wxFruit: A Practical GUI Toolkit for Functional Reactive Programming

Bart Robinson
bartholomew.robinson@yale.edu
May 2004

Abstract

The Haskell programming language has suffered in the past from the lack of a full-featured graphical user interface (GUI) toolkit. The situation has been improved recently by the introduction of wxHaskell, a library built upon the widely-used and feature-rich C++ GUI toolkit wxWidgets. However, the programming idioms used by wxHaskell are decidedly imperative, making wxHaskell a problematical choice for those who wish to keep with the functional and highly formal programming paradigms supported by Haskell.

The need for a "genuinely functional" user-interface toolkit has been addressed in the Fruit library, which brings the concepts of Functional Reactive Programming (FRP), a programming paradigm designed specifically for dynamic systems like user interfaces, to the world of GUIs. Fruit allows for a very elegant, declarative style of GUI programming in Haskell. However, the implementation of Fruit has been hindered by its reliance on a low-level graphics library to draw graphical elements, instead of interfacing to the host platform's GUI API. Thus Fruit, up till now, has been able to offer only a tiny fraction of the features available in most modern user interfaces.

In this paper I present my attempt to address the drawbacks of both wxHaskell and Fruit by replacing the implementation of Fruit with one that uses the GUI resources provided by wxHaskell. The resulting hybrid toolkit, called *wxFruit*, combines some of the power and versatility of wxHaskell with the elegance and simplicity of Fruit. I describe the obstacles faced in the design and implementation of wxFruit, discuss some of the issues that remain unresolved, and present a small application demonstrating the expressive power of the new toolkit.

1 Introduction

The work done by Courtney and others [1,2] to apply to the world of graphical user interfaces (GUIs) the principles of functional reactive programming (FRP) [7], a paradigm for the programming of reactive systems organized around the concept of time flow, resulted in a modest but promising GUI

toolkit for the Haskell language, dubbed Fruit.[1] Fruit is built upon the Yampa [4] framework, an implementation of FRP based on arrows [6,8]. It provides a set of simple GUI components ("widgets")—buttons, static text labels, and so forth—that can be combined on the screen to create functional, if not flashy, user interfaces. The advantages that the FRP approach in general, and Fruit in particular, give to the GUI programmer are explained in full in the above references, and Fruit's usefulness as a GUI backdrop for functional reactive applications that employ animation has been well-defended in [3].[2]

Fruit is very far from being an industrial-strength GUI toolkit, however. Perhaps the most difficult obstacle faced by further development of Fruit, as it is currently implemented, is the enormous overhead involved in adding new widgets to the (currently, severely limited) widget set. This overhead results from the fact that all of Fruit's widgets are created "from scratch," using a third-party graphics library and a Haskell-to-Java bridge. This approach may be satisfactory for a research prototype toolkit, but in a real-world setting a GUI programmer would likely prefer to have access to the entire widget set of the host platform, with widgets capable of exhibiting the platform's native look-and-feel.

On the other end of the spectrum of GUI offerings for Haskell lies wxHaskell.[3] This library leverages a popular and very stable C++ GUI toolkit, wxWidgets, to provide a wide range of features, including an extensive widget set with native look-and-feel. wxHaskell is under heavy development and has already been adopted for use in several applications. It promises to provide for the Haskell community the kind of standard GUI toolkit that has long been available to users of other languages.

Despite these advantages, there is a downside to wxHaskell's decision to base itself on wxWidgets. The nearly wholesale transfer of the programming model used by wxWidgets into the Haskell language performed by wxHaskell leaves the Haskell programmer with no choice but to use an imperative programming paradigm, managing state variables across calls to event handlers, for example. Not only would many Haskell users view this as contrary to the functional spirit of Haskell and the specification-like clarity that it allows, but this is exactly the kind of complicated and error-prone GUI programming that motivated the development of Fruit and similar projects.

It seems, then, that we might be able to simultaneously address the problems of the limited feature set of Fruit and the awkward programming paradigm of wxHaskell if we could somehow combine the two. An attempt to

---

[1] See `http://haskell.org/fruit/`

[2] In the following I assume that the reader is familiar with the concepts presented in these references, including the notion of arrows and the arrow syntax, signals and signal functions, and the most commonly-used functions and combinators in the Yampa library.

[3] See `http://wxhaskell.sourceforge.net/`

do just that is the subject of this paper. I describe a new implementation of Fruit, called wxFruit, which employs wxHaskell to handle the low-level details of GUI appearance and behavior. Pertinent details of the implementation and design decisions are discussed, and I make an attempt to evaluate the usefulness of the new toolkit in its current state, as well as address the need for future work on wxFruit.

2 Fruit, Reincarnated

In this section I report on the process of developing wxFruit. Along the way, I describe the form that this new version of Fruit has taken, and how it diverges from the original. This may serve as a case study for the development of systems which try to bridge the gap between the FRP framework and real-world software applications.

2.1 Reuse or Replace?

The first question that arose when thinking about wxFruit was whether any of the existing Fruit code could be reused, or whether a complete reimplementation of the toolkit was necessary. It was quickly decided that the reimplementation strategy was the best approach, because the original Fruit was tied too closely to the details of the supporting software libraries it used to create its own widgets. Antony Courtney, the original author of Fruit, had written a rough sketch of what a new implementation of Fruit that used wxHaskell would look like. I began with this sketch as my base.

2.2 Input and Output

Widgets in the original Fruit were signal functions that used auxiliary continuous input and output signals to communicate with the underlying system. That is, a widget in Fruit was represented by the type definition

```
type GUI a b = SF (GUIInput,a) (Picture,b)
```

where `GUIInput` represents the state of user inputs, such as keys on the keyboard and the position of the mouse, and `Picture` represents the graphical representation of the widget. In wxFruit, we are no longer concerned with these details—we would like for wxHaskell to handle things like key presses and the drawing of widgets on the screen. We structure our widgets, therefore, around a set of discrete *response* and *request* events, which only cause communication between widgets and the system to occur when absolutely necessary. wxFruit's version of the above type definition, then, becomes

`type Widget a b = SF (Event WidgetResp,a) (Event WidgetReq,b)`[4].

Widgets send request events when some part of their state has changed to tell the system to update the underlying wxHaskell component (which, in turn, sends an update to the host platform). Response events are sent to wxFruit widgets when something occurs on the system-side that the widget needs to know about, such as the user clicking on a button. Responses and requests are also used in a two-stage creation process for each widget: the very first thing a wxFruit widget does is send an event requesting that the system create a corresponding wxHaskell widget, which is then passed back to the widget via a response event. This mechanism allows synchronization between the widget, at the Yampa level, and the wxFruit system residing at the wxHaskell event handler level.

2.3 Layout

How do we specify a GUI than contains more than one widget? The original Fruit answered this question[5] by introducing a new type, `Box`, which built upon `GUI` and could be composed into horizontal or vertical sequences. We take the same approach in wxFruit, allowing the programmer to build up a tree of widgets that together specify a working GUI. This is done by adding alternatives within the `WidgetResp` and `WidgetReq` data types that act as nonterminal nodes in a compositional tree; at the top level, then, every response and request event is actually a tree, with leaf nodes the responses or requests directed at individual widgets.

Part of the state maintained by the wxFruit system is a *contents tree*, with leaf nodes the wxHaskell window components corresponding to each wxFruit widget. By traversing this tree, the system can respond appropriately to an entire tree of widget-request events. The system also uses the contents tree to build up the layout specification model required by wxHaskell, constructing rows and columns of widgets which are sent to wxHaskell to arrange on the screen.

2.4 Encapsulation

Not only does `Box` (in our case, `WXBox`) provide a simple but powerful mechanism for structuring the layout of a GUI, but using it instead of `Widget` also hides from the application programmer the details of the auxiliary input and output signals (i.e., response and request events) of the widgets. This is done, as it is in Fruit, by making `WXBox` a new instance of the `Arrow` class, with customized arrow combinators that do the work of wiring

[4] Some of the names of types and values in wxFruit have been simplified in the code examples given in this paper for the sake of readability.
[5] See [1] and compare with the earlier attempt at widget composition in [2].

the input and output signals within the widget tree correctly, sending responses and requests to the appropriate widgets. When the arrow syntax is used, this enables very clear and uncluttered specifications of GUIs, with the layout of widgets corresponding to the order in which they appear in the source code.

2.5 Reactimation

The above design considerations yielded fairly straightforward solutions. When it came to actually putting the Yampa-based widget code and the wxHaskell-based system code together, however, I hit a snag. The basic problem was that both wxHaskell's `start` function and Yampa's `reactimate` were designed to act as the main application loop. I could not use `reactimate` without sacrificing the message-pumping and event-handling capabilities of wxHaskell. I needed a way to run wxHaskell's main loop and simultaneously obtain periodic input samples for Yampa to send to my widgets.

Fortunately, Courtney had already thought about this problem, and I am indebted to him for adding and publishing his solution in the Yampa source code. This solution was to break `reactimate` into two non-blocking functions.[6] The first function, `reactInit`, performs initialization and returns a *handle* that can be passed to the second function, `react`, which processes a single input sample. Calling `react` at regular intervals, passing in the same handle every time, has the same effect as a single call to `reactimate`.

How do we cause repeated calls to `react` to occur, while still using wxHaskell for our main loop? The answer is found in wxHaskell's `timer` primitive. We simply create a recurring timer object, which we call the *heartbeat*, which calls an event handler at periodic intervals. Setting the interval to 30 ms gives us smooth animation as well as responsive widgets.

It is worthwhile to note that the timer event handler is not the only place from which `react` is called. The button event handler, which is called every time the user clicks on a button, also makes a call to `react`, sending in a response event directed at the appropriate button widget. In fact, any wxHaskell widget that gets input from the user will need an event handler that calls `react` in order to update the corresponding Yampa widget.

This diversity of entry points into Yampa does not create a problem, as `react` does not require that it be called at uniformly spaced intervals. What this does suggest, though, is a possible future optimization: since most of the widgets used in everyday applications do not change until the user performs some action—that is, they are not time-varying, like an animation or a interactive game—if we could find a way to only employ the heartbeat timer when widgets that are time-varying appear in the contents tree, then GUIs

[6] Or, to be more precise, two functions that return non-blocking IO actions.

which only change in response to user input would not have to carry the overhead of the timer event handler's recurring calls to `react`. This would be a challenging task, however, since the way that Yampa is currently structured, there is no way to determine whether a given signal function is time-varying or whether it only varies in its input signals.

2.6 User-defined Widgets

So far I have only discussed widgets that correspond to GUI components available in wxHaskell, that is, standard widgets like buttons and static text labels. We would also like to enable users to go beyond this standard set and create their own custom widgets—animations, games, or customized controls—by allowing them to draw arbitrary images on the screen and giving them access to the unprocessed mouse and keyboard input. This was simple in the original Fruit, since the Fruit `GUI` signal function took these unprocessed inputs as input signals and had a `Picture` as one of its outputs. In wxFruit, however, this functionality requires a little more work.

Since Fruit itself used the Haven graphics library to draw widgets on the screen, all of Haven's drawing primitives were available to users of Fruit to exploit when constructing custom images. It turns out that wxHaskell has a small set of drawing primitives as well, and it was simple to adapt these to Haven's functional style of graphics programming. This resulted in a small but effective graphics library residing within wxFruit, consisting of a set of wrappers around wxHaskell graphics primitives.

This library allows wxFruit users to create arbitrary images (as the new type `WXPicture`), but how do we turn these images into widgets, so that they can appear on the screen among other standard and user-defined widgets? This is what the new `wxpicture` widget in wxFruit does: via a constructor that takes a `WXPicture` signal, it turns an image into a widget that sends a request event to the system telling it to redraw the image whenever it changes.[7]

So much for graphics; what about unprocessed user inputs? The current solution in wxFruit is to add another input signal to the `Widget` type:

```
type Widget a b = SF (RawInput,Event WidgetResp,a)
                     (Event WidgetReq,b)
```

where `RawInput` is a type representing the state of the mouse and keyboard, very similar to Fruit's `GUIInput`. The `RawInput` signal is hidden from the user programmer in the same way as the response and request event signals. A set of widgets provided by wxFruit are able to extract information from the `RawInput` signal and make it available to the user programmer; `wxmouse`, for example, has as its output signal the current position of the mouse. Whether

[7] Actually, the current implementation just uses the quick-and-dirty approach of sending a redraw request at *every* sample time. But ideally this would only occur when the image is modified.

this solution is satisfactory or not has yet to be seen; the current implementation is somewhat incomplete in this area.

2.7 Proof of Concept: Paddleball

As a demonstration of how far wxFruit has come, and as visual evidence of the feasibility of merging functional reactive programming with a practical, real-world applications library like wxHaskell, I have adapted the Paddleball game first introduced by Hudak [5] and rewritten for the original Fruit in [2] to wxFruit. The code for the game itself remains essentially unchanged, as confirmation that the interface to wxFruit successfully mimics that of the original Fruit. As further proof of the usefulness of the new toolkit, I added the additional feature of a slider widget which allows the user to control the speed of the game.[8]

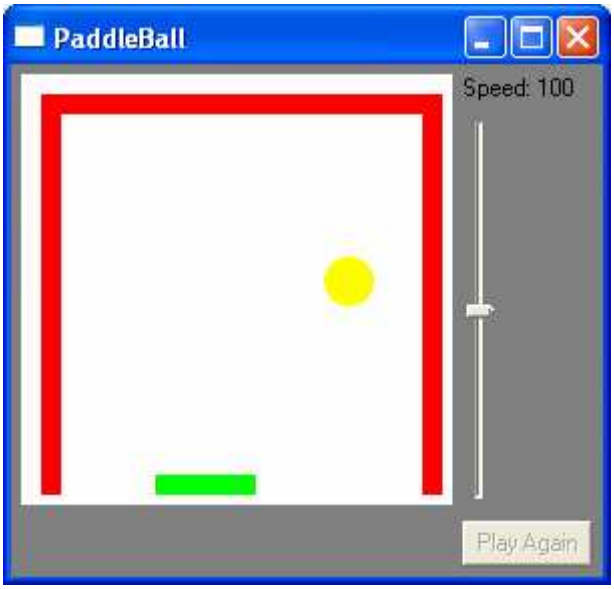

Figure 1. Paddleball in wxFruit.

3 Future Work

As the Paddleball example demonstrates at least in part, the work put into wxFruit so far has produced a functional GUI toolkit, matching and in many ways superceding the original Fruit in simplicity of the implementation, portability, and extensibility. There remain some problem areas in wxFruit,

[8] The source code to this version of Paddleball, as well as the complete source code to the wxFruit toolkit, is available on the website accompanying this paper.

however. Further development on wxFruit, and the ultimate fate of the library, hinges greatly on the successful resolution of these issues.

3.1 Switching

The various switching combinators in Yampa—`switch`, `rSwitch`, and so forth—pose a problem when one attempts to apply them to wxFruit's widgets. In the first place, the switching combinators are designed to operate on pure signal functions of the type `SF`, not the new `WXBox` type we have introduced. In order to switch a `WXBox` (that is, change the contents of some area within a GUI), we would need to first "unlift" the `WXBox` back into a `Widget`. One way of doing this would be to define a new set of switching operators that act on `WXBox`es instead of signal functions. In addition to switching the widgets underlying the switched boxes, these switchers would need to route the auxiliary input and output signals of the widgets correctly.

Simply making the types and signals work out would not solve the issue, though. The lifetimes of widgets in the wxFruit system are delicately managed, through the two-stage creation process we have mentioned, and in (experimental) system-level code which destroys the widget objects corresponding to widgets in the contents tree when parts of the tree disappear. This lifetime-management scheme is rather fragile, and it is not clear yet whether it can support dynamic changes in the makeup of a GUI, which switching would allow.

Note, however, that for many applications the lack of ability to switch widgets is inconsequential, because the makeup of a GUI within a single window tends to be static rather than dynamic. In other cases this limitation may be worked around, which the wxFruit version of Paddleball illustrates. In the Paddleball application, a switching operation occurs when a game ends and the game portion of the screen is replaced with one displaying "Game Over," and another switch occurs when the "Play Again" button is pressed and the game-over screen is replaced with a reinitialized game. These are genuine switches using the `drSwitch` operator; the reason this works is because the game and the game-over screens are represented by simple signal functions which output a `WXPicture` signal. This signal is then routed into a single, static `wxpicture` widget, which displays the picture on the screen. It is the signal functions that are switched, not the widget.

3.2 Extensibility

Currently, wxFruit only supports four widget-types: simple buttons, static text labels, slider controls, and the amorphous `wxpicture` widget. While adding new widgets corresponding to all the GUI components offered by wxHaskell is a straightforward procedure, and much easier and quicker than adding new widgets to the original Fruit, it is still a rather heavyweight

process. New response and request types for each new widget must be added to the response and request data types; a large number of these would make the data types lengthy and unwieldy. Adding some form of run-time typed object representing response and request events would perhaps yield a more modular solution. In addition, there is a significant amount of parallel code in the definitions of widgets; this suggests that a process to automate and streamline the process of adding new widgets to wxFruit's widget set may be possible.

The future of wxFruit does not rest entirely on the addition of new widgets, though. Accommodating diverse graphical components such as menus, the dynamic creation of new windows, interfacing with various host platform resources, and other elements of modern GUI-based applications may require further restructuring of the toolkit. Layering wxFruit upon wxHaskell, with its wide range of resources, should prove to be an advantage when adding these kinds of features.

4 Conclusion

As the discussion above reflects, there is still much work to be done on wxFruit if we hope to transform it into more than a research prototype of a GUI toolkit. I am confident that such work will provide more interesting insights into the ways that the ideas of functional reactive programming can be combined with practical software aimed at a nonacademic programming audience. While the work done so far on wxFruit has not yet met the goal of completely merging and thereby subsuming both the wxHaskell and Fruit libraries, it has shown that such a merging is possible. It is one step further in the ongoing process of bringing the expressiveness and clarity of the FRP approach to the wider software-engineering world.

References

[1] Antony Courtney. Functionally modeled user interfaces. In *Proceedings of the Tenth Workshop on Design, Specification, and Verification of Interactive Systems*, June 2003.

[2] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *Proceedings of the Haskell Workshop*, September 2001.

[3] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the Haskell Workshop*, September 2003.

[4] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, 2003.

[5] Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, New York, 2000.

[6] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[7] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the Haskell Workshop*, September 2002.

[8] Ross Paterson. A new notation for arrows. In *ICFP'01: International Conference on Functional Programming*, pages 229–240, Firenze, Italy, 2001.