

The Technology Behind a Graphical User Interface for an Equational Reasoning Assistant

Andy Gill

Department of Computing Science, University of Glasgow

Abstract

The Haskell Equational Reasoning Assistant (HERA) is an application written in Haskell that helps users construct and present equational reasoning style proofs[1]. In this paper we discuss the technology behind the user interface.

1 Introduction

HERA is an equational reasoning *assistant*. It assists the user by taking requests for actions to be performed on partially complete proofs. Typical requests that the user might make include:

- Use *this* specific lemma, on *this* specific sub-expression.
- Perform case analysis on a formula, splitting it into a set of formulas.
- Specify a new lemma to prove.
- Request a reminder of the exact form of a lemma.

HERA itself looks after the book-work, like what lemmas remain unproven, performing α -conversions, etc.

A first prototype of HERA was constructed with a straightforward ascii interface. This highlighted two significant problems:

- The first and most important shortcoming of this interface was the problem of sub-expression selection. It was unclear how the user could say, in a straightforward manner, “inside *this* sub-expression”.
- Furthermore, it was difficult to get a clear overall picture of the direction of a proof within an ascii interface. An ascii screen is too small to hold both the proof in progress, and respond to the requests for available lemmas.

To address both these problems we have implemented a multi-window implementation, with a point-and-click style of sub-expression selection. Figure 1 gives picture of an equational proof in action. When assisting with a proof, HERA gives the operator two windows, one giving derivations starting at the left hand side, and other derivations from the right hand side. When the two sides meet, the proof is complete. The rest of this paper explains the internals of this graphical user interface.

Figure 2 gives an overview of the main components of HERA. In this paper we are concerned with the components that connect HERA to the display.

- TK/TCL and **XLib** is a separate process, running the TK/TCL shell, wish. The component **Wish I/O** links the implementation of HERA with wish. We talk about these components in Section 2.
- The **Widget Library** is layer of abstraction on top of **Wish I/O**. We talk about this in Section 3.

In Section 4 we discuss how we manage to pretty print with colours and different fonts. In Section 5 we present the algorithm for sub-expression selection that we use in HERA.

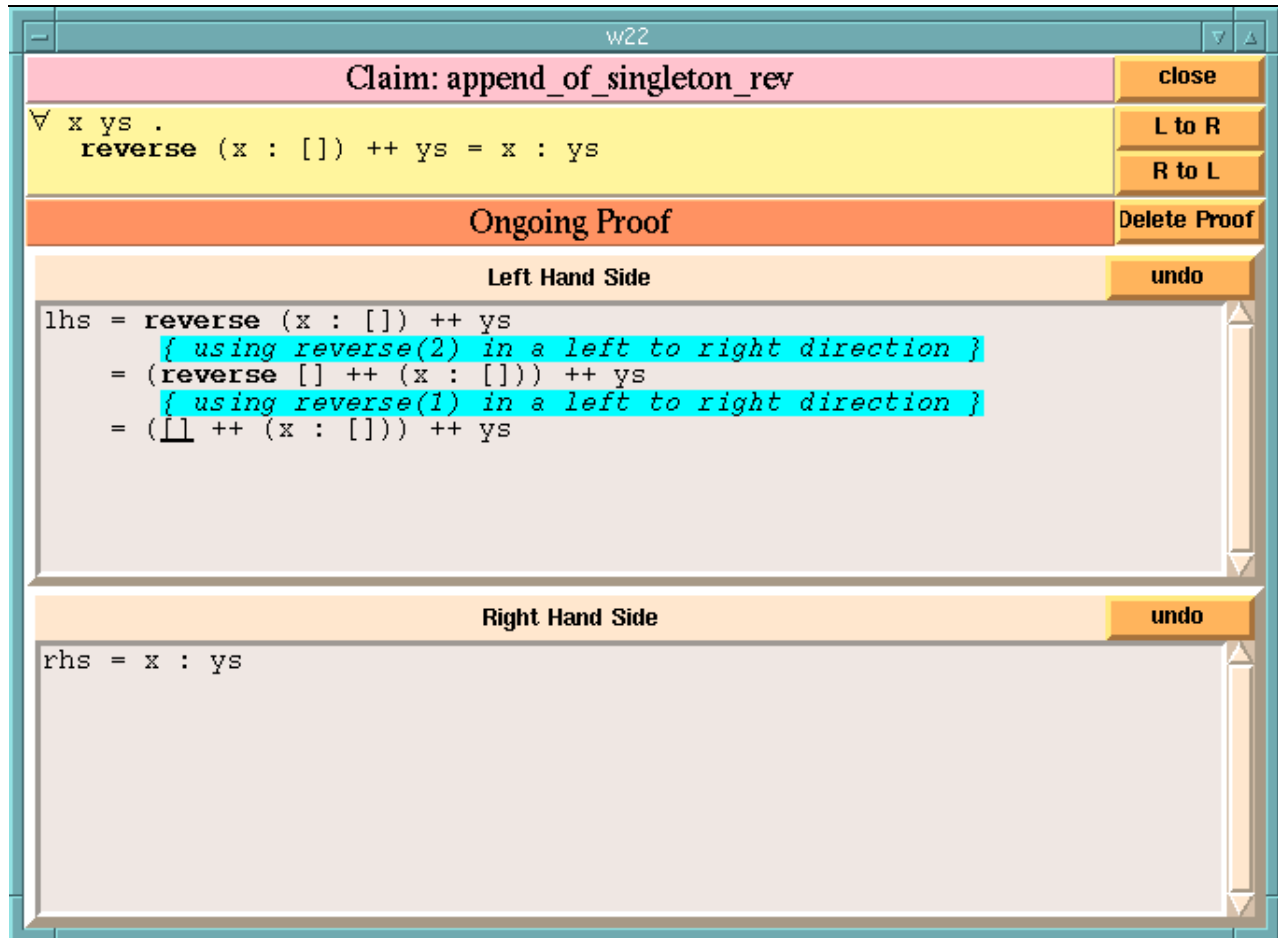


Figure 1: HERA in use

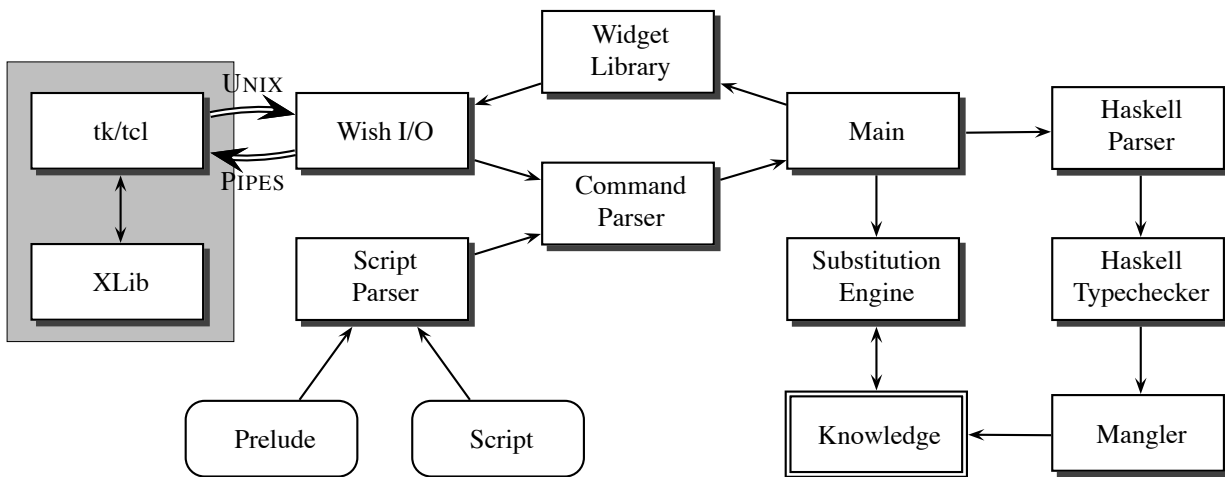


Figure 2: The internal structure of HERA

2 Talking to X via TK/TCL

TK/TCL is a simple scripting language that interfaces with X to provide high level commands for building graphical interfaces. Here is a simple example of TK/TCL program to print a button widget on the screen:

```
button .hello -text "Press Me" -command { puts stdout "someone pressed me" }
pack .hello
```

This displays a button:



When this button is pressed, the program says “someone pressed me”.

HERA is written in Haskell 1.3. Rather than use a graphical interface that would be tied into a particular implementation, we use a system of UNIX pipes into a process running TK/TCL [3] which handles the interaction with X. We use a simplified version of the model used in [4] with only two pipes, one for requests, and one for responses, both feeding directly into TK/TCL. The three key Haskell functions have the types:

```
> initWish      :: IO Wish
> sendCommand  :: Wish -> String -> IO ()
> waitForEvent :: Wish -> IO String
```

`initWish` sets up the pipes, giving us the `Wish` handle. `sendCommand` sends a command to TK/TCL, and `waitForEvent` waits for something to happen, returning the response from TK/TCL.

To execute the above program in Haskell, we could write:

```
> main =
>   initWish                                     >>= \ wish ->
>   sendCommand wish ("button .hello -text \"Press Me\" " ++
>     "-command { puts stdout\ "someone pressed me\" }") >>
>   sendCommand wish "pack .hello"              >>
>   waitForEvent                                 >>= \ resp ->
>   putStr resp
```

3 Our Widget Library

The model used by TK/TCL is an imperative one. We build onto this imperative base a widget library that provides functional combinators for building widget hierarchies. For example, our function for creating a button widget has the type:

```
> buttonW
>   :: String      -- label
>   -> String      -- what you reply
>   -> Widget
```

This function models itself on the TK/TCL `button` command, but provides a functional interface. We place the widget onto the screen using a monadic function `addWidget`:

```
> addWidget :: Wish -> WidgetName -> Widget -> IO ()
```

So rewriting our “button” program using our widget library gives:

```
> main =
>   initWish                                     >>= \ wish ->
>   let
>     button_widget = buttonW "Press Me" "someone pressed me"
>   in
>     addWidget wish defaultWName button_widget   >>
>     waitForEvent                                 >>= \ resp ->
>     putStr resp
```

We still use `waitForEvent` to wait for something to happen, but the construction of our widget is much neater.

In the widget library we also provide combinators for joining smaller widgets to make larger widgets. For example:

```
> aboveW   :: Widget -> Widget -> Widget
> besideW  :: Widget -> Widget -> Widget
> stackOfW :: [Widget] -> Widget
> rowOfW   :: [Widget] -> Widget
```

So if we wanted two buttons, side by side we could write:

```
> buttons_widget =
>   buttonW "Press Me" "someone pressed me"
>   'besideW'
>   buttonW "Or Press Me" "someone pressed button 2"
```

Using this straightforward widget library, we build the necessary pictures to provide our interactive interface.

4 Printing more than Ascii

The TK/TCL system supports extended and alternative character sets, along with various fonts, underlining, and colours. We wanted to use these to aid the presentation of our proofs and lemmas. The current technology for printing abstract syntax trees, called pretty-printers, output ascii pictures. We augmented one pretty printer [2] to support our extended styles of characters. This pretty printer has the following interface:

```
> data Doc = ...
> text      :: [Char] -> Doc           -- Basic Word
> ($$)      :: Doc -> Doc -> Doc       -- Vertical composition
> (<>)      :: Doc -> Doc -> Doc       -- Horizontal composition
> pretty    :: Int -> Int -> Doc -> [Char] -- Building final String
```

Using this library it is straightforward to construct clear and easy to read ascii representations of abstract syntax. There is no support, however, for the extensions that we want to use, like colour, fonts, and emphasis. To allow this, inside HERA we have abstracted our pretty-printing library over the ‘characters’ that the library outputs. We use type classes to do this. First we define a class `Pretty`:

```
> class Pretty a where
>     blank_space :: a
```

The idea behind this class is that for anything that we supply an instance of `Pretty`, we can use our augmented pretty-printing library. By simply “teaching” the library what we mean by white-space, our “characters” can be any datatype. To provide backwards compatibility we provide an instance for `Char`:

```
> instance Pretty Char where
>     blank_space = ' '
```

Now our library has the following interface:

```
> data Doc a = ...
> text      :: Pretty a => [a] -> Doc a
> ($$)      :: Pretty a => Doc a -> Doc a -> Doc a
> (<>)      :: Pretty a => Doc a -> Doc a -> Doc a
> picture   :: Pretty a => Int -> Int -> Doc a -> [[a]]
```

We have replaced `pretty`, which outputted a 1-dimensional list of `Char` (including newlines), with `picture` that outputs a 2-dimensional “array” of our arbitrary characters.

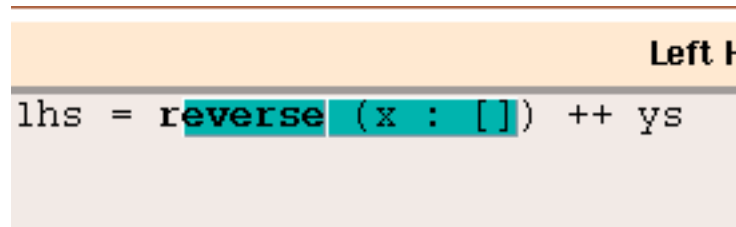
So what do our new characters look like inside HERA? We use the definition:

```
> data HyperChar      -- our special version of Char
>     = HyperChar
>     Char             -- the char 'a'/'b'/'c'/etc
>     [HyperCharStyle] -- Which Font/Boldness/Colour/etc
>     (Maybe ExpAddress) -- the address of *this* expression
>                                     -- on the original syntax tree
```

Inside each `HyperChar` there is a real ascii `Char`, as well as information about how to display this char, for example what font to use, what colour to use, etc. Using this technology we can provide both boldness and alternative fonts, as well as support characters like \forall , as Figure 1 demonstrates. In the next section we explain the `(Maybe ExpAddress)` component of `HyperChar`, as it is used to allow selection of sub-expressions.

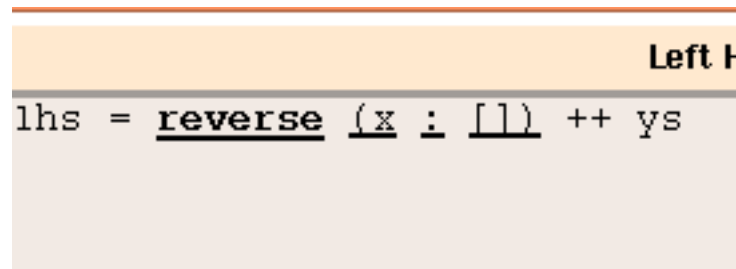
5 Selection of Sub-expressions

When we perform a proof interactively, we need a way of selecting an expression or sub-expression for substitution. We use the mouse to select appropriate expressions:



```
lhs = reverse (x : []) ++ ys
```

and HERA selects `reverse (x : [])` and underlines it, indicating that this is the currently selected sub-expression. Now this selected expression can have substitutions performed on it. Notice also that an exact match was not required, but the sub-expression selection algorithm deduced the correct selected expression.



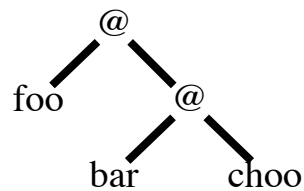
```
lhs = reverse (x : []) ++ ys
```

So how can this be implemented? TK/TCL can be instructed to send a string like

```
newDefaultExpression rev_of_singleton left 1.2 1.15
```

when a new sub-expression is selected. But the problem of identifying the sub-expression inside the original abstract syntax tree remains. We will explain our algorithm by using an example.

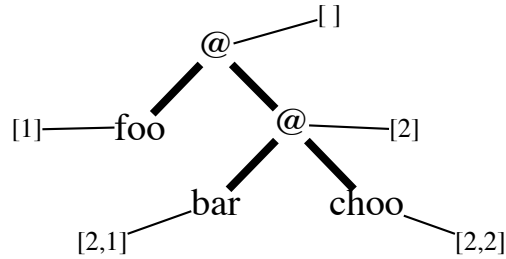
Consider this simple expression tree, where @ is application:



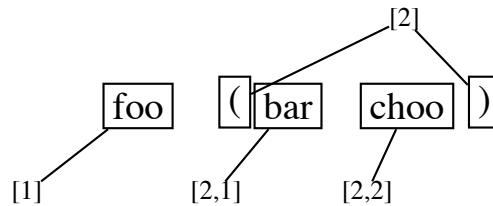
This tree, when pretty printed, becomes:

```
foo (bar choo)
```

Now, with each component of the original tree we associate an “address”.



The address of each component is simply the route required to reach it, starting from the top of the tree. For example, to find the expression “bar choo”, you go down the second branch, so the address is [2]. To find the expression component “bar”, you go down the second branch, then go down the first branch, so the address is [2 , 1]. So, the string “foo (bar choo)” has a tag on each non-whitespace character:



We use the (Maybe ExpAddress) component of HyperChar to add addresses to each character. When a region is selected, we use the following algorithm to find the appropriate sub-expression:

- Take the left most marker, and if it has no associated address, keep moving this marker to the right.
- Do the same with the right most marker, moving it left.
- At this point both markers have an address associated with them. We now take the most specific common address. For example, [1 , 2] and [1 , 2] would common up to [1 , 2], while [1 , 2] and [1 , 1] would common up to [1], and [1] and [2] would common up to [], the whole expression.

This algorithm has been found to work well in practice, with the program correctly finding the expression the user selected.

6 Conclusion

In this paper we have overviewed the technology behind the GUI in HERA.

- We have seen how a reasonably sophisticated interface can be created in portable Haskell by using pipes the the TK/TCL graphics toolkit. A widget library that exploited functional composition is provided on top of the TK/TCL primitives.
- We have used type classes to augment a pretty printer in a general way, allowing the use of fonts, colours, etc. inside our presentation of our abstract syntax.
- We have used a simple addressing scheme for our abstract syntax trees, and tag the presentation information internally with these addresses. This allows a straightforward mapping from a two dimensional representation of an abstract syntax tree, back to the tree itself.

These three aspects are all general purpose, and not specific to our application.

References

- [1] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice-Hall, 1988.
- [2] R. J. M. Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Proceedings of the First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *LNCS*. Springer-Verlag, May 1995.
- [3] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1995.
- [4] Duncan Sinclair. Graphical user interfaces for Haskell. In John Launchbury and Patrick M. Sansom, editors, *Glasgow Workshop on Functional Programming, Ayr, Scotland*, Workshops in Computing, pages 252–257. Springer-Verlag, July 1992.