Haskellで GUIを扱うためのライブラリ Phooey の改良

眞々田 泰裕

本論文の目的は関数型言語の特徴を活かしたプログラミングによる GUI の作成法を開発することである．関数型言語で GUI を扱うためのツールキットはその内部では大抵 C++などのオブジェクト指向な命令型言語に基づいてできている．そのため関数型言語を用いた GUI プログラムは命令的な面影を残し，関数型の力を発揮しきれていない．Haskell の場合は GUI ライブラリのひとつである wxHaskell をより関数型らしく扱えるようにするために Phooey と呼ばれる Haskell のライブラリが開発されたが，こちらはまだ実用的な段階まで開発が進んでいない．

本論文ではこの Phooey に更なる改良を加えて，より実用的に扱えるように実装を行う．そして新しい GUI ツールキットを用いた GUI の実装の手順を紹介する．

The goal of this paper is to develop the GUI library for the functional programming language Haskell. There are several GUI toolkits for functional programming languages but they are based on the object-oriented programming languages such as C++ and Java. Therefore functional GUI programs often take form of imperative codes, and cannot use their functional competence fully. In Haskell, the GUI library called Phooey was developed based on the GUI library wxHaskell to make use of the features of the functional language, but the development of Phooey has been stopped before it reaches to the stage of practical use.

In this paper, we improve Phooey and implement the new GUI library called Neooey. We also show how to make GUI codes by Neooey.

1 はじめに

今日，だれもがコンピュータを操作する時代となり，コンピュータとユーザとのギャップを視覚的な観点から埋める GUI の重要性はより一層増加している．その GUI の実装を行うプログラミングの手法で特に関係が深いものとしては次の 2 つが挙げられる。イベントドリブン（イベント駆動）型プログラミングとオブジェクト指向プログラミングである。イベントドリブンプログラミングとはプログラムがユーザによる入力やシステムによってプログラム外から発生させたものであるイベントが発生するまでイベントループで待機を行い，イベントの感知に伴ってその処理を行う形式のプログラミングである。ユーザとのやりとりで副作用が多く混入するためにコードが複雑化し易く，大規模な GUI となるとコードの生成やバグの発見が難しい．また，オブジェクト指向プログラミングはこの問題を解消させるために生み出された手法でクラスやその継承を利用して類似の処理を一括りにして GUI のコードやその把捉をより容易にする．こういった理由もあり GUI プログラミングで使われる言語はオブジェクト指向を盛り込んだ C++や Java 等の命令型言語が使われることが一般的である．

GUI は副作用が付きまとうため，関数型言語におけるプログラミングは主流ではなかった．しかし副作用と純粋な作用とを分離して考えるという特徴はコードを単純化し，より安全な GUI の作成やバグの発見を容易にする．そのため近年では $C\#$ にラムダ式が導入されるなど関数型の考え方に注目が集まっている．

関数型言語で GUI を作成するためには各言語に用意されている GUI ライブラリを利用する．そのライ

The Improvement of Phooey: libraies to use GUI in Haskell

Yasuhiro Mamada, 千葉大学大学院理学研究科，Graduate School of Science, Chiba University.

ブラリは，基本的にその内部では $C++$ などの命令型言語のバインディングによってできているものが多い．そのため関数型言語を用いた GUI プログラムであってもそのコーディングは関数型言語のスタイルによるものになりにくい．

関数型言語の 1 つである Haskell では，よく使用される GUI ライブラリとして wxHaskell と Gtk2hs の 2 つが挙げられる．wxHaskell については C++で記述されている GUI ツールキット，wxWidgets のバインディングとなっており，その wxHaskell をより関数型らしく扱えるようにするために C.Elliot によって Phooey と呼ばれる Haskell のライブラリが開発された．Phooey は関数型言語を用いた GUI プログラミングの 1 手法である FRP の考え方も盛り込み，Haskell らしいスタイルでのコーディングが可能となったが実用的な段階まで開発されてはなく，汎用性が低い．そこで本論文ではこの Phooey に更なる改良を加えて，より実用的に扱えるように実装を行う．

論文の構成については次の通りである．第 2 章で関数型言語における GUI プログラミングの手法である FRP について説明する．第 3 章で本論文で改良を加える GUI ライブラリである Phooey 及び，その背景にある GUI ツールキット及び GUI ライブラリについて説明する．第 4 章で実際に Phooey の改良を行い，GUI の作成法を例を挙げて説明する．第 5 章で結論を述べる．

2 FRP

FRP とは functional reactive programming の略称である．reactive programming である RP を関数型で表現したもののことである．

RP とは GUI などの副作用を伴うコードで使われている表現の 1 つである．一般的な GUI のプログラムは行わせたい操作や命令，イベントの設定を実行させたい順番に書き並べる．これはプログラマー視点では直感的でわかりやすいかもしれないがイベントや変数の数が増えてくると依存関係を捉えることが困難になる．それに対して RP は値を時間によって変化するものととらえ，この値との関係を利用してプログラミングを行う．この関係性のことをデータフローという．ある値に変化があるとその値との関係を通して別の値も再計算される．一番浸透している例を挙げると Microsoft Excel などの表計算ソフトにその考え方が使われている．値がこの時間に対して連続的に変化する値のことをリアクティブ (reactive) な値と呼び，リアクティブな値の関係性をふるまい (behavior) とよぶ．また，時間に対して断続的に与えられる値をイベント (event) と呼ぶ．一般に RP のコードは長さが短くなるという特徴があり，それゆえにバグが少なく保守性が高くなる．

FRP は C.Elliott, P.Hudak による [1] が原形になっているといわれる．イベントやリアクティブな値についてしばしばストリーム (stream) と呼ばれるものを考える．ストリームは時間ごとの値のリストとして表現される．例えば時間と値の連想リストとして表現される．

```
type Stream a = [(Time, Value a)]
```

リアクティブな値とイベントは連続的なものと離散的なもので違いはあるが，リアクティブな値は変化した時刻とその値のストリーム，イベントは発生した時刻とその値のストリームとして表現することができ，互いに類似している側面がある．C.Elliott による論文 [2] ではその点を踏まえ，イベントとリアクティブな値を相互再帰的な定義で実装している．

```
type Event a = Future (Reactive a)
type Reactive a = a 'Stepper' Event a
```

`Stepper` はイベントに初期値 (時間 0 における値) を付与すること，`Future` は時間をイベントの次の発生時刻まで進めることを表す．

このように時間によって変化する値を逐次ふるまい関数で動かすことによって FRP が実装できる．本論文において改良を行う Phooey においてもこの考え方に基づいて作られている．

また，本論文では深く取り上げないがアローの一種であるシグナル関数を利用して FRP を表現した AFRP と呼ばれるものも存在する．シグナル関数とはふるまいのようなものでリアクティブな値を変換する関数のことを指す．AFRP では複数のシグナル

関数を結合して大きなシグナル関数をつくることで FRP を表現する．[1], [8], [5]

3 Haskell における GUI ライブラリ

この章では Phooey 及びその元となっている GUI ツールキット及び GUI ライブラリについて述べる．

3.1 wxWidgets

wxWidgets は C++で記述されている GUI ツールキットである．1992 年にクロスプラットフォームに動くことを目的としてエディンバラ大学で J.Smart によって開発が開始された当時は wxWindows という名前だったが，マイクロソフト社の Windows と名称が被ることを受け 2004 年に今の名前に変更された．

1 つの特徴としてはクロスプラットフォームに扱えることで，Windows，Mac OS X，Linux といった多くの OS で動かすことが出来る． C++ であることから高速であり，wxWidgets を利用するにあたって他のソフトを用意する必要がない．また，他のプログラミング言語からでも使用可能にするために各種バインディングが用意されている．例えば Python で扱うための wxPython，Perl で扱うための wxPerl， C# で扱うための wx.NET がある．それ以外にも php, java, lua, lisp, erlang, eiffel, BASIC, ruby, Javascript といったかなり多くのバインディングが用意されている．別の特徴としてそれぞれのプラットフォームのソフトウェア開発キットを用いてルックアンドフィールを保つ．つまり Windows などの各プラットフォームでコンパイルされたプログラムは視覚的な面や使い勝手がそのプラットフォームに合わせられるというものである．体裁にこだわる反面，プラットフォームごとに挙動が異なる場合がある．また，かなり昔から開発が続いているので，ユーティリティやドキュメントが充実している．現在も開発は行われており，最新のバージョンは 2014 年 10 月 6 日にリリースされた wxWidgets 3.0.2 である．こちらは公式サイトから無償で入手することができる．[7] Phooey では Haskell で扱えるようにするために wxHaskell というバインディングを使用している．

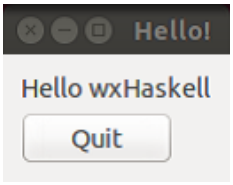

図 1 wxhaskell による GUI

3.2 wxHaskell

wxHaskell は Haskell で GUI を扱うためのツールのひとつである．[4] 内部としては上述した wxWidgets をバインディングした Haskell のライブラリである．類似のライブラリとしては他に GUI ツールキット Gtk+ をバインディングしたものである，Gtk2hs というツールが存在する．wxHaskell と Gtk2hs は Haskell で GUI を扱うツールとしては双璧となっている．

wxHaskell のバージョンは大きく分けて 0.13 系列と 0.90 系列がある．最新版は 2014 年の 8 月 12 日にリリースされたものでバージョンは 0.91.0 になっている．こちらは公式ページから無償で入手することができる．[3]

wxHaskell の概要については例を用いて述べる．図 1 は wxHaskell による単純な GUI の例である．この GUI はボタンが押されるとウィンドウを閉じるだけのものである．この GUI のソースコードは次のように表される．

```
module Main where
import Graphics.UI.WX

main :: IO ()
main = start hello

hello :: IO ()
hello = do
     f    <- frame    [text := "Hello!"]
     p    <- panel f  []
     quit <- button p [text := "Quit",
                       on command := close f]
     set p [layout := margin 10 $
            column 5 [label "Hello wxHaskell"
                           , widget quit]]
     set f [layout := widget p]
```

wxHaskell では `IO ()` 型で表される GUI 関数を記述し，それを `start` 関数で呼び出すことによって

GUI が起動する．GUI の定義の仕方の概略は，以下のとおりである：
まずフレームを作成し，フレームの中にパネルを作成する．そしてパネルの中にボタンやテキストエントリなどのコントロールを作成する．その後でウィジェットの大きさや配置を示すレイアウトやイベントが発生した際の処理は通常その後で定義をする．という順番で行われる．

GUI にはウィジェットという最小単位が存在する．これは GUI オブジェクトのようなものであり，上で名前の挙がったフレーム，パネル，ボタン，テキストエントリなどのコントロールは全てウィジェットの一種である．GUI の構成はこれらウィジェットをつなぎ合わせることによって行われる．

フレームとはメインウィンドウ，すなわちウィンドウ全体の事を指す．基本的に他のすべてのウィジェットはフレームの中に属する．`frame` 関数を用いることによって作成することができてウィジェットとしての型は `Frame ()` である．

パネルはウィジェットコンテナの 1 つで，中にウィジェットを入れることのできるウィジェットである．後に説明するレイアウトを整える目的で使われる．一般的な GUI ではフレームの次にパネルを設置し，他のウィジェットをその中に設置するといった使い方が多くみられる．`panel` 関数を用いることによって作成することができてウィジェットとしての型は `Panel ()` である．

コントロールはコントロールウィジェットとも呼び，ユーザやシステムによる入力を感知することのできるウィジェットである．例としてボタン，テキストエントリ（入力欄，テキストコントロールとも），スライダー（つまみを動かして程度を指定できるコントロール），プロパティグリッド（スプレッドシートのようなもの）などが挙げられる．コントロールはユーザによる選択や操作を感知するとイベントと呼ばれるサインを発生させる．このイベントの種類と発生した時に行いたい処理を結びつけることができて，これによってさまざまなアクションを引き起こすことが可能となる．コントロールウィジェットには多数の種類があるので，それぞれのウィジェットに対して作成関数や型が存在する．例えばボタンウィジェットは `button` 関数を用いることによって作成することができてウィジェットとしての型は `Button ()` である．また，テキストエントリウィジェットは `textCtrl` 関数を用いることによって作成することができてウィジェットとしての型は `TextCtrl ()` である．

ウィジェットを作成する関数は原則として引数に親となるウィジェットとプロパティリストを取る．いくつかのウィジェット作成関数について型を紹介する．`frame` を除いて第一引数として現れている `Window a` は任意のウィジェットの型が入ることを示している．`frame` の引数が少ないのはフレームに親となるウィジェットは存在しないためである．第二引数として現れている `[Prop (*** ())]` はプロパティリストと呼ばれ，ウィジェットに関する各情報を設定することが出来る．

```
frame  ::  [Prop (Frame ())] -> IO (Frame ())
panel :: Window a -> [Prop (Panel ())]
             -> IO (Panel ())
button :: Window a -> [Prop (Button ())]
              -> IO (Button ())
textCtrl :: Window a -> [Prop (TextCtrl ())]
               -> IO (TextCtrl ())
```

各情報については属性と呼ばれ，ウィジェットの種類に応じていくつかの種類の属性を定めることが出来る．必ずしもすべての属性を定める必要はなく，設定しなかった場合はデフォルトとしての値が当てはめられる．属性を定める場合は
属性名 := 値
の形で与えることができる．あるウィジェットに対して属性の設定を行うには 2 つの方法がある．一つはウィジェットを生成する関数を呼び出す際に指定してしまう方法である．ウィジェットを生成する関数は基本的にプロパティリストを引数に取るようにできているのでその際に設定してしまうのである．もう一つは `set` 関数を用いる方法である．`set` はプロパティリストの再設定を行う関数である．逆に，定められた属性の値を抽出する働きを持つ `get` と呼ばれる関数も存在する．

```
set :: w -> [Prop w] -> IO ()
get :: w -> Attr w a -> IO a
```

属性の具体的な例としてここでは text，clientSize，on command，layout を挙げて説明する．

```
text :: Textual w => Attr w String
clientSize :: Dimensions w => Attr w Size
on command :: Commanding w => Attr w (IO ())
layout :: Form w => Attr w Layout
```

`text` は名前や文字列としての値を当てはめる属性である．フレームにおいてはタイトルバーに表示される文字列を表し，ボタンにおいてはボタンに表示される文字列を表す．また，テキストエントリにおいてはエントリ自身に書かれる文字列を表す．

`clientSize`　はウィジェットの大きさを記述する属性である．後に紹介するレイアウトと働きが重複するものもあり，設定の仕方によっては想定通りに反映されない恐れがある．

`on command` はコントロールウィジェットがユーザによる入力，変更を感知した際に処理を行いたいアクションを記述する属性である．また，外的な刺激によって発生するイベントについてはいくつか種類があり，用途に応じて command の箇所を別の関数に置き換えることによって各イベントに対する反応を記述することが可能である．例としては選択されている状態を観測する `select` などがある．

`layout`　はウィジェットのレイアウトを記述する属性である．GUI においてレイアウトは視覚的なウィジェットの配置を表している．wxHaskell においてはレイアウトの型を `Layout` 型で表現する．

レイアウトを取り扱う関数にはいくつか種類がある．レイアウトを生成する関数，レイアウトを変換する関数，レイアウトを結合する関数である．

レイアウトを生み出す原始的な関数としては `label` や `space` などがこれに該当する．`label` 関数は文字列を画面に表示させるだけのレイアウトである．`text` 属性を定めたウィジェットとは違い，表示される文字列が変更されることはない．`space` 関数は引数で受け取ったサイズの矩形の分だけスペースを陣取るレイアウトである．何かが表示されるわけでもない何もない空間である．視覚的にまとまったスペースを開けたいときに有効である．

レイアウトに効果を掛ける関数については `fill` や `margin` などがこれに該当する．`fill` 関数は空いているスペースを満たすようにレイアウトを拡大して埋め尽くす関数である．また，`margin` はレイアウトの外枠を指定されたサイズだけ縁取る関数である．このように，これらの関数は引数にレイアウトを取り，それに変更を加えるという形で定義がなされている．

レイアウトを結合する関数については `row` や `column` などがこれに該当する．これらの関数はリストなどを通して複数のレイアウトを引数から取り，レイアウトを 1 つに統合する．`row` 関数は水平方向にレイアウトの結合を行う．この関数は第一引数に数値を取るようにできており，この数値によって各レイアウトの隙間を調節することが可能である．`column` 関数は `row` 関数に似ているところも多いが，垂直方向にレイアウトの結合を行う関数となっている．

また，特筆すべきレイアウト用の関数に `widget` がある．この関数は子ウィジェットで定められたレイアウトを抽出する．

```
label :: String -> Layout
space :: Int -> Int -> Layout

fill :: Layout -> Layout
margin :: Int -> Layout -> Layout

row :: Int -> [Layout] -> Layout
column :: Int -> [Layout] -> Layout

widget :: Widget w => w -> Layout
```

以上が wxHaskell の概要である．wxHaskell は Haskell で GUI を作るためには有効ではあるが C++(または C) で構成されている wxWidgets をそのまま翻訳したかのように作られており，GUI 関数の定義は命令型言語で行うような逐次実行のコーディングになっている．

これを改善する手法として FRP を GUI に織り交ぜて Haskell のスタイルで GUI の作成が行える Phooey という wxhaskell 上のライブラリが開発された．これについては次の節に述べる．

図 2　Phooey によるショッピングリストの GUI1

3.3　Phooey

Phooey は wxHaskell 上に実装されたシンプルな Haskell の関数型ライブラリである．これは FRP に関する論文 [1] [2] で有名な C.Elliott によって開発されたものである．外面的には関数型らしい記述を追究しており，wxHaskell で見られるような命令の逐次実行的ではない，Haskell らしいコードを組むことができる．実際に Monad や Applicative といったなじみのあるものを用いて GUI を実装することができる．構造的には FRP の考え方を用いて作られており，短いコード量によって表現することが可能となっている．phooey は単語自体が持つ自虐的な意味と，FUI(Functional User Interfaces) の頭文字を発音から文字った Phunctional ooser ynterfaces の略との 2 つの意味を掛け合わせたものである．最終更新は 2011 年 9 月 10 日 のバージョン 2.0.0.1 である．

まずは Phooey の構造や特徴を GUI の例を通して説明する．

図 2 はフルーツのショッピングリストを表現した GUI である．リンゴとバナナに対してそれぞれスライダーと呼ばれるつまみが存在し，下には合計を表すテキストエントリが置かれている．スライダーを動かすことによってフルーツの合計が再計算され，テキストエントリに表示される数が更新される．

このショッピングリストを表現する GUI を Neooey を用いて表現すると次のようになる．

```
sl0 :: Int -> UI (Source Int)
sl0 = islider (0, 10)

apples, bananas :: UI (Source Int)
apples  = title "apples"  $ sl0 3
bananas = title "bananas" $ sl0 7

total :: (Show a, Num a) => Source a -> UI ()
total = title "total" . showDisplay

ui1x :: UI ()
ui1x = title "Shopping List" $
        do a <- apples
           b <- bananas
           total (liftA2 (+) a b)
```

Phooey では GUI を表現するために `UI` 型を使う．構造としてはメインパネルを参照してからアクションを起こすものになっている．`UI` 型はその内部に FRP を表現するためのリアクティブなアクションとレイアウトを情報として保有しており，UI の結合が行われる際に，レイアウトの結合やリアクティブな値のも行われる．結合によって構築された UI は `runUI` という関数を用いることによって起動することができる．

```
runUI :: UI () -> IO ()
```

Phooey の特徴としては Haskell のスタイルによるコーディングを実現しているということが挙げられる．例えば上記の例における `ui1x` では `do` 記法を用いて定義されているが，`UI` 型が `Monad` や `Applicative` といった型クラスのインスタンスになっていることから関数 `liftM2` や `(>>=)` を用いることが可能である．これらを用いることによって次で紹介しているように自然で短い Haskell スタイルによる定義が実現される．

```
(.+.) :: Num a => UIS a -> UIS a -> UIS a
(.+.) = liftA2 (liftA2 (+))

fruit :: UI (Source Int)
fruit = apples .+. bananas

ui1y :: UI ()
ui1y = title "Shopping List" $ fruit >>= total
```

また，`UI` 型の結合を行う際にレイアウトを指定す

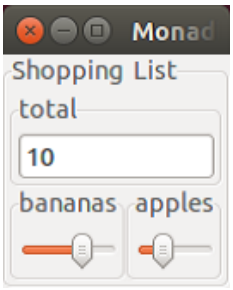

図 3　Phooey によるショッピングリストの GUI

る関数を適用することができる．これを使用することでウィジェットの配置を変えることが可能である．

次のソースコードを動かすと図 3 のようになる．この例では `fromRight` 関数によってリンゴとバナナのウィジェットを右から配置し，`fromBottom` 関数によってフルーツと合計のウィジェットを下から配置している．

```
ui2 = fromBottom $
      title "Shopping  List" $
      fromRight fruit >>= total
```

このように Phooey を用いることで GUI のプログラミングを Haskell のスタイルによって行うことが可能になった．しかしこのライブラリには問題点もある．まずは提供されているライブラリは実用的な段階までは開発がなされていないことである．例えばウィジェットのプロパティはある程度固定されており，ユーザによって設定することのできる属性はほとんどない．ボタンウィジェットを例に挙げるとボタンに書かれる文字列については設定することが可能だがボタン自体のサイズや，ボタンを押すことが許されているかの可否などの詳細な情報についてはユーザーが自由に設定することができない．この制限は簡潔で容易にコードが書けるという点はで優れているが汎用性を欠くものとなる．

また，Phooey には FRP の考え方が用いられている．そしてそれを実現する方法との 1 つとして値や関数を投げる UI とそれを受けて反応を返す UI を結合し，全体としてひとつの UI にまとめあげる構造をとる．しかし結合の際にこれら 2 つの UI の役割は固定されてしまう．言い換えると値を生成する UI として別の UI と結合した場合，その結合した相手から情報を受け取って処理するといったことができない．これは値の伝達が一方通行であれば問題ないがもし相互的で複雑に値を交換する UI を考えた場合，これを有効に表現することができない．

さらに，FRP を表現するために Phooey においては `UI` 型はレイアウトやソースアクションを保有するという性質があったが，それゆえにウィジェット情報については保有しない．つまり一度作成されたウィジェットに対して後になってプロパティを設定し直すことができない．このことにより例えばボタンが押された時のイベント処理は作成時に固定されてしまい，後になってその処理自体を変更するとなった場合に無理がでてしまう．

4　Neooey

この章では Phooey で発生した問題，特に汎用的な定義を行うことができない問題と UI 同士の値のやりとりが限定的なものに限られてしまう問題，UI の情報を後になって変更することができない問題を解決することを意図して Neooey と命名した新しい GUI ライブラリを作成する．

Phooey 自体も FUI(Functional User Interface) の発音のもじりであるが，そこに new という文字を付け加えて New Phooey という意味合いで名前を付けた．根底にあるコンセプトは Haskell のスタイルによる GUI の設計を汎用的に行うことを可能とすることである．

4.1　UI

Phooey の場合，レイアウトの情報はそれぞれの UI の値の中で保持されている．そして UI を結合する際にレイアウトについても UI の内部で結合される形をとる．Neooey におけるレイアウトも Phooey と同様に UI にレイアウトの情報を保有させる形式をとる．また，Phooey においてはメインパネルを参照して動く関数として UI を定義する．Neooey においてもこの考え方を踏襲し，メインフレームとメインパネルを参照して動く関数として UI を定義する．Phooey と違うのはメインパネルの他にメインフレームも参照

するようにした所である．こうすることによってメインフレームに対して働き掛けたい操作を UI に行わせることが可能となる．例えばメインフレームに直接追加させる必要のあるメニューバーやステータスバーが設置可能になる．

上記の働きをもつ UI 型の表現は感覚的にはメインフレームとメインパネルを引数としてとり，結果とレイアウトの情報を返すものとして捉えることができる．つまり次のように考えることができる．

```
type UI a =
  MWin -> Win -> IO (a,  Layout)
```

しかし実際の UI 型の実装は引数に取って実行する形式ではなく，モナド変換子を用いて行う．

モナド変換子は核となるモナドに対して働くことができる．これを用いることでモナドに性質の付与を行い，複数の働きを持つひとつの大きなモナドを構築することが可能になる．Neooey では IO アクションを核のモナドとしてこれにモナド変換子を適用させて UI を示す型である UI 型を構築する．

今回使用する変換子は Writer モナドを変換子化させた WriterT と Reader モナドを変換子化させた ReaderT である．WriterT は核である IO アクションにレイアウト情報を保持させる目的で使用する．また ReaderT は核である IO アクションが外から与えられるメインフレームとメインパネルの情報を参照して動くことを目的として使用する．

```
type UI =
  ReaderT MWin (ReaderT Win
                        (WriterT Layout IO))
```

MWin 型は wxHaskell における Frame () 型，Win 型は wxHaskell における Panel () 型のシノニムである．レイアウト情報を示す Layout 型は wxHaskell で使用されているものと同じである．

また，Phooey における UI が返す結果と Neooey における UI が返す結果はその役割が異なっている．Phooey では必要に応じて UI が持つリアクティブな値を結果として返す．これによって Phooey では FRP を利用した UI の定義が可能になっている．Phooey ではウィジェットを保有する機構は持っていない．そうなるとウィジェットを一度作成してしまうとそのウィジェットの設定が行えなくなる．これはウィジェットの依存関係がより複雑になるとその表現が難しくなることを意味する．

一方，Neooey においては UI の中で使用されているウィジェット情報を結果として返す．こうすることによってウィジェットの設定の変更を後でも行えるようにする．例えばボタンウィジェットである Button () 型を保有する UI は UI (Button ()) 型となる．

Neooey において UI に対して使うことのできる関数は 3 種類に大別することができる．ひとつはウィジェット情報を保有する UI を作る関数，もうひとつは UI にレイアウト情報を付与するための関数，そして最後は複数の UI を結合するための関数である．

ウィジェット情報を保有する UI を作る関数とはボタンウィジェットやテキストエントリウィジェットなどのコントロールウィジェットを作成してそのウィジェット情報を結果として返す関数である．この関数は引数として wxHaskell と同じ記法のプロパティやそれ以外の最低限必要となる基本情報を受け取ることとなる．内部では wxHaskell のコントロールを作る関数を用いており，関数名も参照元の関数を元に付けられている．

```
textCtrlUI
     :: [Prop (TextCtrl ())] -> UI (TextCtrl ())
buttonUI
     :: [Prop (Button ())] -> UI (Button ())
```

UI にレイアウト情報を付与するための関数として一般的なものは，すでに存在する UI を引数にとってその UI が持つレイアウトの変更を行うものである．

```
marginUI :: Int -> UI a -> UI a
fillUI :: UI a -> UI a
```

これらの関数では引数でとられる UI がもつウィジェット情報についてはそのまま引き継がれる．

また，ウィジェット情報を持たないレイアウトを作る関数も存在する．何も置かれていない空間やラベルなど，他のレイアウトを参照しないような自立したレイアウトを作ることができる．こういったレイアウトは一度作成されてしまうと変更は行われない．

```
labelUI :: String -> UI ()
spaceUI :: Int -> Int -> UI ()
```

そして最後は複数の UI を結合するための関数である．UI の結合が行われた際にウィジェットやレイアウトの結合が行われる．レイアウトの結合については垂直方向に結合するか，水平方向に結合するかによって 2 種類の演算子が存在する．

```
(<->) :: UI a -> UI b -> UI (a, b)
(<|>) :: UI a -> UI b -> UI (a, b)
```

(<->) が水平方向，(<|>) が垂直方向にレイアウトを結合する演算子である．ウィジェット情報については引数でとられたそれぞれの UI が保有する情報を組にして保有する形式をとる．例えば型が UI (Button ()) の UI と型が UI (TextCtrl ()) の UI を結合してできた UI の型は UI (Button (), TextCtrl ()) となり，そのウィジェットは Button () 及び TextCtrl () のウィジェット情報を保有することになる．

場合によっては結果として持っておく必要のないウィジェット情報もある．そういった時の為に takeLeft 関数や takeRight 関数を使うことができて，これを用いると結果として持っているウィジェット情報を削ることができる．

```
takeLeftUI :: UI (a, b) -> UI a
takeRightUI :: UI (a, b) -> UI b
```

4.2 ウィジェットアクション

UI の作成や結合によって得られる GUI はウィジェット情報を構造的に保持し，それと同時にレイアウトも表現する．しかし，ボタンが押された際の挙動などのウィジェットの反応についてはこれだけでは行われない．そこで，ここでは UI とはまた別にウィジェットアクションと呼ばれるものを作成する．ウィジェットアクションはウィジェット情報を参照して動くアクションである．最終的にこのウィジェットアクションを UI と結びつけることによって反応を示す GUI が構築される．

Phooey ではこういったアクションは UI 作成時にほぼ定義され，リアクティブな値の受け渡しに関するアクションだけ UI が情報として保持する．

しかし Neooey ではウィジェットの反応を表現するウィジェットアクションを UI から分離した．こうすることによってより複雑なウィジェットの依存関係を持つアクションの指定が行えるようになる．

ウィジェットアクションは Neooey における UI 型と同様にメインフレームとメインパネルを参照する．また，それとは別に UI 型が結果として返すウィジェット情報についても参照する．UI 同様ウィジェットアクションの記述のためにモナド変換子を使用する．

```
type WidAct' w = ReaderT MWin (ReaderT Win
                                 (ReaderT w IO))
type WidAct w = WidAct' w ()
```

ここで定義された WidAct 型は UI 同様に各ウィジェットを引数で取って動くアクションとして考えることもできる．

```
type WidAct' w a = MWin -> Win -> w -> IO a
type WidAct w = WidAct' w ()
```

一番基本的なウィジェットアクションは属性やプロパティなどの引数を受け取ってウィジェットアクションを作る関数 getF 及び setF である．

```
getF :: Attr attr val -> WidAct' attr val
setF :: [Prop a] -> WidAct a
```

これらは wxHaskell における get 及び set 関数を元に作られている．ほとんどのウィジェットアクションを作る関数はこれらを利用して作られている．例えばボタンが押された際の挙動などを示すコントロールの反応を記述する関数が挙げられる．

```
setWidAct :: Commanding a =>
             WidAct a -> WidAct a
setWidAct2 :: Commanding a =>
              WidAct b -> b -> WidAct a
```

ウィジェットアクションはモナドであるので do 記法や (>>=) などの結合演算子などを利用して合成する．

このとき，ウィジェットアクションが参照する型に相違があるとそのまま合成することができない．その

ため，これを解決させるために要求する型を変形して形をそろえる必要がある．これを行う関数が right 及び left 関数である．

```
right :: WidAct' b c -> WidAct' (a, b) c
left :: WidAct' a c -> WidAct' (a, b) c
```

これらの関数はウィジェットアクションを引数としてとり，そのウィジェットアクションが参照するウィジェット情報を変える働きがある．具体的には引数のウィジェットアクションが要求するウィジェット情報を組の形にする．right 関数はウィジェットの組を作り，その中で本当に使用したいウィジェットは組の右におかれることを示し，left 関数はそれが組の左におかれることを示している．

例えば，以下の例において 2 つのウィジェットアクション waA と waB があった場合型が一致しないためにこのままでは結合をすることができない．しかし left 関数及び right 関数を用いてウィジェットアクションの型をそろえることによって一度にまとめることができるようになる．

また，効率よくウィジェットアクションを作成するための関数として widgetSPA 及び widgetSPAC が挙げられる．これらの関数は第一引数にくる take 関数で抜き出されたウィジェットがトリガーとなって第二引数で与えられたアクションを引き起こす関数となっている．

```
widgetSPA :: Commanding w' =>
     (w->w')      -> WidAct w    -> WidAct w
widgetSPAC :: Commanding w' =>
     (w1->w') -> WidAct w2 -> WidAct (w1, w2)
```

特に下の widgetSPAC は入力側のウィジェットと出力側のウィジェットが分離されている場合に用いることができて，入力側と出力側をつなげたウィジェットアクションを構築する．

ウィジェットアクションを UI に結び付けるには putWAinUI 関数を用いる．この関数はウィジェットアクション及び UI を引数にとり，ウィジェットアクションが要求する挙動を UI に付与させることが可能になる．

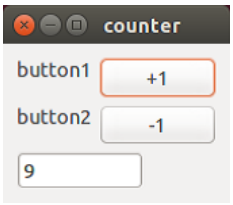

図 4　Neooey によるカウンターを行う GUI の例

```
putWAinUI :: WidAct' a b -> UI a -> UI a
```

4.3　例：カウンター

本節では実際に Neooey を使用して GUI を作成する例を見せる．そしてその過程を通して効率的な作成手順について説明する．

作成する GUI は図 4 のように 2 つのボタンがついたカウンターである．「+1」と書かれたボタンを押すことによってテキストエントリに表示されている数値が 1 増加し，「−1」と書かれたボタンを押すことによってテキストエントリに表示されている数値が 1 減少する．

この GUI を Neooey を用いて作成するには次の 3 つの手順に分けて行うのがよい．

UI と型の作成
→ take 関数の作成
→ ウィジェットアクションの作成

まず始めに行う工程は UI 及びウィジェットごとの型の作成である．ウィジェットごとの型とは例えば入力や出力となるウィジェットについて別名をつけるということである．単一のウィジェットのみでなく，ウィジェットを組にしたものについても名前をつけておくとよい．

```
type Input = Button ()
type Input2 = (Input, Input)
type Output = TextCtrl ()
type Count = (Input2, Output)
```

ウィジェットごとの型に名前をつける理由はウィジェットの数が多くなった場合に煩雑でわかりにく

くなってしまうのを避けるためである．もし途中で型に名前を付けない場合，今回の例ではメインの GUI 関数の型は UI ((Button (), Button ()), TextCtrl ()) となる．型の命名を先に紹介したが，実際は UI の生成と同時に型の命名を行うとよい．

UI の構築ついては基本的に UI を生成する関数と UI を結合する関数の組み合わせによって定義する．

UI の生成については各ウィジェットごとに専用の関数を用いて行う．今回の例の場合作成する必要があるのは値の増減を行うためのボタン 2 つ，各ボタンの隣にあるラベル 2 つ，出力用のエントリ 1 つの計 5 つである．

```
outputUI :: UI Output
outputUI = entryUI [text := "0"]

label1, label2 :: UI ()
label1 = labelUI "button1"
label2 = labelUI "button2"

button1, button2 :: UI Input
button1 = buttonUI [text := "+1"]
button2 = buttonUI [text := "-1"]
```

UI の生成を行ったら，次は垂直方向，又は水平方向に結合を行い UI をまとめ上げる．このとき 2 つのラベル用 UI label1, label2 についてはウィジェット情報を持たないので結合の際に takeRightUI 関数を用いて保持するウィジェットの情報を剪定する．これをすることによって UI の型が UI ((),Input) から UI Input になる．尚, margin10 関数はレイアウトとして UI の縁を空けるために使用されている．

```
preUI :: UI Count
preUI = margin10 $ buttonset1 <|> buttonset2 <|> outputUI

buttonset1, buttonset2 :: UI Input
buttonset1 = takeRightUI $ label1 <-> button1
buttonset2 = takeRightUI $ label2 <-> button2
```

ここで作成された preUI 関数はボタンが押された際の処理を組み込んでいないためにこのまま動かしてもボタンに反応がない．ボタンの処理を記述するためにはウィジェットアクションを用意して盛り込む必要がある．そこでウィジェットアクションの定義を行うわけだが，その前の準備として take 関数を用意しておくとよい．

ここでいう take 関数とは組で表現されたウィジェット構造から特定のウィジェットの情報を抜き出す関数のことを指す．これを定義することによって目的のウィジェットを抜き出しやすくする効果がある．

今回の例では，がカウントアップを行うボタンウィジェット及びカウントダウンを行うボタンウィジェットを抜き出す関数を準備する．具体的には，入力側のウィジェット Input2 から fst，snd を用いて目的のウィジェットを選ぶ．

```
takeB1, takeB2 :: Input2 -> Input
takeB1 = fst
takeB2 = snd
```

take 関数の定義を行ったら最後に行うのはウィジェットアクションの定義である．updateWA’ 関数は引数の演算子を出力先のウィジェットに適用させる．actB1 は
takeB1 が示すカウントアップボタンウィジェットがイベントを感知した際に出力先の数値に 1 加算することを表している．また，actB2 は
takeB2 が示すカウントダウンボタンウィジェットがイベントを感知した際に出力先の数値に 1 減算することを表している．入力側の UI と出力側の UI が分離されている場合はこのように widgetSPAC 関数が使うことが可能となる．

```
--widget actions

actB, actB1, actB2 :: WidAct Count
actB = actB1 >> actB2

actB1 = widgetSPAC takeB1 (updateWA’ (+1))
actB2 = widgetSPAC takeB2 (updateWA’ (+(-1)))
```

最後にウィジェット構造やレイアウト情報を保持している UI にウィジェットに行わせる操作を担うウィジェットアクションを結合すれば完成である．これによって目的のカウンタープログラムが動く．

```
mainUI :: UI Count
mainUI = putWAinUI actB preUI
```

このカウンタープログラムは Phooey を用いて表現することも可能である．しかし Neooey では Phooey で表すことのできない例についても表現することが出

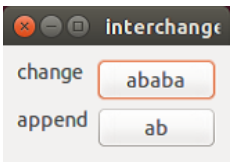

図 5　Neooey によるウィジェット同士で情報交換をする GUI の例

来る．次節では Phooey で表すことのできない GUI の実装について述べる．尚，カウンタープログラムの全ソースコードについては付録 A に載せる．

4.4　Phooey で表現できない GUI を Neooey で実装する

3.3 節で述べたように Phooey は FRP を利用して作られているが，その表現力については限定的であり，Phooey のみの力によって GUI を作ることができない例も存在する．

本節では Phooey で表現できない例のうち，UI が相互的に情報を交換させる例とコントロールの処理を途中で変更させる例について Neooey を用いた実装を行う．

まずは UI が相互的に情報を交換する例について考える．

図 5 は 2 つのボタンウィジェットが互いに情報を交換させる例になっている．上のボタンを押すと双方のボタンに書かれた文字列が入れ替わり，下のボタンを押すと双方のボタンに書かれた文字列を連結したものが下のボタンに反映される．どちらのボタンが押された場合でも双方のボタンの情報について参照及び更新を行うようになっている．すなわちどちらのウィジェットも入力及び出力を行うものとなっており，この GUI は Phooey によって表現することができないものある．

これからこの GUI を Neooey で表現できることを実際に実装することによって示す．

Neooey で行う 3 つの工程のうち，UI 及び型の作成についてはカウンターの例と同様にして行うことができるので省略する．ただし `take` 関数の実装を説明するために，ここでは命名した型のみ紹介する．

```
type InOut = Button ()
type InterChange = (InOut,  InOut)
```

カウンターの例のように入力側の UI と出力側の UI が分離されている場合は

`take` 関数を入力側の範疇で考え，定義した．そして `widgetSPAC` 関数によって入力と出力の結び付けを行った．しかし今回の例ではどちらのボタンも入出力をともに行うようになっているため，カウンターと全く同じ方法を取ることはできない．

今回の例のように入力側の UI と出力側の UI を分離することが出来ない時は，`take` 関数を入力側のウィジェット構造から考えるのではなく，入出力全体のウィジェット構造から抜き出すことを考える．

```
takeE1, takeE2 :: InterChange -> InOut
takeE1 = fst
takeE2 = snd
```

また，ウィジェットアクションについては `widgetSPAC` 関数ではなく `widgetSPA` 関数を用いて定義する．この関数は，第一引数の `take` 関数によって摘出されるコントロールに第二引数で指定されたウィジェットアクションを定めるものとして解釈できる．

それぞれのウィジェットアクションについては次のように従来の列挙による定義も行うことが可能である．その際は前述したように `left` 及び `right` 関数を用いてウィジェットアクションの型を統一させる．

```
actE, actE1, actE2 :: WidAct InterChange
actE = actE1 >> actE2

actE1 = widgetSPA takeE1 change
actE2 = widgetSPA takeE2 appendT

change :: WidAct InterChange
change = do a <- left getText
            b <- right getText
            right (setText a)
            left (setText b)

appendT :: WidAct InterChange
appendT = do a <- left getText
             right (updateWA (a++))
```

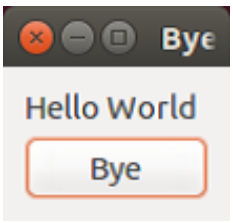

図 6　Neooey によるボタンの働きが切り替わる例 1

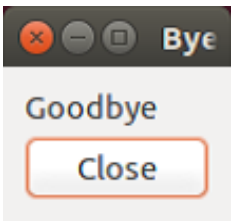

図 7　Neooey によるボタンの働きが切り替わる例 2

最後にカウンターの例と同じように putWAinUI 関数を使って UI にウィジェットアクションを埋め込むことによって UI が完成される．これによって Phooey で表現することができない GUI の例の 1 つを Neooey で書けることが示された．

尚，このプログラムの全ソースコードについては付録 B に載せる．

次はコントロールの処理を途中で変更させる例について考える．

図 6 で表される GUI はボタンが行う処理が変更される例となっている．まずボタンを押すと図 7 のように画面に表示されているテキストが「Hello World」から「Goodbye」へと変わる．そしてそれと同時にボタンに書かれている文字列も置き換わる．ここでもう一度ボタンを押すと今度はウィンドウが閉じられる．このとき，ボタンが押された際に行われる処理は一度目は文字列の表示とボタン自体の挙動の変更，二度目はウィンドウを閉じる，といったようにその働きが大きく変わるものになっている．こういった操作については Phooey を用いて表現することが出来ない例となっている．

この GUI についても Neooey で表現できることを実際に実装することによって示す．

Neooey で行う 3 つの工程のうち，UI 及び型の作成についてはこれまでの例と同様にして行うことができるので省略する．ただし take 関数の実装を説明するために，ここでは命名した型のみ紹介する．

```
type Input = Button ()
type Output = StaticText ()
type Bye = (Output, Input)
```

型を見ると一見今回の例は入力側と出力側が分離されているように見えるが，実際は入力側の UI であるボタンが自身の働きを変化させるために出力側として UI の役割も兼ねるので実質的には分離されていない．

そのためこの例でもウィジェットアクションの結合には widgetSPA 関数を使用し，take 関数の定義を入力側のウィジェット構造から考えるのではなく，入出力全体のウィジェット構造から抜き出す関数として考える．

```
takeB :: Bye -> Input
takeB = snd
```

ウィジェットアクションの定義については次のように行うことが出来る．setText とはウィジェットに引数で受け取った文字列を当てはめるための関数である．また，closef 関数はメインウィンドウを閉じるための関数である．

```
actWA :: WidAct Bye
actWA = widgetSPA takeB byeWA

byeWA :: WidAct Bye
byeWA = do left $ setText "Goodbye"
           right $ setWidAct closef
           right $ setText "Close"
```

後はこれまでの例と同様に putWAinUI 関数を使って UI にウィジェットアクションを埋め込んで UI が完成される．これによって Phooey で表現することができない例のうち，コントロールの働きを後になって変更する GUI を Neooey で表現できることが示された．

このプログラムの全ソースコードについては付録 C に載せる．

5　結論と展望

Phooey を元に Neooey を構築したことで UI については関数型らしい表現でコードを書けるようになった．また，UI とウィジェットアクションの分離を行ったことによって Phooey のコードでは表せなかった GUI の例も Neooey を用いて表現することができるようになった．

しかし実装にあたって問題点もいくつか残った．それを今後の展望とする．

問題点の一つはウィジェットをリスト的に扱えないという点である．現状の Neooey における UI は複数のウィジェット情報を組の状態で結合して保有するようにできている．各ウィジェットの型が異なる可能性があるために組を用いた定義を採用した．しかし wxHaskell ではレイアウトをリストとしてまとめて処理する関数も多くそういった関数を Neooey で表現するには限界がある．ウィジェットをリスト的に扱う一つの案として木構造のウィジェットを考えるというものがあるがこちらについても深刻な問題があり，まだ思案の段階である．

次に Neooey において FRP が活かしきれてないことである．Phooey の場合は UI 中にアクションを保持する機能が備わっていた．そして保持したアクションを使って FRP の実装を行っていた．しかし Neooey の場合は UI とウィジェットアクションでイベント処理に関する操作を分離してしまったために UI の中にアクションを保持させる必要がなくなってしまった．対してウィジェットアクションの方で FRP を活かすという方法が考えられるが現在はこの考え方が用いられていない．

また，より本格的な GUI でも難なく関数的な実装ができるように追究を行っていく．

謝辞 本論文の内容について議論していただいた桜井貴文氏 (千葉大) に感謝する．

参 考 文 献

[1] Conal Elliott and Paul Hudak. “Functional reactive animation.” ACM SIGPLAN Notices. Vol. 32. No. 8. ACM, 1997.

[2] Conal Elliott. “Push-pull functional reactive programming.” Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. ACM, 2009.

[3] Daan Leijen. “The wxhaskell wiki at haskell.org.” https://wiki.haskell.org/WxHaskell

[4] Daan Leijen. “wxHaskell: a portable and concise gui library for Haskell.” Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. ACM, 2004.

[5] Henrik Nilsson, Antony Courtney, and John Peterson. “Functional reactive programming, continued.” Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. ACM, 2002.

[6] Julian Smart and Stefan Csomor. Cross-platform GUI programming with wxWidgets. Prentice Hall Professional, 2005.

[7] Julian Smart, et al. “The wxWidgets library.” https://www.wxwidgets.org/

[8] Paul Hudak, et al. ”Arrows, robots, and functional reactive programming.” Advanced Functional Programming. Springer Berlin Heidelberg, 2003. 159-187.

A コード:カウンター

```
module Main where

import Graphics.UI.WX
import NeooeyUI
import NeooeyWidAct

-----------------------------------------------
-- widget types

type Input = Button ()
type Input2 = (Input, Input)
type Output = TextCtrl ()
type Count = (Input2, Output)

-----------------------------------------------
-- main function

main :: IO ()
main
  = runNAGUI "counter" mainUI

-----------------------------------------------
-- UIs

mainUI, preUI :: UI Count
mainUI = putWAinUI actB preUI
preUI = margin10 $ buttonset1 <|> buttonset2 <|> outputUI

outputUI :: UI Output
outputUI = entryUI [text := "0"]

buttonset1, buttonset2 :: UI Input
buttonset1 = takeRightUI $ label1 <-> button1
buttonset2 = takeRightUI $ label2 <-> button2

label1, label2 :: UI ()
label1 = labelUI "button1"
label2 = labelUI "button2"

button1, button2 :: UI Input
button1 = buttonUI [text := "+1"]
button2 = buttonUI [text := "-1"]
```

```
-- widget actions

actB, actB1, actB2 :: WidAct Count
actB = actB1 >> actB2

actB1 = widgetSPAC takeB1 (updateWA' (+1))
actB2 = widgetSPAC takeB2 (updateWA' (+(-1)))

-----------------------------------------------
-- take functions

takeB1, takeB2 :: Input2 -> Input
takeB1 = fst
takeB2 = snd
```

B コード:情報の相互交換

```
module Main where

import Graphics.UI.WX
import NeooeyUI
import NeooeyWidAct

-----------------------------------------------
-- widget types

type InOut = Button ()
type InterChange = (InOut,  InOut)

-----------------------------------------------
-- main function

main :: IO ()
main
  = runNAGUI "interchange" mainUI

-----------------------------------------------
-- UIs

mainUI, preUI :: UI InterChange
mainUI = putWAinUI actE preUI
preUI = margin10 $ buttonset1 <|> buttonset2


buttonset1, buttonset2 :: UI InOut
buttonset1 = takeRightUI $ label1 <-> button1
buttonset2 = takeRightUI $ label2 <-> button2

label1, label2 :: UI ()
label1 = fillUI $ labelUI "change"
label2 = fillUI $ labelUI "append"

button1, button2 :: UI InOut
button1 = buttonUI [text := "a"]
button2 = buttonUI [text := "b"]
```

```
-- widget actions

actE, actE1, actE2 :: WidAct InterChange
actE = actE1 >> actE2

actE1 = widgetSPA takeE1 change
actE2 = widgetSPA takeE2 appendT

change :: WidAct InterChange
change = do a <- left getText
            b <- right getText
            right (setText a)
            left (setText b)

appendT :: WidAct InterChange
appendT = do a <- left getText
             right (updateWA (a++))

-----------------------------------------------
-- take functions

takeE1, takeE2 :: InterChange -> InOut
takeE1 = fst
takeE2 = snd
```

C コード:コントロールの働き変更

```
module Main where

import Graphics. UI.WX
import NeooeyUI
import NeooeyWidAct
-----------------------------------------------
-- widget types
type Input = Button ()
type Output = StaticText ()
type Bye = (Output, Input)

-----------------------------------------------
-- main function

main :: IO ()
main
  = runNAGUI "Bye!" mainUI

-----------------------------------------------
-- UIs

mainUI, preUI :: UI Bye
mainUI = putWAinUI actWA preUI
preUI = margin10 $ messageUI <|> focusOnUI byeUI

messageUI :: UI Output
messageUI = staticTextUI [text := "Hello World"]

byeUI :: UI Input
byeUI = buttonUI [text := "Bye"]



-- widget actions

actWA :: WidAct Bye
actWA = widgetSPA takeB byeWA


byeWA :: WidAct Bye
byeWA = do left $ setText "Goodbye"
           right $ setWidAct closef
           right $ setText "Close"

-----------------------------------------------
-- take functions

takeT :: Bye -> Output
takeT = fst
takeB :: Bye -> Input
takeB = snd
```