Haskellでシグナル関数を利用した　FRP に基づく GUIライブラリの実装

眞々田 泰裕

本論文では強い型付けを持つ関数型言語 Haskell による Arrowized Functional Reactive Programming の考え方を用いた GUI のライブラリ Yeooey を実装する．Yeooey は小さい GUI ユニットを関数結合的に組み合わせることでより大きな GUI ユニットを構築する実装方式をとる．また、木構造を用いて GUI のウィジェット構造を管理することで GUI の構造によって異なっていた型を統一し, リスト処理をはじめとする GUI ユニットの一括操作を可能とする．Yeooey ではイベント処理や内部処理を表現するために pull ベースの AFRP ライブラリである Yampa を使用する．シグナル関数ベースで実装することで内部処理を副作用から分離し, データの流れや入出力の関係を追いやすくする．また,push ベースの FRP の考え方と共存させることも可能にする．また, 本論文では実装したライブラリを用いて GUI の例を構築し, 実装方法や他の GUI との比較を行う．

The goal of this paper is to develop the GUI library ”Yeooey”, which uses arrowized functional reactive programming in Haskell, which is a strongly typed functional programming language. GUIs with Yeooey uses a implementation method in which we create a GUI by composing some GUIs functionaly. We unify GUI widget types and use a tree structure to manage them, so we can process GUIs at once by using list-processing functions. We use the pull-based AFRP library Yampa to describe interactive processing. The use of signal functions separate it into pure operations and side-effects and makes easy to comprehend data streams and relationship between input and output. Also, Yeooey enable to use push and pull-based FRP. Furthermore, we show how to use Yeooey by use of GUI examples and compare other GUIs.

1　はじめに

GUI は従来命令型言語による実装が主だっていた．しかし大規模な GUI になると変数や関数の依存関係が複雑化し, それ故に予期せぬシステム異常が発生し易い．

そういった問題の解決法として注目されたのがオブジェクト指向プログラミングであり, 現在ではオブジェクト指向の性質を持ったプログラミング言語による GUI が主流となっている．

しかしそれもいいところばかりではない．規模が大きくなればオブジェクト同士の関係やメッセージの流れがつかみにくくなり, それによってエラーが発生したり, 原因箇所の発見や修正が難しくなり易い．

それに対し, 近年は関数型言語を用いた GUI の実装が注目されている．

関数型言語が特徴として持つ参照透明性はデータのや関数の依存関係を捉えやすいものとし, エラーの発生を軽減し, 発見し易くすることで強固なシステムを確立することができる．その中でも Haskell をはじめとする型に厳密な言語であればその性質はより増強される．

GUI は副作用を伴うものであるため, その点に関してはなるべく純粋であらんとする関数型言語に適合しないようにも思われる．

しかしそれでも関数型言語の特長を副作用のある領域で有効に使うために FRP(Functional Reactive Programming) [5] というフレームワークがある．

そこで本論文では関数型言語の中でも特に型に厳密な Haskell で FRP を活用し, GUI の実装における

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.

可能性を模索する．

その目的を基に以前 Neooey [10] という GUI ライブラリを作成した．これは別の GUI ライブラリ Phooey [3] を元に作られている．Phooey は小さな GUI ユニットを関数的に結合させせることによってより大きな GUI ユニットを作ることができる特長を持つ．この Phooey の性質は魅力的であるがその一方で Phooey によって実装可能な GUI の範囲は狭く, 大規模な例に応用することができなかった．その要因の一つと考えたのが GUI ユニットの構造である．Phooey における GUI ユニットはインターフェースを担うレイアウトの情報と内部関数を担うイベント処理を一つにまとめた構造をしている．この時 GUI ユニットを結合させる際にレイアウトと内部関数も同時に結合するのでコードの可読性が高まり,GUI ユニットを直感的に扱いしやすくするメリットはあるが, それ故に実装の柔軟性に欠ける．こういった問題を解決させるため, Neooey としての出発点はレイアウトと内部処理を分離するというところにあった．実際に Neooey では Phooey が持つ GUI ユニットの結合性をレイアウトと内部処理のそれぞれで保持し, その上で GUI の表現性を高めることに成功したがその一方で課題もいくつか見つかった．

今回作成した Yeooey は Neooey を改良したものである．特徴をまとめると次の通りである．

- レイアウトと内部処理の分離: Neooey と同様に GUI をレイアウトを担う GUI ユニットである UI ユニットとイベント等の内部処理を担うウィジェットアクションに分けてそれぞれで構築する．それらをまとめて起動することで一つの GUI として動かすことができる．
- WidgetTree の導入: Neooey では UI やウィジェットアクションの型がウィジェット構造によって変化するものになっていた．Yeooey では WidgetTree を導入することでリスト処理をはじめとする GUI ユニットの統一的な操作が可能となった．
- ウィジェットアクションの実装に Yampa を導入: Neooey では個々のイベント処理を行うウィジェットアクションを関数的に結合させて大きなウィジェットアクションを形成するものとなっていた．このとき形成されるウィジェットアクションは push ベースの FRP に基づくものであった．Yeooey ではウィジェットアクションの実行にシグナル関数を活用し, push ベースと pull ベースの双方の FRP に基づく処理を可能とした．シグナル関数の処理部分と副作用を伴う入出力部分を分離して用意することによって入出力の関係や値の流れを読みやすくする設計がなされている．

また, 本論文では Yeooey を用いて GUI の例を実装し, 使用方法の紹介や Neooey との比較を行う．

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 Yeooey の背景にある GUI ツールキット及びライブラリについて説明する．第 4 章で Yeooey の実装とを行い,Neooey との比較を行う．また,Yeooey を用いた GUI の作成法を例を挙げて説明する．そして第 5 章で関連研究, 第 6 章で結論を述べる．

2 FRP

命令型プログラミングやオブジェクト指向プログラミングをはじめとするプログラミングパラダイムは多数存在する．そのひとつに FRP(Functional Reactive Programming) がある．FRP は RP(Reactive Programming) に関数型の特徴を組み合わせたパラダイムである．FRP について述べるために, まずは RP について言及する．

RP は GUI や株システム, ロボットなどの外界とのやり取りが多く存在するような副作用を伴うコーディングで有用なパラダイムの一つである．副作用が多いコードは入出力の正当性を追うのが難しく, 予期せぬ値の代入や変更が行われたりすることでバグが発生し易く, 原因の特定がしにくくなる．

例えば伝統的なパラダイムのひとつである命令型プログラミングは内部で用意する変数や行わせたい命令, イベントの設定を実行させたい順番に記述する．このようなコーディングはその命令周辺の処理の流れを把握する上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規模が大きくなる

と互いの依存関係を捉えることが困難になり, 不正な値の参照が発生し易くなる. それに対して RP では入出力で取り扱う値を時間によって変化するものとして考え, この値との関係を記述してプログラムをくみ上げる.

例えば次のような例を考える.

```
x = 3;
y = 2 * x;
x = 5;
print (y);
```

伝統的なパラダイムでは 6 が印字される. y の値を決定する際の x の値は 3 であり, その後で x の値が変更されても y の値に影響を及ぼすことはない. 対して RP においては 10 が印字される. 2 行目の $y = 2 * x$ は時間によって変化する値 x と y の関係として宣言されたものであり, x の値がその後変化するとそれに反応するように y の値が再計算される. この考え方は予期せぬところで変数の中身が変わってもそれぞれの変数の関係の記述によって各値の再計算が行われるためにバグが発生しにくく, 値を追いやすくなり保守性が高まる. また一般に RP のコードは長さが短くなるという特長がある.

尚,RP において時間によって連続的に変化する値のことをリアクティブ (reactive)，リアクティブ同士のデータフローをふるまい (behavior) と呼ぶ. また，時間によって離散的に与えられる値をイベント (event) と呼ぶ.

RP の考え方は名称以上に浸透しており, 例えば Microsoft Excel などの表計算ソフトでその考え方が使われている.

これらの考え方を関数的に実現したものが FRP である. FRP は Elliott, Hudak による論文 Fran [5] が原形になっている. FRP では時間によって変化するリアクティブやイベントを主にストリーム (stream) で表現する. ストリームは時間によって変化する値のリストとして表すことができ, 次のように時間と値の連想リストとして考えることができる.

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

リアクティブとイベントは連続的と離散的という違いがあるが深い関連性もある. リアクティブは初期値及び値の変化を表現するイベント, イベントは最初の発生を待つリアクティブとして表現することができる. Elliott による論文 [4] 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.

このような時間によって変化する値をふるまい関数のもとで動かすことによって FRP に基づいたコードが表現される. FRP で設計されている関数部分は純粋なので, 値の正当性を保証する.

FRP はリアクティブやふるまい関数を処理するタイミングによって push 方式と pull 方式の二つの駆動方式に分類することができる. push 方式は一端の値が変化した際, その変化を他の値にも伝播させ, 芋づる式に再計算を行う方式である. つまりイベントが発生したときにそれに合わせて再計算を行うことが一般的である. 一方で pull 方式は一端の値が変化した時ではなくその値の参照が行われる時に再計算を行う方式である. 定められたサンプリング間隔に合わせて各値の再計算を行うことが一般的である.

この二つの方式の優位性は値が変化する頻度によって左右される. 変化の頻度が高い場合, その変化を逐一伝える push 形式ではタイムリークやスペースリークを多く生じさせる原因となる. そのためその情報が必要となるタイミングで再計算を行う pull 形式の方が優れている. 逆に変化の頻度が低い場合は push 形式の方が優れている. 値の変化に即時反応して計算させることができるためタイムラグが小さくて済むためである.

FRP にもいくつか種類があり, そのうち本論文と関係するのは AFRP(Arrows-based Functional Reactive Programming) である ([7], [9], [1]). AFRP はアローの一種であるシグナル関数 (signal function) を利用して FRP を体現したものである. AFRP においてイベントやリアクティブはシグナル (signal) と呼ばれる. そしてそれらのふるまい関数に当たるのがシグナル関数である.

シグナル関数はふるまいのようなものでシグナル (signal) と呼ばれるリアクティブを変換する関数である. AFRP では複数のシグナル関数を結合し，大き

なシグナル関数をつくることでプログラムの構築を行う.

3 Yeooey の背景にある GUI ライブラリ

この章では Yeooey の開発に当たってその内部で使用されている Yampa, wxHaskell 及び Yeooey の以前のバージョンである Neooey について言及する.

3.1 Yampa

Haskell における AFRP の伝統的なライブラリとして Yampa が挙げられる. Yampa は Fran [5] に基づいて開発された Haskell の GUI ライブラリであり, Yale Haskell Group [12] によって開発された. FRP としての基本設計は pull 方式に基づいている.

Yampa ではシグナル関数を表現するための型として SF が使われる.

```
data SF a b
```

SF 型はアローとして扱うためのプリミティブな演算子がいくつか用意されており, それらを活用して一つのシグナル関数を構築する. 以下にその演算子の例を載せる.

```
arr :: (a -> b) -> SF a b
(>>>) :: SF a b -> SF b c -> SF a c
first :: SF a b -> SF (a, id) (b, id)
second :: SF a b -> SF (id, a) (id, b)
(&&&) :: SF a b -> SF a b' -> SF a (b,b')
loop :: SF (a, p) (b, p) -> SF a b
loopPre :: b -> SF (a, p) (b, p) -> SF a b
```

3.2 wxHaskell

wxHaskell [8] は C++で記述されている GUI ツールキット wxWidgets を Haskell で扱えるようにバインディングした GUI ツールキットである.

現在も更新は行われており, 公式ページ †1 から無償で入手することができる.

ボタンやテキストコントロールなどの GUI を構成するオブジェクトをウィジェットと呼ぶ. wxHaskell では専用の関数でウィジェットを生成することができる. このとき同時にウィジェットハンドラが用意され, このハンドラを通してウィジェットに視覚的な変更, あるいは内部の処理を要求することができる.

GUI ウィジェットはそれぞれプロパティを保持している. プロパティはいくつかの属性の集合で表され, ウィジェット上に表示される文字列やサイズなどの情報が与えられる. wxHaskell では属性のリストとしてプロパティを表現しており, ウィジェットハンドラを通して属性を定めることが可能となっている. 以下に wxHaskell による GUI の簡単な実装の例を載せる.

```
main :: IO ()
main = start hello

hello :: IO ()
hello
  = do f    <- frame
                 [ text := "Hello!"]
       quit <- button f
                 [ text := "Quit"
                 , on command := close f]
       set f [layout := widget quit]
```

3.3 Neooey

Neooey [10] は GUI ユニットの結合性を意識して構築された Haskell の関数型ライブラリである. そのベースとなる GUI ライブラリには wxHaskell が用いられている. このライブラリは Elliott によって開発された Phooey [3] の問題点を解決させるべく実装したものである. Phooey においては UI と呼ばれる GUI ユニットがレイアウトとふるまい関数の両方を同時に保有するように設計されている. そのことは手軽に GUI を構成できるメリットがあるが, その一方でイベントのやり取りが複雑になると実装が難航するという問題があった. そこで Neooey ではレイアウトを保有する GUI ユニットとふるまい関数を保有する GUI ユニットを分離し, それぞれ設計してからまとめ上げる形にすることによって実装できる GUI の表現幅が広がった. しかしその一方でいくつか問題点も見られた.

- GUI ユニットに対して包括的な処理ができない:

†1 https://wiki.haskell.org/WxHaskell

Neooey では各 GUI ユニットが自身の構造を表現する型をそれぞれ持っている．そしてそれらを結合させた場合はウィジェットを表す型もタプル結合される．

```
uiAppend :: UI (TextCtrl, Button)
uiAppend = uiEntry <-> uiButton

uiEntry :: UI TextCtrl
uiButton :: UI Button
```

しかし GUI ユニットが保有しているウィジェット情報が異なる場合はリスト構造などの複数の GUI ユニットに対して一括で処理を与える操作が制限されてしまう．

- 大規模な GUI ユニットの型が長大になる: 前項でも述べた通り, Neooey では GUI ユニットを結合させた場合, ユニットが保有するウィジェット型がタプル結合される．このことは GUI の規模が大きくなればなるほど型が長く複雑になってしまう問題を引き起こす．そこで結合された型にその都度新たな名前を定めることによって可読性を助ける工夫をユーザ側に促していた．
- pull ベースで実装することができない: FRP はイベントやデータを処理するタイミングで push 方式と pull 方式の 2 つに区分される．push 方式はイベント駆動とも呼ばれ, ユーザの入力などによって外部からイベントが発生した際に状態の再計算を行う方式である．それに対して pull 形式はデータ駆動とも呼ばれ, 定められた間隔でデータが与えられ, その際に状態の再計算を行う方式である．Neooey では push 方式で実装するように設計されており, pull 方式による実装が行えないようになっていた．

これらの問題を解決させることが次章で述べる新しいライブラリの開発の動機づけとなっている．

4 Yeooey

この節では我々が今回実装した GUI ライブラリ Yeooey とその特徴について述べる．Yeooey は以前実装を行った GUI ライブラリ Neooey をベースに改良を加えて汎用性を高めたものである．Neooey の特長の一つである GUI をウィンドウやウィジェットのレイアウトを表す UI ユニットと GUI 内でやり取りする内部データの処理を表すウィジェットアクションを分離して記述する点については共通している．本節では Yeooey の構造に関して

- WidgetType
- `UI` ユニット
- ウィジェットアクション
- expand 関数

の順で述べる．また, Yeooey を用いて GUI を実装する方法を具体例を挙げて述べる．

尚, 開発に当たって使用したライブラリやモジュールのバージョンはそれぞれ wxWidgets が 3.0.2, GHC が 7.10.3, Yampa が 0.10.7, wxHaskell が 0.92.2.0 である．

4.1 WidgetType

Yeooey ではウィジェット構造を表現するために `WidgetType` 型を導入する．`WidgetType` 型はボタンやテキストコントロールなどの個々のウィジェットを一つのデータ型としてまとめたものである．`WidgetType` 型はその内部で指定されたウィジェットのハンドラを保有する.GUI に対する処理は Yeooey 内部においてこのウィジェットハンドラを経由して行われる．

```
data WidgetType =
    NoWidget
  | WMWin MWin
  | WWin  Win
  | WBitmapButton (BitmapButton ())
  | WBitmapToggleButton
  ..(中略)
  | WTextCtrl (TextCtrl ())
  | WToggleButton (ToggleButton ())
  | WTreeCtrl(TreeCtrl())
```

実際の GUI では階層的にウィジェットの管理が行われる．そこで Yeooey ではウィジェットの階層構造を `WidgetTree` 型で表現する．`WidgetTree` 型は `WidgetType` 型で表される個々のウィジェットを木構造で管理するものである．

```
type WidgetTree = Tree WidgetType
```

WidgetTree 型においてウィジェットハンドラは葉のみが持つようにし, 節点にはウィジェットハンドラを持たせない. また, 節点が持つ枝の本数は任意の正整数可能であり, 設計としてはこれらの枝をリストの形で管理している.

4.2 UI ユニット

4.2.1 UI ユニットの構造

Yeooey では Neooey と同様にすべての GUI をメインフレームとメインコンテナを共通のレイアウトとして持ち, メインコンテナの中で独自のウィジェット構造を構成していくものとしている.

そこで GUI の視覚情報を担う構成単位として UI 型を導入する. UI 型はメインフレームとメインコンテナを参照して動くアクションとして定義され, さらに WidgetTree 型で表されるウィジェット構造とレイアウトの情報を保有している.

```
type UI =
  MWin ->
  Container ->
  IO (WidgetTree, Layout)
```

ここで, MWin は wxHaskell におけるウィンドウフレーム型 Frame () のシノニムであり, Container は Panel () などのウィンドウ内で使用される梱包ウィジェット型をまとめたデータ型である. そして Layout はレイアウト情報を示す型であり,wxHaskell で使用されている同名の型と同一のものである.

UI 型は直感的には上記のようなメインウィンドウとメインコンテナを引数にして動くアクションとして表現されるが, 実際は次のようにモナド変換子を使用した定義として表される.

```
type UI =
  ReaderT MWin
  (ReaderT Container
  (WriterT Layout IO) WidgetTree)
```

先の定義と比較すると ReaderT が型引数,WriterT が返り値の型に対応している. モナド変換子は引数を伴うモナド関数を状態を内包する単一のモナドとして扱うことができる. Yeooey では UI ユニットをモナド変換子で定義することで複数の状態を表面上一つのモナドで保有し, 結合的に扱えるなどのメリットを得ることができている.

UI ユニットを構築するためにいくつかの基本的な関数を提供している. そのうちのいくつかを次の小節で紹介する.

4.2.2 UI ユニットを新しく生成する

一般的な GUI はボタンやテキストエントリなどのいくつかの基本的なウィジェットの組み合わせで構成されている. Yeooey ではそういったウィジェットを新しく作成するための関数が多数用意されている. これらの関数はプロパティリストを引数にとり, そのウィジェットに行わせる処理を記述するためのウィジェットハンドラを WidgetTree として保有する UI ユニットを形成する. 行わせたい処理は後述するウィジェットアクションによって記述される. プロパティリストは wxHaskell の書式を用いており, 作成するウィジェットの種類に応じた属性を指定することができる. 以下にその例を示す.

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

buttonUI はボタンウィジェットを作成するための関数である. また textCtrlUI はユーザによって入力することができるテキストコントロールを作成するための関数である.

4.2.3 複数の UI ユニットを結合する

前小節の関数などによって作成された複数の UI ユニットは最終的に一つに統合される必要がある. それらを行うために Yeooey では UI ユニットを結合するための関数をいくつか備えている. 結合関数によって UI ユニットを結合するとその内部で保有する WidgetTree も結合される. このとき新たに生成させる WidgetTree は根にウィジェットハンドラを持たず, 結合される UI ユニットの WidgetTree をそれぞれ枝として持つ. 以下にその例を紹介する.

```
(<->), (<|>) :: UI -> UI -> UI
a <-> b = rowUI 5 [a,b]
a <|> b = columnUI 5 [a,b]

rowUI, columnUI :: Int -> [UI] -> UI
```

二つの UI ユニットを連結させる場合は演算子 (<->) 及び (<|>) を使用する．前者は横方向に, 後者は縦方向に結合させる働きを持つ．これらはそれぞれ rowUI, columnUI の特殊な例であるため, より細かい指定を行いたい場合はこちらを使用する．rowUI は横方向に, columnUI は縦方向に複数の UI ユニットをまとめて連結させることができる．また, 第一引数の Int 型は配置するウィジェット同士の間隔幅を表している．

4.2.4 UI ユニットにレイアウト情報を付与する

UI ユニットは GUI の視覚的な面を担うものであるため, そのレイアウトを調整するためのレイアウト関数を提供している．これらの関数は UI 型の値を変換する形で使用し, この関数らを通してウィンドウ内におけるウィジェットの配置やサイズ, 他のウィジェットとの関係などを設定することができる．以下にその例を示す．

```
marginUI :: Int -> UI -> UI
fillUI :: UI -> UI
minsizeUI :: Size -> UI -> UI
```

marginUI 関数は引数で与えられた UI ユニットのレイアウトの端に余白をつくる関数である．この関数は UI ユニットのレイアウトにおける最後の仕上げとして使用される場合が多い．また, fillUI 関数は与えられた UI ユニットのレイアウトを拡大して余白を埋めるレイアウト関数である．更に fillUI 関数は UI ユニットのサイズを指定する目的で使用する．

4.2.5 GUI を実行する

最終的に, ユーザーによって作成された GUI は起動する手段が必要となる．Yeooey では定義した GUI を起動するための関数として runYUI 関数を提供している．runYUI 関数はウィジェット構造や視覚的な配置を管理する UI とその内部処理を後述するウィジェットアクション WidAct を引数にとり GUI を起動する．

```
runYUI :: UI -> WidAct’ a -> IO ()
```

4.2.6 Neooey との比較

UI 型は Neooey でも導入していたが, ウィジェットハンドラの管理方法に違いがある．Neooey ではウィジェットハンドラそのものやその構造をタプル結合の組み合わせで表現することで管理を行っていた．ウィジェットハンドラの種類や構造が変わると型も変わるため, 異なる型の組み合わせでもまとめることができるタプルを使用した経緯があった．Neooey における UI 型は型引数を一つ伴うようになっており, そこにはしばしばタプル結合されたウィジェットハンドラの構造を表すようになっていた．この点については型を見ることで構造がわかるというメリットはあるが, 複雑なウィジェット構造を持つ UI ユニットでは型の表示が長く難解なものとなり, 違ったウィジェット構造を持つ UI ユニットをリストのような一括処理に対して限界があった．それに対して Yeooey における UI 型はウィジェット構造によらずその構造は WidgetTree 型となる．そのため型引数を伴わずに表現することができるようになり, 異なるウィジェット構造を持つ UI ユニット同士でもリストや木構造で包括的な処理や探索が可能となった．

4.3 ウィジェットアクション

4.3.1 ウィジェットアクションの構造

Yeooey では GUI のイベント処理や内部計算などの処理を行うための構造としてウィジェットアクションを導入している．すべてのウィジェットアクションはメインフレームとメインコンテナ, 及び WidgetTree を参照して動くアクションとして定義される．このうち WidgetTree については UI ユニットが保有するウィジェット構造と対応しており, ウィジェットアクションを UI ユニットに埋め込む際は対象の UI ユニットが持つものと一致している必要がある．

```
type WidAct' a =
  MWin ->
  Container  ->
  WidgetTree ->
  IO a

type WidAct = WidAct' ()
```

また, 実際に使用する際は UI 型と同様に直接引数を伴う形ではなく, ウィジェットアクションもモナド変換子を使用した定義として表される. これによって UI ユニットと同様にウィジェットアクションも単一のモナドとして扱うことを可能にしている.

```
type WidAct' =
  ReaderT MWin
  (ReaderT Container
  (ReaderT WidgetTree IO))
```

ウィジェットアクションを作成する際はモナドの性質を利用し, do 記法や (>>=) 演算子などを使用する. ただしウィジェットアクションに対してモナド結合を適用させる際は結合されるそれぞれのウィジェットアクションが保有しているウィジェットハンドラの構造を一致させておく必要がある. もし結合を行いたいウィジェットアクションが一致していない場合は後述する expand 関数を通して構造の拡張を行って一致させる必要がある.

4.3.2 イベントウィジェットアクション

ウィジェットアクションを実装する方式のひとつとして Yeooey ではシグナル関数を経由して行う方法を提供している. その方法とはまずユーザー入力など外部からイベントが発生するとシグナルを生成させる. 次にそれを入力シグナルとして GUI の本質的な計算部分を記述したシグナル関数を動かして出力シグナルを生成する. 最後にシグナル関数を通して得られたシグナルをもとに盤面を更新させるというものである. このとき, 入出力で与えられるシグナルを管理する機能を兼ね備えたウィジェットアクションとしてイベントウィジェットアクションを導入する.

イベントウィジェットアクションはほとんどウィジェットアクションと同一のものであるが, シグナルを一時格納するために SignalHolder を追加で引数にとるようになっている.

```
type EvWidAct' sig a =
  MWin ->
  Container ->
  WidgetTree ->
  SignalHolder sig ->
  IO a

type EvWidAct sig = EvWidAct' sig ()
```

ただしモナドとして扱うメリットを享受するため, EvWidAct についても UI ユニットや WidAct と同様にモナド変換子による表現が使われる.

```
type EvWidAct' sig =
  ReaderT MWin
  (ReaderT Container
  (ReaderT WidgetTree
  (ReaderT (SignalHolder sig) IO)))

type EvWidAct sig = EvWidAct' sig ()
```

イベントウィジェットアクションもウィジェットアクションと同様にモナドとして扱い, 結合や生成を行うことができる. また, 結合の際には結合されるそれぞれのイベントウィジェットアクションが保有しているウィジェットハンドラの構造を一致している必要がある. この点についても通常のウィジェットアクションと同じである. イベントウィジェットアクションについても保有するウィジェット構造が異なるものを結合させたい場合は後述する expand 関数を通して構造の拡張を行い, 一致させる必要がある.

4.3.3 シグナル関数をウィジェットアクションに埋め込む

シグナル関数経由でウィジェットアクションを作る場合は makeWA 関数を使う. この関数はイベントウィジェットアクションとシグナル関数, 及びウィジェットアクションを作る関数を引数に取り, それらをまとめ上げたウィジェットアクションを構成する.

```
makeWA ::
      EvWidAct inp
   -> SF (Event inp) oup
   -> (oup -> WidAct)
   -> WidAct
```

第一引数のイベントウィジェットアクションはシグナル関数に渡す入力シグナルを構成するために用意するアクションである．このアクションはボタンやクリックなどの GUI に対する操作と関連付けられ, 行われた操作に応じた入力シグナルを生成する．このシグナルはイベントシグナルとなって第二引数で指定したシグナル関数へと伝達される．

第二引数のシグナル関数は必ずイベントシグナルを入力シグナルとして動くものとして定義する必要がある．シグナルの動かし方は push, pull, push-pull の三種類を提供している．そのうち上記の `makeWA` は push-pull 形式の動かし方に則っている.pull を含む形式で動かした場合はシグナルが発生していない場合でもシグナル関数の再計算が行われる. このときシグナル関数の入力として `NoEvent` が与えられる．

第三引数の関数はシグナル関数を動かして得られた出力シグナルを引数に取る．その得られた値に応じて GUI の視覚的な更新を行わせる働きを持つ. このシグナル関数実行後に呼び出す事後処理用関数をシンク関数と呼ぶ．

4.3.4　入出力シグナル用ウィジェットアクション

`makeWA` 関数でウィジェットアクションをまとめ上げるには, シグナル関数の他に入力シグナルを作るイベントウィジェットアクションと出力シグナルを使って視覚面を整えるウィジェットアクションが必要となる．Yeooey ではこれらのウィジェットアクションを作るための関数を各種用意している. 以下にその例を示す．

```
setSigOnActionEWA :: sig -> EvWidAct sig
setTextWA :: ShowText a => a -> WidAct
```

イベントウィジェットアクションは関数名の末尾に EWA, ウィジェットアクションは関数名の末尾に WA をつけて区別している．`setSigOnActionEWA` 関数はボタンなどのアクションが外部から行われた際にシグナルを発生させるイベントウィジェットアクションである. また,`setTextWA` 関数は引数で受け取った `String` に変換できる値を反映させるウィジェットアクションである．

4.3.5　Neooey との比較

`WidgetAction` 型も Neooey には導入されていたが, いくつか違いがある. そのうちの一つは `UI` ユニットと同じように型引数を持つか否かにある.Neooey の方は `WidgetAction` 型についてもウィジェット構造に沿った型引数を伴うため, 探索や統一的な処理に限界があったが Yeooey ではそれが可能になっている．また, ウィジェットアクションの実装方法については Neooey と Yeooey で大きく異なる.Neooey ではウィジェットアクション内部で動的に変化する値を管理し,push 方式の FRP に基づくアクションを構築することができた. その一方で pull 方式で動かすことはできず, 自動で盤面を再計算するような GUI については実装がややこしく苦手としていた．その点 Yeooey ではウィジェットアクションの実装に Yampa 由来のシグナル関数を導入することで, push 形式でも pull 形式でも動かすことを可能とした．それにより自動で再計算される GUI についてもシグナル関数さえあれば容易に実装できるようになった．さらには内部計算を行うシグナル関数部分と GUI の入出力に関与するウィジェットアクションとが明確に分離されていることで値の流れを追いやすくなっている．

4.4　expand 関数

Yeooey において,`UI` ユニットやウィジェットアクションを結合する際はウィジェット構造が一致している必要がある. もし一致していない場合は特殊な関数を通してウィジェット構造を調整する必要がある．

ウィジェット構造の調整は `UI` ユニット側で行う場合とウィジェットアクション側で行う場合の二種類存在する．`UI` ユニット側で行う場合はウィジェットアクションと連携させる必要のないウィジェットを切り捨ててウィジェット構造を簡潔にする目的で使用する．また, ウィジェットアクション側で行う場合は保有するウィジェット構造の範囲を拡張する目的で使用する．これらの拡張関数は型クラス `ExpandableWidgetStructure` のメソッドとして提供しており, それによって `UI` ユニット, ウィジェットアクション, イベントウィジェットアクションのいずれにおいても適用させることが可能である．その拡張関数の一例は次の通りである. た

だしウィジェットの木構造の二つの枝を伴う節に対して使用するもののみをここでは取り扱う.

```
left, right ::
  ExpandableWidgetStructure wa =>
  Unop wa

(<=>) ::
  (Monad wa,
   ExpandableWidgetStructure (wa a)) =>
   wa a -> wa a -> wa ()
```

これらは UI ユニットに適用する場合とウィジェットアクション, 及びイベントウィジェットアクションに適用する場合で仕様がいささか異なる.

UI ユニットに対しては left 及び right 関数を使用することができる. 名前が示す方向のウィジェット構造を残し, そうでない方のウィジェット構造を除去する. ただし上下方向の結合に対しては上側のウィジェット構造のみを抜き出す場合は left, 下側のウィジェット構造のみを抜き出す場合は right を使用する.

ウィジェットアクション, 及びイベントウィジェットアクションに対しては left, right, (<=>) を使用することができる. left, right については引数で与えられたアクションのウィジェット構造が関数名が示す方向の枝になるように拡張する. また,(<=>) は二つのウィジェットアクション, あるいはイベントウィジェットアクションを結合する. 引数で受け取るアクションはウィジェット構造が一致している必要がなく, アクションの結合と同時に保有するウィジェット構造も結合される.

4.5 実装例：カウンター

本節では実際の例を通して Yeooey による GUI の実装の過程やその例を紹介する. 特にここではボタンクリックによってカウントアップが行われていくカウンターを実装する. ここで実装する GUI はボタンとテキストコントロールが一つずつあり, そのテキストコントロールには正の整数が記載されている. その数値はボタンが押された回数を意味しており, ボタンを一回押すたびにカウントが 1 ずつ増えていくというものである.

Yeooey で GUI の実装を行う場合は入出力,UI ユニット, シグナル関数,WA の順番で決定及び実装を行うことを推奨する. 本節においてもその順番で紹介を行う.

このカウンターの例の場合, 入力はボタンが押されたことによるイベント, 出力はテキストコントロールに表示させる文字列となる. この入出力はシグナル関数に対する入力シグナルと出力シグナルに直結する. そこで各シグナルに型を定めておくと見通しが立てやすくなる.

```
type SigInp = Int -> Int
type SigOup = Int
```

入力シグナルを示す型 SigInp はボタンが押されたときに伝達される情報の型を表す. 今回は Int 型の変換を行う関数を伝達させるようにし, シグナル関数内でそれをまとめ挙げる方式をとる. わかればいいので Int 型を与えている. また, 出力シグナルを示す型 SigOup は表示させる数値を扱うものとするため,Int 型である. ここで, テキストコントロールは文字列の形で出力されるため, 出力シグナルについては String 型として考えてもよい. どちらの定義で実装を行ってもシグナル関数実行後に呼び出すシンク関数によってつじつまを合わせることが可能である.

次は UI ユニットの定義である. ここでは思い描く GUI のイメージに合わせて小さな UI ユニットを結合的に構築しつつ構成する. そのとき必要のないウィジェットを保持しないなど, 保有するウィジェット構造についても気を遣っておく.

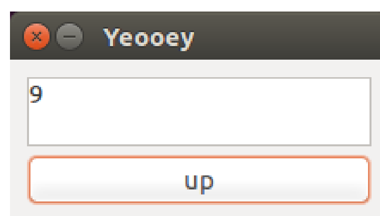

図 1 シンプルカウンター

```
counterUI :: UI
counterUI
  = marginUI 10 $ countBoardUI
    <|> fillUI upButtonUI

countBoardUI :: UI
countBoardUI
   = minsizeUI (sz 200 40)
    $ textCtrlUI [text := "0"]

upButtonUI :: UI
upButtonUI
   = buttonUI [text := "up"]
```

細かいレイアウト関数を適用してはいるが, 本質だけを述べるとテキストコントロールウィジェットを作成するために textCtrlUI, ボタンウィジェットを作成するために buttonUI を呼び, それらを (<|>) 演算子で縦方向に結合することで構成されている. このとき構築されるウィジェット構造は木構造の左枝にテキストコントロールウィジェットハンドラ, 右枝にボタンハンドラが来る.

次は処理の本体を記述するシグナル関数の実装である. 入出力シグナルの型が合うように構築する.

```
countSF :: SF (Event SigInp) SigOup
countSF = accumHold 0

accumHold ::
  b ->
  SF (Event a) b
```

accumHold は内部で値を保有し, 再計算の度にその値を出力するシグナル関数である. 保有する値の初期値は引数として与えられ, 入力イベントシグナルから受け取った関数でその都度更新及び出力が行われる.

シグナル関数の実装が終わったらあとはそれを包み込むようにウィジェットアクションを構築する. この例ではボタンが押されるまで待機し, 自動で再計算を行わせる必要がないため, push 形式の FRP に基づいて実行する. このとき makePushWA 関数を使用することで push 形式の実行が可能になる.

```
wact :: WidAct
wact = makePushWA evWA countSF sink

sink :: SigOup -> WidAct
sink = left.setTextWA

evWA :: EvWidAct SigInp
evWA = right setSigOnActionEWA (+1)
```

今回の例ではボタンからの入力しか考えないため, イベントウィジェットアクション evWA は単一のイベントウィジェットアクション setSigOnActionEWA で表現することができる. また, このボタンハンドラはウィジェット構造の右枝に位置することになるため rightEWA 関数で拡張を行う.

シンク関数についても唯一のテキストコントロールに対する処理しか考えないため, 単一のウィジェットアクション setTextWA で実装可能である. ただし, テキストコントロールハンドラはウィジェット構造の左枝に位置するので leftWA 関数で拡張を行う必要がある.

これで一通り必要な部品の実装ができたので runYUI 系の関数を使用してひとつにまとめれば完成である.

```
main :: Action
main = runFYUI counterUI counterWA
```

Yeooey の特徴の一つに拡張や UI ユニット, ウィジェットアクションの差し替えがしやすい点が挙げられる. 例えば UI ユニットは変えずにテキストコントロール上で右クリックをするとカウントをリセットする機能を追加したい場合は 0 に変換するシグナルを伴うイベントウィジェットアクションを追加して

```
evWA2 :: EvWidAct SigInp
evWA2 =
  setSigOnClickREWA (const $ const 0)
  <=> setSigOnActionEWA (+1)
```

とすることで達成される.

また,UI ユニットも変更してカウントダウン用のボタンを追加することを考える. この場合は追加するべきウィジェットの UI ユニットとイベントウィジェット

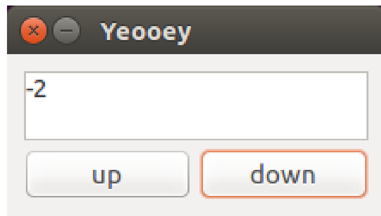

図 2　拡張されたカウンター

アクション, 及びそれぞれを結合したものを用意することで実装することができる.

```
counterUI2 :: UI
counterUI2 =
  marginUI 10
  $ countBoardUI <|> buttonsUI

buttonsUI :: UI
buttonsUI =
  fillUI upButtonUI
  <-> fillUI downButtonUI

downButtonUI :: UI
downButtonUI =
  buttonUI [text := "down"]

evWA2 :: EvWidAct SigInp
evWA2
  = setSigOnClickREWA (const $ const 0)
    <=> buttonsEWA

buttonsEWA :: EvWidAct SigInp
buttonsEWA
  = setSigOnActionEWA (+1)
    <=> setSigOnActionEWA (subtract 1)
```

4.6　実装例：ロケットランダー

前小節ではカウンターの小さな GUI の実装例を取り扱ったが, 本節ではもう少し規模を大きくしてロケットランダーと呼ばれるゲームを制作する. 尚, この例は Hunger による関数型言語 Elm で記述されたソースコード †2 を Yeooey で動くように移植したものとなっている.

ロケットランダーはロケットを操作してプラットホームに着地させるゲームである. 安全に着陸させることができればゲームクリアであり, 地形に衝突してしまうとゲームオーバーとなる.

この例はユーザからのキー入力を待つため push 的な側面を持つと同時にゲーム内時刻を経過させて定期的に盤面を更新する必要があるため,pull 的な側面も持つ例となっている.

実装方式はカウンターの例と同様にシグナルの型,UI, ウィジェットアクションの順で考察を行う.

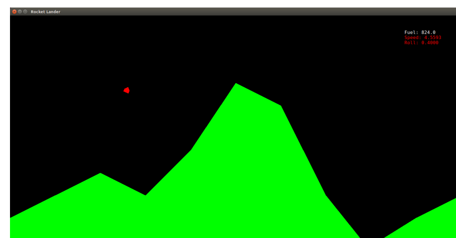

図 3　ロケットランダー

まず考えるのはシグナルに関してである. シグナル関数はゲームの状態を内部に保有し, 入力シグナルで送られるイベントに応じて更新を行う. 今回の例ではユーザのキーボード入力を入力シグナルとして考える. どのキーが押されたか, あるいは離されたかを捕捉するように設計する. また, 出力シグナルはシグナル関数がその時保持している内部状態を出力させる. 内部状態は盤面を表現するのに必要なロケットや地形, ゲームの実行状態を持ち, シンク関数を用いて盤面の更新を行う.

入力シグナルの型 `SigInp` 及び出力シグナルの型 `SigOup` については次のように定義される.

†2　https://blog.wearewizards.io/experience-report-rocket-lander-in-elm

```
data SigInp
    = KeyDown Key
    | KeyUp Key
    | NoOp

data SigOup = Model
    { gravity :: Gravity
    , ship :: Ship
    , state :: GameState
    , platformPos :: PlatformPos
    , height :: Height
    , width :: Width
    }
```

シグナルの考察が終わったら, 次は UI ユニットの実装である. 今回の例では描画用のスクリーンウィジェットを一つだけ配置するシンプルな形をとる. そしてスクリーン用のウィジェットの上でキーの入力を感知するように設計する.

```
landerUI :: UI
landerUI =
  minsizeUI (sz 3000 2000)
    $ panelUI []
```

次はウィジェットアクションの実装である. そのために入力シグナルを生成するためのイベントウィジェットアクション, 内部状態の更新を記述するシグナル関数, その都度状態を盤面に反映させるためのシンク関数を定める.

```
landerWA :: WidAct
landerWA = makeWA
             landerEWA
             landerSF
             paintSink

landerEWA :: EvWidAct Msg
landerEWA =
  do setSigOnKeyUpEWA
       $ KeyUp .fromEvKey
     setSigOnKeyDownEWA
       $ KeyDown .fromEvKey

landerSF :: SF (Event SigInp) SigOup
paintSink :: SigOup -> WidAct
```

イベントウィジェットアクション landerEWA は先ほどの考察通り, キーが押された場合と離された場合に応じたシグナルを発生させる. シグナル関数 landerSF とシンク関数 paintSink については詳しい定義を省略するが, シグナル関数 landerSF は入力シグナルやゲームの状態に応じた場合分け, シンク関数 paintSink はゲームの状態に応じた場合分けを意識することで比較的容易に実装することができる.

最後に実装した UI ユニットとウィジェットアクションをまとめて実行を行う. runNSYUI 関数はタイトルバーに表示させる文字列とウィンドウサイズを追加で指定することができる実行関数である.

```
main
  = runNSYUI
      "Rocket Lander"
      (sz 1600 800)
      landerUI
      landerWA
```

5 関連研究

関数型言語を用いて設計された GUI ライブラリは多く存在する. しかしその多くは wxWidgets や Gtk+ などの C 系の言語で設計されたライブラリを少なからず活用しており, GUI の実装スタイルも逐次実行的な形をとるものがほとんどである.

関数型言語で GUI の内部処理を記述するため, FRP を活用するライブラリも増えてきているがそれらは特に Haskell において活発に開発されている.

reactive-banana [6] は Apfelmus によって開発された FRP ライブラリである. できるだけシンプルに記述できるよう目指して現在も開発が続けられている. このライブラリは単独で使用するというよりも wxHaskell や Gtk2Hs などの他の GUI ライブラリと組み合わせて使用することを想定しており, 一般の GUI ライブラリに FRP の性質を持たせることができる. 構造としては主に FRP の概念を確立した Elliott の論文 [5], [4] に影響を受けており, 時間によって変化する値を Behavior 時間のある一点で発生する値を Event としてこれらを結合, 変換させることで処理を記述していく.

例えば wxHaskell と reactive-banana を組み合わ

せて GUI を作成する場合, まず wxHaskell による記法でウィジェットを作成し, レイアウトや簡単なイベントなどの設定を行い, そのあとで reactive-banana を活用した内部処理を記述していくことになる. ウィジェットハンドラについては wxHaskell によるウィジェット関数によって生成された生のものを流用する形になる. wxHaskell による GUI アクション内で内部処理を記述するため, GUI アクション内部での分離は行われているが, アクション単位での分離は通常行われていない. Yeooey の場合は `UI` ユニットでレイアウトやウィジェットハンドラの構造を設計し, ウィジェットアクションで内部処理を記述するように関数単位で分離しており, ウィジェットハンドラについては `UI` ユニットや `WidAct` 型の内部に秘匿して直接見えないように設計している. ただし,reactive-banana が提供する領域はあくまでも GUI の内部処理面だけなので, 他の GUI ライブラリと組み合わせることによって関数単位での分離は可能になる. reactive-banana の領域は Yeooey におけるウィジェットアクションを作成する際に実装する Yampa のシグナル関数部分に対応している.

FRP に注目が集まっている一方で FRP の考え方から脱却しようとする動きも存在する. Elm [2] は Czaplicki によって開発された関数型のドメイン固有言語である. Facebook が Open Source Software として公開している Java Script の UI ライブラリ React.js [11] などに影響を受けており, Elm は JavaScript にコンパイルを行いウェブサイトやウェブアプリケーションを生成するツールとして使用する. また, 設計に関しては強い型付けを持つ関数型言語にも影響を受けており,Haskell と類似した構文も多く見られる. 実装に関しては FRP の考え方を使用していたが, より簡潔に記述することを追究した結果, 現在のバージョンでは FRP を使用しないものとなっている.

Elm には Elm アーキテクチャという実装スタイルが確立されており, このスタイルに沿ったコーディングを推奨している. Elm アーキテクチャによると Elm コードは model,message,update,view の順に設計を行うとよいとされる. model はイベントの発生に応じて変化していく値のデータ型を表す. message は発生させるイベントのデータ型を表す. update は発生したイベントに応じた model の変換を定める関数である. view は画面のレイアウトやウィジェットに対するイベントの設定を行う関数である. 特に view は HTML5 の構文に沿って階層的に記述できるように設計されている.

Elm アーキテクチャの役割ごとに別々に記述するスタイルは Yeooey の実装スタイルと通じる所がある. 比較すると `UI` ユニットが view のレイアウト部分, ウィジェットアクションが update に対応する. またウィジェットアクション内部で動かすシグナル関数の入力シグナルの型が message, 出力シグナルの型が model に対応している.

6 結論と展望

レイアウトと内部処理を分けて実装する特徴を Neooey から引きつぐことで GUI を結合的に構築できる性質を保持した. その上でそれぞれの型を再考して型引数をとらない形にしたことで視覚的な煩雑さがなくなり, またリスト処理などの柔軟な処理が行えるようになった.

また,Yampa を導入し, シグナル関数ベースでウィジェットアクションを実装することで push-pull な FRP に基づく処理が可能となり, 簡潔に表現できる GUI の幅が広がった. さらにイベントウィジェットアクションを導入することで入力シグナルの伝達を表面から隠すことでユーザーによる不注意や想定外によるミスを起こしにくくし, 可読性を高めることができた.

現状の Yeooey では `UI` ユニットやウィジェットアクションの型が型引数をとらなくなったことでウィジェット構造をユーザーが管理する必要があり, それに従って `expand` 系関数を適用する必要がある. これは小規模な GUI ではさほど苦痛ではないが, 大規模な例になればなるほど構造を常に把握しなくてはならない. この問題を緩和, あるいは解消に努めていきたい.

尚, 本論文で紹介した Neooey のソースコード及び GUI の実装例についてはウェブサイト上で公開 †3 し

†3 http://www.math.s.chiba-u.ac.jp/ ymamada/yeooey.html

ている.

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] Courtney, A. and Elliott, C. : Genuinely functional user interfaces. Haskell workshop. 2001.

[2] Czaplicki, E. Elm : Concurrent frp for functional guis. Senior thesis, Harvard University. 2012.

[3] Elliott, C. : Phooey. `https://wiki.haskell.org/Phooey/`

[4] Elliot, C. : Push-pull functional reactive programming. Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. ACM, 2009.

[5] Elliott, C. and Hudak, P. : Functional reactive animation. ACM SIGPLAN Notices. ACM, Vol. 32, No. 8, 1997, pp.263-273.

[6] Apfelmus, H. : Reactive-banana. `https://wiki.haskell.org/Reactive-banana/`

[7] Hudak,P., et al. : Arrows, robots, and functional reactive programming. Advanced Functional Programming. Springer Berlin Heidelberg, 2003. pp.159-187.

[8] Leijen, D. : wxHaskell: a portable and concise gui library for Haskell. Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. ACM, 2004.

[9] Nilsson, H., Courtney, A. and Peterson, J. : Functional reactive programming, continued. Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. ACM, 2002.

[10] 眞々田泰裕: Haskell で GUI を扱うためのライブラリ Phooey の改良. コンピュータソフトウェア,Vol. 33,No.4,pp.30-49, 2016.

[11] React - A JavaScript library for building user interfaces `https://reactjs.org/`

[12] Yale Haskell Group. `http://haskell.cs.yale.edu/`