needs to plot different colors based on the status of the point
    the If...Then...Else picks a color depending on the status
    Cells(PointNumber + 4, 3).Value is TRUE if the point is an inlier
      and FALSE if the point is an outlier
    5 and 6 are the indices of blue and yellow

  Obstacles:
    this is a bit much for one step
      but it's quite straightforward
    the only new bits are using an If...Then...Else
      which is somewhat analogous to IF formula in the spreadsheet
    and using a Boolean value as a branch condition
      which was also done in spreadsheet formulas (see step 5)


The PlotPoints macro now looks like:

Sub PlotPoints()
    For PointNumber = 1 To Cells(1,4).Value
        ActiveSheet.Ovals.Add(797, 54, 16, 14).Select
        If Cells(PointNumber + 4, 3).Value Then
          Selection.Interior.ColorIndex = 5
        Else
          Selection.Interior.ColorIndex = 6
        End If
    Next PointNumber
End Sub


Step 43:
  decide that the points should have width and height 10
    and change the third and fourth numbers in the .Ovals.Add line to 10

  Guiding Knowledge:
    the oval drawn by the macro is 16x14
      the square background is smaller, so the points should be, too
    10 is a nice round number

  Obstacles:
    none, really
    15x15 would probably still be ok

Step 44:
  recognize that Cells(row,col).Value
    can be used to refer to the coordinates needed to plot the point
    (for suitable values of row and col)

  Guiding Knowledge:
    Cells(1,4).Value was used to get the number of points from D$
    the coordinates are in other cells (A5:B14, to be precise)

  Obstacles:
    this shouldn't be a problem; the conceptual work is already done
```

```
Step 45:
  recognize that the row of each x- and y-coordinate
    is the number of the point plus 4

  Guiding Knowledge:
    the points start in A5:B5 and are listed there and in subsequent rows
    the point numbers start with 1
    5 - 1 = 4
    the last point (number 10) is in A14:B14
    14 - 10 = 4

  Obstacles:
    the HU must be able to see this to make the iteration happen

Step 46:
  conclude that the correct references are
    Cells(PointNumber + 4, 1).Value for the x-coordinate
    Cells(PointNumber + 4, 2).Value for the y-coordinate

  Guiding Knowledge:
    the x-coordinate is in column A
    column A is numbered 1
    the y-coordinate is in column B
    column B is numbered 2

  Obstacles:
    this should be trivial compared with step 45

Step 47:
  replace the first two numbers of in the .Ovals.Add line
    with Cells(PointNumber + 4, 1).Value
    and Cells(PointNumber + 4, 2).Value

  Guiding Knowledge:
    these are the correct references for the current point number

  Obstacles:
    this should be straightforward
    previous steps are working towards it

Step 48:
  recognize that to plot the points with the y-axis increasing upwards
    the y-coordinate must be subtracted from 100
  and replace Cells(PointNumber + 4, 2).Value
    with 100 - Cells(PointNumber + 4, 2).Value
  in the .Ovals.Add line

  Guiding Knowledge:
    the y-coordinates increase downwards (see step 26)
    subracting the y-coordinate from 100
      will reverse the axis
      and put the origin at the lower left corner

  Obstacles:
    the HU must know that the y-axis points down (see step 26)
      (possibly by running the macro and observing the results)
    the HU must know how to change that
      not just negating, but also adding 100
    (see also step 49)

Step 49:
  recognize that the points will appear in the upper left corner of the screen
    not in the square drawn by DrawBorderRectangle
  and that the coordinates of the upper left corner of the square
    must be added to the coordinates in the .Ovals.Add line
  and thus
    replace Cells(PointNumber + 4, 1).Value
      with Cells(PointNumber + 4, 1).Value + 450
    and replace 100 - Cells(PointNumber + 4, 2).Value
      with 100 - Cells(PointNumber + 4, 2).Value + 30
      or actually, 130 - Cells(PointNumber + 4, 2).Value

  Guiding Knowledge:
    the rectangle's upper left corner is (450,30)
    the correct x-coordinate for each oval is the old coordinate + 450
    the correct y-coordinate for each oval is the old coordinate + 30
    100 + 30 = 130

  Obstacles:
```

```
    the HU must recognize the need for an offset
      (possibly by running the macro and observing the results)
    the HU must know to add the upper left corner
      (not the lower left corner, that's taken care of by the 100 in step 48)
    the HU must recognize that the 100 - y expression
      is in terms of the upper left corner (step 48)
    the HU must know to add x to x and y to y


The PlotPoints macro now looks like:

Sub PlotPoints()
    For PointNumber = 1 To Cells(1,4).Value
        ActiveSheet.Ovals.Add(Cells(PointNumber + 4, 1).Value + 450, _
                              130 - Cells(PointNumber + 4, 2).Value, _
                              10,                                    _
                              10).Select
        If Cells(PointNumber + 4, 3).Value Then
          Selection.Interior.ColorIndex = 5
        Else
          Selection.Interior.ColorIndex = 6
        End If
    Next PointNumber
End Sub

(the _ is the line-continuation character (used here for readability),
 but the HU doesn't need to know about it or use it)


Step 50:
  decide to write the macro that draws the line next
    and write Sub DrawFittedLine() before
    and End Sub after
      the remaining scrap from Macro1

  Guiding Knowledge:
    the remaining two lines of Macro1
      draw a line on the screen and set its color to red

  Obstacles:
    none, really, but see steps 38 and 40


The DrawFittedLine macro now looks like:

Sub DrawFittedLine()
    ActiveSheet.Lines.Add(483, 346, 808, 78).Select
    Selection.Border.ColorIndex = 3
End Sub


Step 51:
  recognize that the endpoints of the fitted line
    are referred to by Visual Basic as
      Cells(23, 7).Value    (= x1)
      Cells(23, 8).Value    (= y1)
      Cells(23, 9).Value    (= x2)
      Cells(23, 10).Value   (= y2)

  Guiding Knowledge:
    the endpoints are in G23, H23, I23, J23
      which are referred to by Cells(23,7), Cells(23,8), Cells(23,9), Cells(23,10)

  Obstacles:
    this should be straightforward given steps 37 and 46
      (even though the indices are larger)

Step 52:
  recognize that the line needs to be offset in the same way as the points
    minus half the width of the points
  and replace the .Lines.Add line with
    ActiveSheet.Lines.Add(Cells(23, 7).Value + 455, _
                          135 - Cells(23, 8).Value, _
                          Cells(23, 9).Value + 455, _
                          135 - Cells(23, 10).Value).Select

  Guiding Knowledge:
    the .Lines.Add line takes the values x1, y1, x2, y2 in that order
    the cells G23:J23 hold those values in that order
```

```
    450 is the y offset for the left edge of the points, and they are 10 wide,
      so 455 is the correct x offset for the line
    130 is the y offset for the top edge of the points, and they are 10 wide,
      so 135 is the correct y offset for the line
    the '+ 455' needs to be put after both x-coordinates
    the '135 -' needs to be put before both y-coordinates

  Obstacles:
    the HU could easily miss the need to compensate for the points
      being anchored by top and left instead of center
      and knowing which way to offset and how much
        can be tricky
      but this relates more to the graphics than the programming
        and so is noted, but not pursued deeply, here
    there are a few chances for typographical errors
      like leaving off the .Select
      or forgetting to offset x2 and y2
        (since with the points, the widths and heights were unchanged)
    (the _ for line continuation are here for readability)
      (we aren't concerned about the HU learning or not learning them)


The DrawFittedLine macro now looks like:

Sub DrawFittedLine()
    ActiveSheet.Lines.Add(Cells(23, 7).Value + 450, _
                          130 - Cells(23, 8).Value, _
                          Cells(23, 9).Value + 450, _
                          130 - Cells(23, 10).Value).Select
    Selection.Border.ColorIndex = 3
End Sub


Step 53:
  write a CreatePlot macro as follows:

Sub CreatePlot()
    DrawBorderRectangle
    PlotPoints
    DrawFittedLine
End Sub

  Guiding Knowledge:
    drawing the rectangle, plotting the points, and drawing the line
      must all happen in that order
    writing a macro makes one macro that will do all of those things
    parameterless macros are called by other macros without using ()

  Obstacles:
    the HU might not know not to use ()
      but the syntax editor would catch it
      (although the message "Expected: expression" is minimally helpful)
    it's reasonable for the HU to make a procedure here

Step 54:
  infer that somehow a macro must be assigned
    either to each dot
    or to the plot as a whole
  and that the macro should (when a dot is clicked)
    change the color of the dot
    and change the status in the corresponding cell of the status column

  Guiding Knowledge:
    when a dot is clicked
      it should change both color and status
    a macro will do this
      but it must be assigned somehow to the dot or to the plot
    macros can be assigned to objects in Visual Basic

  Obstacles:
    this seems ok; no obvious pitfalls

Step 55:
  infer that, before looking into assigning a macro,
    a macro must be written

  Guiding Knowledge:
    Tools...Assign Macro... allows selection of an existing macro
    the macro to toggle the color and status hasn't been written
```

```
  Obstacles:
    this is reasonable
    attempting to assign the macro before it exists will lead to this step anyway

Step 56:
  deciding that the macro will do two things
    figure out which cell corresponds to the oval that is clicked
    changing the status of the cell and the color of the oval

  Guiding Knowledge:
    the macro must have some means of indicating which oval was clicked
      (else we are sunk before we begin)
    there must be some way of determining the corresponding cell
      (else ditto)
    once the cell and the oval are both known,
      then the purpose of the macro can be carried out
      (changing the cell status and dot color)

  Obstacles:
    at this point (and in fact also at the last step)
      it's quite challenging to know how to proceed
      because several important pieces of information are needed
        * that macros are parameterless
        * that graphic objects are indexed by integers in the order of creation
        * that these indices are obtained by a pair of special function calls
            (not by simply being passed as parameters to the assigned macro)
      or, more bluntly,
        + you don't know how to get the index
        + you don't know what it is or how to use it
      and, additionally, you don't really know where to start to find this out
    finally, all of these things need to be known or learned together to proceed
    interestingly, the online help mentions the ability
      to add an argument to a macro (via the "Assign Macro To Object..." dialog box)
    but there is only an "Assign Macro..." dialog box
      that doesn't support adding an argument

Step 57:
  decide to use Application.Evaluate(Application.Caller).Index
    to find out the index (in order of creation) of the clicked oval

  Guiding Knowledge:
    assigned macros are parameterless
      (the online help to the contrary notwithstanding)
    the Application.Caller property
      stores the non-Visual Basic caller of Visual Basic
      which, for graphic objects, is a string such as "Oval 2"
    the Application.Evaluate method
      converts Microsoft Excel names into values or objects
      for graphic objects, Evaluate takes a name and returns a graphic object
    the .Index property of graphic objects
      contains the index of the graphic object in the collection of similar objects
      so SomeOval.Index contains the index of SomeOval
        in the collection of all Ovals
        (they are indexed by order of creation)

  Obstacles:
    see obstacles for step 56
    this is not one, but three pieces of knowledge
      ranging from mildly obscure (.Index) to quite obscure (.Caller, .Evaluate)
    they expose the user to unnecessary system detail (MS Excel names)
    they don't even come close to attempting a coherent language design
    it is far from obvious how anything other than
        scouring the documentation
        being told how by another user
        already knowing how to do this
      would lead the HU to this solution
    it's tricky enough learning that assigned macros take no parameters
    the HU has nothing to tell her/him that the .Index is in order of creation

Step 58:
  begin writing the TogglePoint macro:

Sub TogglePoint()
    IndexOfClickedDot = Application.Evaluate(Application.Caller).Index
End Sub

  Guiding Knowledge:
    Sub TogglePoint() ... End Sub is the frame for a macro
```

```
    IndexOfClickedDot = Application.Evaluate(Application.Caller).Index
      sets the variable IndexOfClickedDot to the index of the clicked dot
        (which is what Application.Evaluate(Application.Caller).Index returns)

  Obstacles:
    this is a simple application of the result of step 57

Step 59:
  infer that IndexOfClickedDot is completely analogous
    to PointNumber in PlotPoints
  and in particular, that they both index the dots by order of creation
    and the rows in increasing row number
  so that IndexOfClickedDot can index the rows in the same way that PointNumber did

  Guiding Knowledge:
    PointNumber indexed the points by row
    PlotPoints created the dots in the same order as the rows
    IndexOfClickedDot indexes the dots by order of creation

  Obstacles:
    this is not obvious
      but forturnately, it works
    the sensible HU would check this inference out carefully
      (perhaps with the completed program)

Step 60:
  infer that Cells(IndexOfClickedDot + 4, 3).Value
    is the status of the clicked dot

  Guiding Knowledge:
    Cells(PointNumber + 4, 3).Value was the status of the point in PlotPoints

  Obstacles:
    this seems reasonable (although not trivial) given step 59

Step 61:
  write (in analogy to part of PlotPoints) after the Application.Evaluate line
    If Cells(IndexOfClickedDot + 4, 3).Value Then
      Selection.Interior.ColorIndex = 5
    Else
      Selection.Interior.ColorIndex = 6
    End If

  Guiding Knowledge:
    PlotPoints set the color of the dot based on the status of the point
      using exactly these lines (with PointNumber rather than IndexOfClickedDot)
    IndexOfClickedDot is the analog of PointNumber here
    5 and 6 are the color indices of blue and yellow respectively
      5 corresponds to True (inlier)
      6 corresponds to False (outlier)

  Obstacles:
    this seems pretty reasonable
      the HU is performing an identical task with virtually identical code
    however, there is a risk that the HU might forget step 62 altogether

Step 62:
  add Cells(IndexOfClickedDot + 4, 3).Value = False
    just before the Selection.Interior.ColorIndex = 5 line
  add Cells(IndexOfClickedDot + 4, 3).Value = True
    just before the Selection.Interior.ColorIndex = 6 line
  interchange the 5 and 6 in the Selection.Interior.ColorIndex lines

  Guiding Knowledge:
    not only does the dot color need to be toggled
      but so does the status as stored in column C
    the status is stored as a Boolean (TRUE or FALSE)
    in the first pair of lines, the status is True (inlier)
      and thus must be set to False (outlier)
      and the color must be set to yellow (the outlier color)
    in the second pair of lines the status is False (outlier)
      and thus must be set to True (inlier)
      and the color must be set to blue (the inlier color)

  Obstacles:
    not inverting correctly is a risk here
    the HU must know about setting Boolean values
    using Boolean 'not' on the Cell value might make more sense
      (but this is simpler from a constructive standpoint
```

```
    actually, adding Cells(IndexOfClickedDot + 4, 3).Value = _
                                        Not(Cells(IndexOfClickedDot + 4, 3).Value)
      before the whole If...Then...Else bit is the smallest change
        (but we're not going back and changing it now -- it's not important enough)
    (see also step 61 obstacles)


The TogglePoint macro now looks like:

Sub TogglePoint()
    IndexOfClickedDot = Application.Evaluate(Application.Caller).Index
    If Cells(IndexOfClickedDot + 4, 3).Value Then
      Cells(IndexOfClickedDot + 4, 3).Value = False
      Selection.Interior.ColorIndex = 5
    Else
      Cells(IndexOfClickedDot + 4, 3).Value = True
      Selection.Interior.ColorIndex = 6
    End If

End Sub


Step 63:
  decide to use macro recording to find out
    how to assign the TogglePoint macro to the plot or dots
  and turn on macro recording as described in step 16

  Guiding Knowledge:
    macro recording is an efficient way to capture Visual Basic statements
      that duplicate what can be done using the interface
    there is an Assign Macro ... menu option on the Tools menu
    (see also step 16)

  Obstacles:
    this seems pretty straightforward
    knowing about the Assign Macro menu makes a significant difference here

Step 64:
  select the oval drawn in step 18

  Guiding Knowledge:
    an object must be selected before a macro can be assigned to it

  Obstacles:
    the HU might forget to do this
      if so, the Assign Macro menu item will be grayed out
    we're avoiding the issue of needing to use Cmd-click
      to select an object that already has a macro assigned to it

Step 65:
  from the Tools menu, select Assign Macro...
  in the dialog box, double-click on TogglePoint

  Guiding Knowledge:
    the Assign Macro... menu option
      is on the Tools menu
      and assigns a macro to the selected object

  Obstacles:
    this shouldn't be a problem

Step 66:
  turn off macro recording as described in step 20

  Guiding Knowledge:
    (see step 20)

  Obstacles:
    (see step 20)

Step 67:
  look at the macro as in step 21

  Guiding Knowledge:
    (see step 21)

  Obstacles:
    (see step 21)
```

```
The macro is:

Sub Macro2()
    ActiveSheet.DrawingObjects("Oval 2").Select
    Selection.OnAction = "TogglePoint"
End Sub


Step 68:
  infer that the ActiveSheet line selects the oval

  Guiding Knowledge:
    the ActiveSheet line ends in .Select
    in the first macro, the lines ending in .Select selected object (see step 24)

  Obstacles:
    the HU might think further understanding of this line is required
      but it isn't (see step 70)
    the use of "Oval 2" in the generated code
      seems like gratuitous exposure of Microsoft Excel names

Step 69:
  infer that the Selection.OnAction line
    assigns the TogglePoint macro to the oval

  Guiding Knowledge:
    the macro was assigned by direct manipulation
    assigning the macro was the second of the two actions
      and this line is the second of the two lines
    the name "TogglePoint" is on the right side of the =
    the object was selected, and the left side begins with Selection.
    the other line very clearly doesn't assign the macro

  Obstacles:
    this seems pretty clear, even if .OnAction is completely new

Step 70:
  infer that the ActiveSheet line does not need to be used
    and that the
      Selection.OnAction = "TogglePoint"
    line can be used as is

  Guiding Knowledge:
    the PlotPoints macro selects each point as it is created
      and so the Selection.OnAction line will assign the macro to that point

  Obstacles:
    this involves a bit of insight into the Selection idea
    it also involves some of the knowledge from step 71

Step 71:
  infer that TogglePoint
    doesn't take any arguments
    can't get any from being assigned to the oval
    doesn't need any

  Guiding Knowledge:
    TogglePoint is written with no arguments
    the macro assignment shown in Macro2 doesn't indicate arguments
      (and in fact, none are possible)
    TogglePoint uses Application.Evaluate(Application.Caller).Index
      to find out which object was clicked on

  Obstacles:
    this step is a little subtle
    but since none of the macros written in the walkthrough have arguments
      the HU might well not take this step at all
      (although she/he might wonder why macro definitions ended in '()'!)

Step 72:
  insert the Selection.OnAction = "TogglePoint" line
      after the End If
      and before the Next PointNumber
    lines in PlotPoints

  Guiding Knowledge:
    'Selection' was just used in previous lines (and thus should be defined)
    the TogglePoint macro needs to be assigned to every dot
```

```
    the TogglePoint macro will work for every dot
    the TogglePoint macro must not be assigned in the If statement
      because it would be assigned only to points of one status

  Obstacles:
    it doesn't matter where this line goes so long as it's
      in the For statement
      and outside the If statement
    the tricky part is knowing
       that the TogglePoint macro will work for every dot


The PlotPoints macro now looks like:

Sub PlotPoints()
    For PointNumber = 1 To Cells(1,4).Value
        ActiveSheet.Ovals.Add(Cells(PointNumber + 4, 1).Value + 450, _
                              130 - Cells(PointNumber + 4, 2).Value, _
                              10,                                    _
                              10).Select
        If Cells(PointNumber + 4, 3).Value Then
          Selection.Interior.ColorIndex = 5
        Else
          Selection.Interior.ColorIndex = 6
        End If
        Selection.OnAction = "TogglePoint"
    Next PointNumber
End Sub


Step 73:
  learn that Application.OnCalculate is the way to have the line redrawn
    each time a point is clicked

  Guiding Knowledge:
    Application.OnCalculate can be set to a macro (which could be DrawFittedLine)
      and then that macro will run each time the spreadsheet recalculates
    the spreadsheet will recalculate each time a point is clicked
      because the status cell will change, and it is referenced by a formula

  Obstacles:
    the HU must find out about .OnCalculate
      it is analogous to .OnAction
      but there is no other help in finding it
        or even for knowing it exists
    because this is an application, the functionality provided is harder to infer

Step 74:
  add Application.OnCalculate = "DrawFittedLine"
    as the last line of the CreatePlot() macro

  Guiding Knowledge:
    the CreatePlot() macro is the "main" program
      and thus is the correct place to assign DrawFittedLine
    as the recalculation macro

  Obstacles:
    this seems pretty reasonable
    the line could also go in PlotPoints (outside of the For loop)
      or in DrawBorderRectangle
      or even in DrawFittedLine, which makes little sense


The CreatePlot macro now looks like:

Sub CreatePlot()
    DrawBorderRectangle
    PlotPoints
    DrawFittedLine
    Application.OnCalculate = "DrawFittedLine"
End Sub


Step 75:
  recognize that when the line is redrawn, the old one must be deleted
  and add an ActiveSheet.Lines.Delete line
    before the ActiveSheet.Lines.Add line
    in DrawFittedLine
```

```
Guiding Knowledge:
  if the line isn't deleted, the display will show a confusing clutter of lines
  modifying the line in XLVB means changing its width and height
    and its left and top extremities
    which is trickier than changing its endpoints
  ActiveSheet.Lines.Delete deletes all lines
    and, importantly, doesn't complain if there aren't any
    which means that the initial special case doesn't need to be handled
  obviously, deleting before adding avoids deleting the new line

Obstacles:
  recognizing the need for deletion is relatively easy (run the program)
  knowing that the resizing the line is messy is a little hard
    and the resizing-by-ltwh model is cumbersome for lines
  the existence of ActiveSheet.Lines.Delete
    is guessable from ActiveSheet.Lines.Add
  knowing that it deletes all lines and not just some is a little trickier
  knowing that it doesn't complain if there are no lines is more tricky
    (the HU would probably have to test it out)
```

D.2 Final Line-Fitting Example in Excel Visual Basic

```
'
' DrawBorderRectangle Macro
'
Sub DrawBorderRectangle()
'   ActiveSheet.Rectangles.Add(210, 13, 208, 195).Select
    ActiveSheet.Rectangles.Add(Range("$G$17").Left, _
Range("$G$17").Top - 200, _                                  200, _
200).Select
    With Selection
        With .Interior
            .Color = RGB(255, 255, 255)
        End With
        .Placement = xlMove
        .PrintObject = True
    End With
End Sub
'
' TransformX Function
'
Function TransformX(x)
    TransformX = 2 * x + Range("$G$17").Left
End Function
'
' TransformY Function
'
Function TransformY(y)
    TransformY = Range("$G$17").Top - 2 * y
End Function
'
' TogglePoint Procedure
'
Sub TogglePoint()
    i = Application.Evaluate(Application.Caller).Index
    If Cells(i + 4, 3).Value Then
        ActiveSheet.Ovals(i).Interior.Color = RGB(255, 255, 0)
        Cells(i + 4, 3).Value = False
    Else
        ActiveSheet.Ovals(i).Interior.Color = RGB(0, 0, 255)
        Cells(i + 4, 3).Value = True
    End If
End Sub
'
' PlotPoints Procedure
'
Sub PlotPoints()
    r = 5
    For i = 1 To Cells(1, 4).Value
        ActiveSheet.Ovals.Add(TransformX(Cells(i + 4, 1).Value) - r, _
TransformY(Cells(i + 4, 2).Value) - r, _                                2 * r, _
2 * r).Select
        With Selection
            .OnAction = "TogglePoint"
            With .Interior
                If Cells(i + 4, 3).Value Then
```

```
                .Color = RGB(0, 0, 255)
            Else
                .Color = RGB(255, 255, 0)
            End If
        End With
        .Placement = xlMove
        .PrintObject = True
    End With
Next i
End Sub
'
' DrawLine Procedure
'
Sub DrawFittedLine()
    x1 = TransformX(Cells(23, 7).Value)
    y1 = TransformY(Cells(23, 8).Value)
    x2 = TransformX(Cells(23, 9).Value)
    y2 = TransformY(Cells(23, 10).Value)
    ActiveSheet.Lines.Delete   'so that this procedure can be used for redrawing
    ActiveSheet.Lines.Add(x1, y1, x2, y2).Select
    With Selection
        With .Border
            .Color = RGB(255, 0, 0)
        End With
        .Placement = xlMove
        .PrintObject = True
    End With
End Sub
'
' CreatePlot Procedure
'
Sub CreatePlot()
    DrawBorderRectangle
    PlotPoints
    DrawFittedLine
    DisplayGraphicsData   'will trigger DrawLine again; it's ok and unavoidable
    Application.OnCalculate = "DrawFittedLine"
End Sub
'
' DisplayGraphicsData Procedure
'
Sub DisplayGraphicsData()
    Cells(30, 1) = "Object"
    Cells(30, 2) = "Left"
    Cells(30, 3) = "Top"
    Cells(30, 4) = "Width"
    Cells(30, 5) = "Height"

    Cells(31, 1) = "Rectangle"
    Cells(31, 2) = ActiveSheet.Rectangles(1).Left
    Cells(31, 3) = ActiveSheet.Rectangles(1).Top
    Cells(31, 4) = ActiveSheet.Rectangles(1).Width
    Cells(31, 5) = ActiveSheet.Rectangles(1).Height

    Cells(32, 1) = "Line"
    Cells(32, 2) = ActiveSheet.Lines(1).Left
    Cells(32, 3) = ActiveSheet.Lines(1).Top
    Cells(32, 4) = ActiveSheet.Lines(1).Width
    Cells(32, 5) = ActiveSheet.Lines(1).Height

    For i = 1 To ActiveSheet.Ovals.Count
        Cells(32 + i, 1) = "Oval " & i
        Cells(32 + i, 2) = ActiveSheet.Ovals(i).Left
        Cells(32 + i, 3) = ActiveSheet.Ovals(i).Top
        Cells(32 + i, 4) = ActiveSheet.Ovals(i).Width
        Cells(32 + i, 5) = ActiveSheet.Ovals(i).Height
    Next i
End Sub
'
' DeleteGraphics Procedure
'
Sub DeleteGraphics()
    ActiveSheet.Lines.Delete
    ActiveSheet.Ovals.Delete
    ActiveSheet.Rectangles.Delete
End Sub
```

Appendix E

Temperature Conversion Walkthrough in Esquisse

A screen image of the final temperature conversion example in Esquisse appears in figure 4.6.

Temperature Conversion Walkthrough in Esquisse

(1) Click on filled rectangle radio button.

- The palette offers lines and filled and outline squares, circles, rectangles, and ellipses. The HU wants to draw a thermometer; a narrow rectangle is a reasonable choice.
- (A line is also a reasonable choice. Lines are harder to draw vertically, and extracting the height from a line would be a bit harder.)

(2) Click on red color button.

- The palette offers a selection of colors. No problems here.

(3) Drag out a narrow rectangle in the left side of the work area.

- No problem here, either; this requires only basic drawing-package knowledge.
- (The lack of automatic refresh in Haggis can sometimes present a small potential confusion about the size of the rectangle.)

(4) Click on the Expressions button.

- Here, the HU must understand the difference between the Text and Expression buttons ("modes"). (In the former, character strings can be typed; in the latter, the cell is valid only if the contents are an evaluable to a Gofer value). "Expressions" is not the best name for that mode (but I don't know what's better).
- Another potential stumbling block is that the HU might not create all of the cells before starting to edit some of them. The risk is that she/he might refer in a formula to a not-yet-created cell, and thus the formula will not be valid. (In the current version of Esquisse, cells can not be renamed from the generated defaults, and so this obvious drawback lessens the risk.)

(5) Click under the red rectangle (a text cursor appears) and type 212.

- This seems ok given experience with graphics packages.

(6) Click on filled rectangle radio button again.

- See step 1.

(7) Click on the blue color button.

- No problem here. See also step 2.

(8) Drag out a narrow rectangle next to the first one.

- No problem here. See also step 3.

(9) Click on the Expressions button again.

- The HU might notice that the mode has changed to Interact. It's not necessary for her/him to understand why, but it's good for her/him to know that it's intentional. Also, the automatic switching to the Interact cursor (pointing hand) could be confusing.

(10) Click under the blue rectangle (a text cursor appears) and type 100.

- This seems ok given experience with graphics packages.

(11) Decide that the next step is to edit each rectangle cell to (a) make dragging change the height of the rectangle, (b) link the rectangles to their respective numeric cells, and (c) link the numeric cells with formulae.

- The HU must know that the rectangles are the "interfaces" to cells, must understand that cells hold values, display and event functions, and formulae. To know to change the value, the HU must know that the default value for the rectangle is ((x,y),(w,h), not simply the height. To know to change the event function, the HU must know that it must be consistent with the value. The HU must also know that formulae are used to specify constraints between cells.

(12) Click on the Select mode button.

- Here, the HU must understand the mode buttons, especially the distinction between Select (used for select objects to edit, raise, lower, etc., that is, to manipulate outside of the cell framework) and Interact (used for interacting with the cell framework, with the interaction specified by the cell event functions and cell formulae).
- This is an important and not clearly avoidable distinction. However, the interface might provide better support for understanding and choosing between these two modes.
- (The Draw and Type modes aren't so critical, because clicking on the drawing panel or font panel automatically changes the mode to Draw or Type, respectively.)
- (Note also that in most drawing packages, the default mode is Select; it is also the default mode right after drawing or typing, so the user really doesn't need to think about modes explicitly. In Esquisse, having two seemingly similar (and explicit) modes is confusing.)

(13) Click on the red rectangle.

- This is pretty standard graphics-package behavior.

(14) Click on the Edit button.

- This also seems ok as an action. See step 11 for the knowledge required to choose to edit the cell (rectangle).

(15) In the event function, realize or infer that in `MouseDrag _ dx dy`, the `dx` and `dy` are the relative x and y distances that the mouse was dragged.

- The HU might have this knowledge from explanation or experience, or she/he might infer it from MouseDrag and the mnemonic names dx and dy. Also, the HU might have dragged an Esquisse object with the mouse pointer. In any case, this step is critical to understanding 16.

(16) In the Ok part of the TwoFacedObject in the MouseDrag case of the event function, change `((x+dx, y+dy), (w,h))` to `((x,y),(w,h+dy))`.

- The HU must understand that she/he wants the height `h` to be "changed" by the event function, and that it should change by the vertical relative distance `dy`.
- This is one of the more subtle changes, because the HU must convert the event function from moving an object to resizing it (in the vertical dimension only; in practical terms, this means combining the height `h` with the vertical displacement `dy`).

(17) Change the argument to `show` in the TwoFacedObject part of the MouseDrag case in the same fashion, from `((x+dx, y+dy), (w,h))` to `((x,y),(w,h+dy))`.

- Here, the HU must understand that the first and third elements of a TwoFacedObject are "value" (evaluated) and textual (unevaluated) versions of the same entity, and thus must be kept in agreement. The HU must also know about `show`.

(18) Leaving the editor for the red rectangle open, click on the number 212 (in Select mode) and click the Edit button.

- This should be ok given prior experience in steps 13, 14.

(19) Notice that the name of this cell is `expression_2`.

- This unlikely to be problematic; the HU is needs the name to use in step 23. The HU can open the editor again if she/he forgets the name or forgets to look at the name.

(20) In Formula 1 Source Cells, type `rectangle_1`.

- The the red bar cell is named `rectangle_1` (the HU knows this from its editor).

(21) In Formula 1 Formula, type `snd.snd`.

- This number cell should have the same value as the height of the red rectangle cell. The height of the red rectangle is the second component of the second pair, hence `snd.snd`. This should be all in a day's work for a functional programmer.

(22) Click Close to close the editor for the number cell under the red rectangle

- This should be ok given basic interface experience.

(23) In the editor for the red rectangle, in Formula 1 Source Cells, type `expression_2`.

- The the red bar cell is named `rectangle_1` (the HU knows this from its editor).

(24) In Formula 1 Formula, type `\ h -> ((65,291),(9,h))`.

- The rectangle's height should be the same as the value of the number cell. The origin `(65,291)` and width `9` are constants. This should be ok for a functional programmer.

(25) Click Close to close the editor for the red rectangle.

- This should be ok given basic interface experience.

(26) Edit the numeric cell and drag the rectangle to test the linking of the rectangle and number.

- This should be ok given basic graphical interface experience.

(27) Edit the blue rectangle and its numeric cell as in steps 13 to 26, with the following differences in detail: for `expression_2` read `expression_4`, for `rectangle_1` read `rectangle_3`, and in step 24, use `\ h -> ((128,289),((10,h)`.

- These steps should pose no additional challenges.

(28) Reopen the editor for the numeric cell under the red rectangle, and in Formula 2 Source Cells, type `expression_4`.

- The HU must recall (or confirm) that the numeric cell under the blue rectangle is named `expression_4`. The HU must also know that the source cells are the arguments to the formula function, and that their names must be entered into the Source Cells box.

(29) In Formula 2 Function, type `\ c -> round ((9.0/5.0)*(fromInteger c)) + 32`.

- This is the formula for converting Celsius temperatures to Fahrenheit; since the numeric cell under the red rectangle holds the Fahrenheit temperature (and the height of the red rectangle), the formula must convert from Celsius to Fahrenheit. (The formula does require an awareness of the distinction between floating-point and integer numeric classes in Haskell.)

(30) Click Close to close the editor for the number cell under the red rectangle

- This should be ok given basic interface experience.

(31) Reopen the editor for the numeric cell under the blue rectangle, and in Formula 2 Source Cells, type `expression_2`.

- See step The HU must recall (or confirm) that the numeric cell under the red rectangle is named `expression_2`. The HU must also know that the source cells are the arguments to the formula function, and that their names must be entered into the Source Cells box.

(32) In Formula 2 Function, type `\ f -> round ((5.0/9.0)*(fromInteger (f-32)))`

- This is the formula for converting Fahrenheit temperatures to Celsius; since the numeric cell under the blue rectangle holds the Celsius temperature, the formula must convert from Fahrenheit to Celsius. (Again, the type classes of Haskell come into play.)

(33) Click Close to close the editor for the number cell under the blue rectangle.

- This should be ok given basic interface experience.

(34) Test the example by dragging the red and blue thermometer bars, and by typing into the numeric cells.

Appendix F

Line-Fitting Walkthrough in Esquisse

A screen image of the final line-fitting example in Esquisse appears in figure 4.6.

Line-Fitting Walkthrough in Esquisse

(1) Click on filled circle radio button, and on the blue color button.

- The palette offers filled circles, which seem appropriate for drawing the points. (Squares are a somewhat less likely possibility; using them would be at most slightly more challenging.)

(2) Draw a small circle in the work area.

- This should be ok given basic drawing package knowledge. (The lack of automatic refresh in Haggis can be a little confusing.)

(3) Recall that the points will be either inliers or outliers, and decide to store that status in the cell's value (as a Bool).

- The problem statement specifies that the points change color when clicked, and that the line will be fitted to the inliers (in blue; the outliers will be in yellow). Since the points must be two different colors, it seems natural to store the inlier/outlier status in the point cells.

(4) Click on the Select mode button, then the circle, then the Edit button.

- See the Temperature Conversion walkthrough for a discussion of the issues surrounding the Select button. Clicking the circle and the Edit button should be ok given basic interface knowledge.

(5) In the editor, notice that the value is `((93,157),5)`. Recall (or infer) that this is ((x,y),r) — the center and radius of the circle. Also, notice that the name of the cell is `circle_1`.

- The HU needs to understand that the displayed circle is a representation of the cell value. The HU needs to know or to infer that the value of a circle cell drawn with the interface is ((x,y),r). The HU will also need to know the name of the cell for use in the formula of the point list cell 23, but she/he can open the editor again if necessary. [The best solution would be to be able to specify formula sources by direct manipulation.]

(6) Decide to add the Boolean value to the cell as the second element of a pair, with the existing value as the first element.

- This solution leaves the point unaltered, which is least disruptive if the point is used in other computations. (References to the point need only be prefixed by `fst`).

(7) Change the cell value to `(((90,96),5),True)`.

- `(90,96)` is the first pair in the data series. A radius of 5 seems reasonable.
- The HU should have no problem editing the value given that she/he understands it. Understanding that it is a pair (whose left element is a pair) is helpful but probably not necessary.
- [An interesting unanticipated shortcoming of Esquisse is the lack of aggregate operations on cells. In this case, the HU might be greatly aided by the opportunity to apply `(map (\ p -> (p,True)))` to a list of the ten circles (of course, the display and event functions would have to change accordingly). Another example more closely aligned with Haggis: `(map (circleCellOfRadius 5) listOfPairs)`.]

(8) In the Ok case of the display function, replace `((x,y),r)` with `(((x,y,),r),b)`.

- This case must match (i.e., have the same type as) the (Ok version of the) argument of the display function, and the argument of the display function is the cell value.

(9) Change `Just blue` to `if b then Just blue else Just yellow`.

- `Just blue` specifies the color of the circle. The Boolean value of `True` for `b` indicates that the point should be blue, while a value of `False` for `b` indicates that the point should be yellow.

(10) Change the Ok case of the event function from `((x,y),r)` with `(((x,y,),r),b)`.

- As in step 8, this argument to the event function must be the same type as the cell value.

(11) Change the `Ok` case in the `MouseDrag` case from `((x+dx,y+dy),r)` to `(((x+dx,y+dy),r),b)`.

- The event function returns a TwoFacedObject whose first and third elements are "value" (evaluated) and textual (unevaluated) versions of the same entity, and thus must be kept consistent. The Ok case returns the value, which must have the same type as the cell value.

(12) In the same way, change the argument to `show` in the MouseDrag case from `((x+dx,y+dy),r)` to `(((x+dx,y+dy),r),b)`.

- The `show` expression returns the textual form of the value, and it must be kept consistent, too. [Note: there is no enforcement of this, and so the value and textual versions could get out of sync.]

(13) Notice the "unresolved top-level overloading" error. Infer from the type signature presented in the error message (and possibly from the program text itself) that the types of `r` and `b` are the one that can't be figured out, and that `show` is where the overloading issue is occurring.

- This is a rather subtle issue, even (I think, anyway) for functional programmers. Gofer's message (passed on by Esquisse) provides the context `Text (((Int,Int),-_171),_164)`. The context Text indicates that `show` is where the overloading needs to be resolved, because `show` is a (heavily) overloaded function in the `Text` class — but the HU needs to know that (or at least guess that `Text` and the function `show` that returns text strings are related). Although `show` is a common function, this error might not crop up much for many functional programmers [this is a guess, of course].
- To identify `r` and `b` as the unresolved variables, the HU must pattern-match the expression against `(((x,y),r),b)` and guess that `_171` and `_164` are unresolved type variables that correspond to `r` and `b`. The pattern-matching is a reasonable expectation of a functional programmer, but the type variable names (while common in error messages) are unnecessarily obscure
- [Also, Esquisse's presentation is messy, because it doesn't resize the picture when the error message is added (and also because it just dumps Gofer's message in there verbatim).]

(14) Decide to continue with the modifications to the event function and resolve the overloading later, if necessary.

- To make this choice, the HU must know that overloading is a property of the function as a whole, and thus the error need not be addressed until the function is fully written. [I think that this is a reasonable choice for a functional programmer to make, but I'm not sure just how reasonable.]

(15) Add a new case under `case eventType` as follows:

```
MouseButton Down _ ->
  TwoFacedObject
    (Ok (((x,y),r), not b))
    NoCursor
    [show (((x,y),r), not b)]
```

- The HU needs the event function to change the color of the point and toggle its status when a mouse press occurs. The event that is reported when a mouse click occurs is MouseButton Down followed by the button number. There is no case in the event function for a mouse press, so one must be added. The case must return a TwoFacedObject (see steps 11 and 12), whose value and textual representation should be consistent and of the same type as the cell value. Toggling the Boolean value in the point changes both the status of the point and its color. The Boolean function not "toggles" the Boolean value.

(16) Notice the "unresolved top-level overloading" error again. Infer that the type of `r` is still unresolved with respect to `show`.

- See step 13. In this case the context is `Text (((Int,Int),_254),Bool)` — the case added in step 15 resolved the type of `b`, but `r` is still unresolved.

(17) In the `show` expression, replace `r` with `(r::Int)`.

- The HU must know that (a) type annotations can be provided to variables, (b) that providing such an annotation will resolve the overloading problem, and (c) that `r` is an `Int`. Of these, (c) should be no problem, but (a) and (b) are relatively unusual (as is the overloading problem itself). [Unresolved overloading is more of a problem in Gofer than Haskell. Would this error occur in (for example) Hugs?]

(18) Repeat steps 1 through 17 nine more times (changing each value to a different member of the data series), until ten points are placed in the work area.

- Repeating this procedure should be unchallenging (except for the tedium of editing; a Copy feature would make a big difference).

(19) Recognize that the individual cell values need to be combined somehow into a list.

- The fittedLineAndCorrelation function needs a list of the points in order to compute the fitted line and the correlation. All that exists now is a scattering of individual cells, with their individual values.

(20) Decide to create a cell containing a list of the individual values. Decide to use an expression cell.

- Cells can hold arbitrary data structures, including lists. [There really should be better support for aggregations of cell values, either via container cells or in some other fashion.] An expression cell is the simplest way to create a cell containing an arbitrary value. See the Temperature Conversion Walkthrough for a discussion of the issues surrounding Text and Expression cell creation modes.

(21) Click on the Expressions button, and on the black color button.

- See the Temperature Conversion walkthrough for a discussion of the issues surrounding the Text and Expression buttons.

(22) Click to the right of the points (a text cursor appears). Click Select, the text cursor, and then Edit. Notice that the name of the cell is `expression_11`.

- Creating the empty expression cell and opening the editor should be ok. To refer to the list of points in the fitted-line-and-correlation formula, the HU must know its name. [Here, in particular, renaming would be a helpful feature.]

(23) In the editor, in Formula 1 Source Cells, type `circle_1 circle_2 circle_3 circle_4 circle_5 circle_6 circle_7 circle_8 circle_9 circle_10`.

- The formula needs to create a list of cell values from the values of the point cells. The Source Cells box lists the source cells for the formula. This step should be ok provided that the HU understands this knowledge about source cells. This step is where the names the HU saw in (repeated) step 5 are used.

(24) In the editor, in Formula 1 Function, type `\ c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 -> [c1,c2,c3,c4,c5,c6,c7,c8,c9,c10]`.

- This lambda expression creates a list from its (ten) objects.
- Here, another problem can crop up. If the HU mistypes the list, the type of the function might be inferred prematurely. This is normally not a problem, but in this case the desired type of the function is `a -> a -> a -> a -> a -> a -> a -> a -> a -> a -> [a]`. A typo, for instance, `10` for `c10` in the list, could result in the type becoming specialized to `Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> Int -> [Int]`. Since the base type of the point cells is not `Int` but `(((Int,Int),Int),Bool)`, this will cause a formula error.

(25) Close the editor for the list expression cell.

- This should be ok given interface experience.

(26) Infer that result of the fittedLineAndCorrelation function needs to be split into two cells somehow.

- The fittedLineAndCorrelation function returns a pair consisting of the fitted line and the correlation. The line needs to be displayed graphically, the correlation textually, which suggests two cells.

(27) Decide to use an expression cell to store the line and correlation as a pair, and to use separate cells connected by formulae to display the line and correlation.

- An expression cell is the simplest cell that contains an arbitrary value (compare step 20).

(28) Click on the Expressions button, then click to the right of the points and above the point list expression cell.

- See step 21.

(29) Recall that the fittedLineAndCorrelation function needs not only the list of points, but also the clipping rectangle ((x1,y1),(x2,y2)).

- The HU might not think of this before starting to enter the formula, but doing so now simplifies the walkthrough without loss of information.

(30) Decide to draw the clipping rectangle in the work area.

- The clipping rectangle could be just typed in to the formula, which would probably be faster, but it can be moved (but not (by default) resized) if drawn in the work area. Also, drawing the clipping rectangle will help frame the points (and line).

(31) Click on the square outline button, and on the black color button, and draw a square around the points.

- This should be ok given basic interface and graphics package experience. The HU might also choose a rectangle.

(32) Click the Select button, click on the square, click the Edit button, read the name of the square cell (`square_14`), and click the Close button to close the editor.

- To refer to the clipping rectangle in a formula, the HU must know its name, which she/he can find out by opening the editor. [This is clunky; as mentioned in step 5, specifying source cells by direct manipulation would be smoother.]

(33) Click on the empty fitted-line-and-correlation cell (that is, the text cursor above the list of points), and click the Edit button.

- This should be ok given basic interface and graphics package experience.

(34) In the Formula 1 Source Cells box, type `expression_11 square_14`.

- The HU must understand that the Source Cells box must contain the names of the cells whose values will be the arguments to the formula function. The list of points and the clipping rectangle are the two cells to whose values the fittedLineAndCorrelation function will be applied.

(35) In the Formula 1 Function box, type this expression:

```
\ listOfPoints ((x,y),s) ->
    fittedLineAndCorrelation
      (map (fst.fst)
             (filter snd listOfPoints))
      ((x,y),(x+s,y+s))
  where
    fittedLineAndCorrelation theCoordinateList
                          theClippingWindow =
      (((round x1, round y1),
        (round x2, round y2)),
       correlation)
    where
     ((x1, y1), (x2, y2)) =
         clipToRectangle
           (slope, intercept)
           ((\ ((a,b),(c,d)) ->
             ((fromIntegral a, fromIntegral b),
              (fromIntegral c, fromIntegral d)))
            theClippingWindow)
      (slope, intercept, correlation) =
        slopeInterceptAndCorrelation $
                zip (map (fromIntegral . fst)
                         theCoordinateList)
                    (map (fromIntegral . snd)
                         theCoordinateList)
      slopeInterceptAndCorrelation ::
        [(Float, Float)] -> (Float, Float, Float)
      slopeInterceptAndCorrelation xys =
        (slope, intercept, correlation)
      where
        intercept =
          (sum ys) / (fromIntegral n)
          - slope * (sum xs) / (fromIntegral n)
        slope = sxy / sxx
        correlation = sxy / sqrt (sxx * syy)
        sxx =
          sum (map square xs)
          - (square (sum xs)) / (fromIntegral n)
        syy =
          sum (map square ys)
          - (square (sum ys)) / (fromIntegral n)
        sxy =
          sum (zipWith (*) xs ys)
          - (sum xs) * (sum ys) / (fromIntegral n)
        square n = n * n
        (xs, ys) = unzip xys
        n = length xys

      clipToRectangle :: (Float, Float)
                      -> ((Float, Float),
                          (Float, Float))
                      -> ((Float, Float),
                          (Float, Float))
```

```
-- line is (slope,intercept)
-- rectangle is ((left,bottom),(right,top))
-- returned line segment is
--  ((x1,y1),(x2,y2))  with  y1 <= y2

clipToRectangle
    (slope, intercept)
    ((left, bottom), (right, top))
  {- given that y1 <= y2 -}
  | y1 > top                    {- no line -}
    = ((0.0, 0.0), (0.0, 0.0))
  | y1 < bottom && y2 < bottom {- no line -}
    = ((0.0, 0.0), (0.0, 0.0))
  | y1 < bottom && y2 > top    {-  both   -}
    = (( (bottom - intercept) / slope,
         bottom),
       ( (top - intercept) / slope,
         top))
  | y1 < bottom                 {- y1 only -}
    = (( (bottom - intercept) / slope,
         bottom),
       ( x2,
         y2))
  | y2 > top                    {- y2 only -}
    = (( x1,
         y1),
       ( (top - intercept) / slope,
         top))
  | otherwise                   {- neither -}
    = ((x1, y1), (x2, y2))

where (x1, y1, x2, y2)
        | slope >= 0
          = (left,
             slope * left + intercept,
             right,
             slope * right + intercept)
        | otherwise
          = (right,
             slope * right + intercept,
             left,
             slope * left + intercept)
```

- The fittedLineAndCorrelation function calculates the fitted line and the correlation from the list of data points and the clipping rectangle. This seems ok for the HU given that the HU must have this level of understanding of the computation to want to write this example in the first place. Since the value of `square_14` is the lower-left corner and the side length, and the clipping rectangle passed to fittedLineAndCorrelation must be the lower-left and upper-right corners, the upper-right corner must be calculated. The HU might miss this, but it's easy to fix (in the worst case, the formula will show a type error, and can be reedited).

- Typing the formula in should be ok, although the editor will not, in fact, expand to hold it all; and if it did, it would extend off the screen. [So Esquisse needs a more practical

way to support the entry of large computations (if Haggis offered multiline text entry, that would help). Supporting X-selection pasting, input of files into cell fields (such as the formula function field), and supporting importing from / linking with Haskell files would all be useful.]

- If the fittedLineAndCorrelation function were built in or linked in, the formula would instead be simply

```
\ listOfPoints ((x,y),s) ->
  fittedLineAndCorrelation (map (fst.fst)
                                (filter snd
                                 listOfPoints))
                           ((x,y),(x+s,y+s))
```

(1) Notice that the name of the cell is `expression_13`.

- This seems ok given interface experience.

(2) Close the editor window by clicking on the Close button.

- This seems ok given interface experience.

(3) Now draw the line cell by clicking on the line button, clicking on the red color button, and drawing the line over the points.

- This seems ok from the standpoint of graphics package/interface experience. The HU might have a little trouble remembering what step is next.

(4) Open the editor for the line cell by clicking on the Select button, clicking on the line, and clicking on the Edit button.

- This seems ok given interface experience and repeated practice in this example.

(5) In the Formula 1 Source Cells box, type `expression_13`.

- The Source Cells box contains the cell names to whose values the formula will be applied. Extracting the fitted line endpoints from the pair containing the fitted line and the correlation requires a function. This all seems ok given the HU's knowledge of simple functional programming.

(6) In the Formula 1 Function box, type `fst`.

- The fitted line endpoints are the first element of the pair that is the value of the `expression_13` cell. The `fst` function returns the first element of the pair to which it is applied.

(7) Close the editor by clicking on the Close button.

- This seems ok given interface experience.

(8) Now create a cell to hold the correlation by clicking on the Expression button, and clicking above the points (a text cursor appears).

- This seems ok given interface experience and practice during this example.

(9) Open the editor for the correlation cell by clicking on the Select button, clicking on the line, and clicking on the Edit button.

- This seems ok given interface experience and repeated practice in this example.

(10) In the Formula 1 Source Cells box, type `expression_13`.

- The Source Cells box contains the cell names to whose values the formula will be applied. Extracting the correlation from the pair containing the fitted line and the correlation requires a function. This all seems ok given the HU's knowledge of simple functional programming.

(11) In the Formula 1 Function box, type `snd`.

- The correlation is the second element of the pair that is the value of the `expression_13` cell. The `snd` function returns the second element of the pair to which it is applied.

(12) Close the editor by clicking on the Close button.

- This seems ok given interface experience.

(13) Test the example by clicking on the points and seeing the line and correlation change.

- The problem here is that the redrawn line's bounding box ends up covering the points, so to click on the points, the HU must first select the line, click the Back button, then click the Interact button. (It's also necessary to do this to the clip rectangle, if it ends up in front.) Worse, (a) deselecting a graphic object also moves it to the front (because the picture is recreated), so objects to be kept in the back must be intentionally (and counterintuitively) left selected; (b) each formula evaluation redraws the line and sticks it in front of the points again.