オブジェクト指向的GUIフレームワークへの関数的アプローチ

2013年6月28日 Revision 1.00

古田秀和 (FURUTA Hidekazu)

背景

オブジェクト指向言語を特徴づけるものとして、GUIを扱うオブジェクトを言語で表すことができることが考えられます。オブジェクトを言語の中でも、関数型言語的機能を持つものがあり、数学的な手法による関数の組み合わせによるプログラミングができます。しかし直接オブジェクト指向ライブラリを関数として扱うことは難しいと考えられます。この文章では関数の考え方により、GUIを表すオブジェクトの特徴を考察します。

ユーザーとGUIシステムの関係

ユーザーはある時刻にGUIシステム(例えばMicrosoft WindowsのGUI)に入力をすることができます。入力をするとその後、GUIシステムの状態が更新されます。GUIシステムへの入力から更新までをイベントということにします。

ユーザーはある時刻にGUIシステムの表示を見ることができます。これはユーザーから見た場合はある種のイベントと考えられますが、GUIシステムには何も影響を与えないので、イベントとは考えないことにします(GUIで反射するため反射的イベントということにします)。

イベントの木

あるイベントにはその「後の時点」を「前の時点」とする「次のイベント」があると考えることにします。ある時点に対して、ユーザーが選択できる入力が複数ある場合、複数の「次のイベント」があると考えます。ある最初の時点を「根」として、時点を「節点」、イベントを「枝」とする「根を持つ木」を作ることができます。

時点を節点とし、その間のイベントを枝とする木をイベントの木と呼ぶことにします。イベントの木は、ある時点を根とするものとし、この時点を開始時点ということにします。イベントの木の時点全体の集合をT、イベントの木のイベント全体の集合をEとします。イベントeに対して、そのイベントの直前の時点$before(e)\in T$とそのイベントの直後の時点$after(e)\in T$が決まります。イベントeに対して$b=before(e)$かつ$a=after(e)$を満たすとき$b\xrightarrow{e}a$と書くこととし、そのようなeが存在するとき$b<a$とします。この関係$<$の反射的推移的閉包を$\leq$とします。この関係によってTは順序集合(ここでは半順序集合の意味)になります。

イベントeに対して、そのイベントでユーザーが入力したデータ$data(e)$が決まります。これをイベントのデータということにします。イベントのデータ全体の集合をDとおきます。

GUIの状態を表す集合Gを考えます。GUIの状態とイベントのデータから新しいGUIの状態を決める写像$f:G\times D\to G$を考えます。このfに対して、次のような$g:T\to G$と$h:E\to G\to G$を作ることを考えます。イベントeと$b=before(e)$と$a=after(e)$に対して、$g(b)\in G$が決まっているものとします。このとき$g(a)=h(e)(g(b))=f(g(b),data(e))$として、開始時点から順に決めていくことにより$g$と$h$を定義することができます。

$$\begin{array}{ccc} b & \xrightarrow{e} & a \\ \downarrow g & \downarrow h & \downarrow g \\ g(b) & \xrightarrow[h(e)]{} & g(a) \end{array}$$

イベントパス

時点 t と t' が $t \leq t'$ を満たすとき、$t=t_1 \xrightarrow{e_1} t_2 \xrightarrow{e_2} t_3 \ldots t_{n-2} \xrightarrow{e_{n-2}} t_{n-1} \xrightarrow{e_{n-1}} t_n=t'$ というイベントの木の部分木が存在します。この部分木を時点 t から t' までのイベントパスと呼ぶことにします。このイベントパスを $e=e_{n-1} \circ e_{n-2} \circ \ldots \circ e_2 \circ e_1$ と書き、$t \xrightarrow{e} t'$ と書くことにします。

開始時点を t_0 とし、時点 t に対して t_0 から t までのイベントパスを $e(t)$ とおきます。$e(t)=e_{n-1} \circ e_{n-2} \circ \ldots \circ e_2 \circ e_1$（各 e_i はイベント）とすると $h(e(t))=h(e_{n-1}) \circ h(e_{n-2}) \circ \ldots \circ h(e_2) \circ h(e_1)$ とおくことによって、h を $EP=\{e(t) : t \in T\}$ から $G \to G$ への写像 $h : EP \to G \to G$ に拡張することができます。

$$
\begin{array}{ccccccc}
t_1 & \xrightarrow{e_1} & t_2 & \ldots\ldots & t_{n-1} & \xrightarrow{e_{n-1}} & t_n \\
\downarrow g & \downarrow h & \downarrow g & \ldots\ldots & \downarrow g & \downarrow h & \downarrow g \\
g(t_1) & \xrightarrow[h(e_1)]{} & g(t_2) & \ldots\ldots & g(t_{n-1}) & \xrightarrow[h(e_{n-1})]{} & g(t_n)
\end{array}
$$

このような、任意の長さのイベントパスに対して定義されている写像をH関数と呼ぶことにします。

Gオブジェクト

プログラミング言語F#(Microsoft)では（一般にはオブジェクト指向言語のフレームワークでは）、GUIを扱うクラスを、フレームワークに用意されたクラス(例えばMicrosoft .NET FrameworkのForm)を継承したクラス(サブクラス、派生クラス)として定義することができます。このようなクラスについて考えてみます。

イベントの木 $K=(T, E, before, after)$ とイベントのデータ D と $data : E \to D$ は、ユーザーのGUIシステムへの入力が行われた時刻と入力された内容を表しています。これに関数 $f : G \times D \to G$ を付け加えることによって、$g : T \to G$ と $h : EP \to G \to G$ を構成することができます。これはオブジェクト指向言語のGUIの処理を行うオブジェクトに対応するものと考えられます。

オブジェクト指向言語ではGUIの処理を行うオブジェクトはある「基底クラス」の「派生クラス」のインスタンスとして表されています。この「基底クラス」のようなものを表すものとしてベースGオブジェクトというものも考えます。「基底クラス」にイベントの処理を表す関数を付け加えた「派生クラス」がGUIの処理を行うクラスと考えられます。

$Gobj(f)=(K, D, data, f)$ をGオブジェクトと呼ぶことにします。$Gobj(f)$ の f を既定の処理を表す関数 f_0 で置き換えた $Gobj=(K, D, data, f_0)$ をベースGオブジェクトと呼ぶことにします。ベースGオブジェクトにイベントの処理を表す写像 f を付け加えてGオブジェクトにする処理 $setEvent(Gobj, f)=Gobj(f)$ が考えられます。f をイベント処理関数、$setEvent$ をイベント処理登録関数と呼ぶことにします。f と $setEvent$ を関数型プログラミング言語で記述することによって、GUIの処理を記述するシステムをF/Gシステムということにします。

純粋関数型言語の定義

変更可能な変数や実行順序を持たないような関数型言語では、関数は数学でいうような関数（部分関数）と考えることができます。ここではそのようなプログラミング言語Hを定義します。

V を変数の集合(可算)、C を定数の集合(有限)、F を関数の集合(有限)とします。各定数 c および関数 f には、項数(項の個数)を表す自然数(0以上)が対応しています。項は変数、「コンストラクタの項」、「関数の項」のどれかで、「コンストラクタの項」は定数(項数を n とする)と n 個の項の組（アリティ１以上のものはコンストラクタと呼ぶことにします）、「関数の項」は関数(項数を n とする)と n 個の項の組、と再帰的に定義されます。(項の集合 T は $T = V + C \times T^* + F \times T^*$ とすることもできます。コンストラクタ、関数に対して一つの項数が決まります。ここで集合 X に対して $X^n = X \times X \times \ldots \times X$（$n$ 個）、$X^* = 1 + X + X^2 + \ldots$、＋は集合の直和、×は集合の直積、１は１個の元からなる集合を表します）

節は頭部と本体の組で、頭部は1個の項、本体は有限個(0個でもよい)の項からなります。節の論理的な意味は、頭部の項をh、本体の項を$b_1, b_2, \ldots, b_n$、節に現れる変数を$v_1, v_2, \ldots, v_m$とすると「任意の$v_1, v_2, \ldots, v_m$に対して$b_1 \wedge b_2 \wedge \ldots \wedge b_n$ならば$h$」となります($\wedge$は論理積を表します)。

プログラミング言語Hの記法は、プログラミング言語Prologの記法と同様に、変数、定数、関数は英数字からなる文字列とします。変数名は大文字から始まるものとします。定数cから作られる項(項数n)は$c(t_1, t_2, \ldots, t_n)$のように書きます。$n=0$のときはcと書きます。関数fから作られる項(項数n)は$f(t_1, t_2, \ldots, t_n)$のように書きます。$n=0$のときはfと書きます。項数が1以上のとき、定数および関数は1個の出力変数(関数の戻り値を表す変数を出力変数と呼ぶことにします)を持つことがあります(出力変数を持たない場合もあります)。

節は
$h :- b_1, b_2, \ldots, b_n.$(本体の項の個数が1個以上のとき)
または
$h.$(本体の項の個数が0個のとき)
のように書きます。プログラミング言語Hのプログラムは節を並べた物となります。プログラミング言語Hのプログラムは、プログラミング言語Prologのプログラムと同様に以下のように書くものとします。

$h_1 :- b_{11}, b_{12}, \ldots, b_{1n_1}.$
$h_2 :- b_{21}, b_{22}, \ldots, b_{2n_2}.$
……
$h_m :- b_{m1}, b_{m2}, \ldots, b_{mn_m}.$

プログラミング言語Hのプログラムによる項の評価はPrologとほぼ同じですが、ただしPrologでは$h_1, h_2, \ldots$の順で最初に成功したものだけが評価に使うものとなるのですが、プログラミング言語Hではそのような順序はないものとします。

関数型言語とイベントの対応

プログラミング言語Hの項でH関数を表すことを考えます。イベント処理関数$f: G \times D \to G$はプログラミング言語Hの項$f(v, x, d)$で表すことができるものとします。すなわちvは出力変数でxとdが決まればvも決まるとします。ここでxはGの元を表すプログラミング言語Hの項、dはDの元を表すプログラミング言語Hの項となります。イベントパス$e=e_n \circ e_{n-1} \circ \ldots \circ e_2 \circ e_1$に対してリスト$[data(e_1), \ldots, data(e_n)]$をイベントパス$e$のデータリストと呼ぶことにします。$d_i$は$data(e_i)$の元を表すプログラミング言語Hの項とします。

$h(v, x, d_1, d_2, \ldots, d_n) :- f(v_1, x, d_1), f(v_2, v_1, d_2), \ldots, f(v_{n-1}, v_{n-2}, d_{n-1}), f(v, v_{n-1}, d_n).$
をプログラムに追加したとき項$h(v, x, d_1, d_2, \ldots, d_n)$は写像$f(v_i, x_i, d_i)$の合成になるため写像となります。
$h(v, x, d_1, d_2, \ldots, d_n)$はイベント$e$に対応する関数と考えられます。これは(=という関数があるとして)以下のようにも書くことができます。

$h(v, x, d_1, d_2, \ldots, d_n) :- v_0=x, f(v_1, v_0, d_1), f(v_2, v_1, d_2), \ldots, f(v_{n-1}, v_{n-2}, d_{n-1}), f(v_n, v_{n-1}, d_n), v=v_n.$
これをリストを使って以下のような意味に書き直すことによって、任意の長さのイベントパスに対応することができます。

$h(v, x, [d_1, d_2, \ldots, d_n]) :- v_0=x, f(v_1, v_0, d_1), f(v_2, v_1, d_2), \ldots, f(v_{n-1}, v_{n-2}, d_{n-1}), f(v_n, v_{n-1}, d_n), v=v_n$
これは以下のようになります。

$h(x, x, [\,]).$
$h(v, x, [d \mid l]) :- f(v_1, x, d), h(v, v_1, l).$

この写像をh^*とおきます。イベントパスにこの写像h^*を対応させる写像ηがH関数に対応するものとなります。

$$\begin{array}{ccccccc} t_1 & \xrightarrow{e_1} & t_2 & \ldots\ldots & t_{n-1} & \xrightarrow{e_{n-1}} & t_n \\ \downarrow g & & \downarrow g & \ldots\ldots & \downarrow g & & \downarrow g \\ g(t_1) & \xrightarrow[h^*(d_1)]{} & g(t_2) & \ldots\ldots & g(t_{n-1}) & \xrightarrow[h^*(d_{n-1})]{} & g(t_n) \end{array}$$

写像 η はプログラミング言語Hでは記述することができません。

プログラミング言語Hでは、イベント処理関数 f を記述することによってGUIを記述することはできますが、イベント処理登録関数は記述することができません。

次にプログラミング言語Hを拡張して、写像 η を表すことを考えます。プログラミング言語Hに $enter(x)$ と $return(x)$ という、項の代わりに書けるものを追加します。この言語をH^+と呼ぶことにします。言語H^+によって写像 η は以下のように表すことができます。

$h(x) :- enter(d),\ f(v, x, d),\ return(v),\ h(v).$

$enter(x)$ は、あるイベントに対応します。x はそのイベントのデータを表します。そのイベントの後、GUIの状態が変更されます。$return(x)$ はGUIの状態を表示に反映することを表します。$enter(x)$ と $return(x)$ には実行順序があり、左から順に実行されるとします。これによって写像 η はプログラミング言語H^+では関数のように記述することができます。これは関数ではないので疑似関数と呼ぶことにします。

$$
\begin{array}{ccccccc}
t_1 & \xrightarrow{e_1} & \longrightarrow & \longrightarrow & t_2 & \longrightarrow & \cdots\cdots \\
\downarrow g & \downarrow enter & & \uparrow return & \downarrow g & & \cdots\cdots \\
g(t_1) & \longrightarrow & \xrightarrow[f]{} & \longrightarrow & g(t_2) & \longrightarrow & \cdots\cdots
\end{array}
$$

例：ABCエディタ

次のようなGUIを使ったエディタを考えます。テキストボックスのウィンドウが表示されていて、そこにキーボードから入力できるものとします。入力できるものは「A」、「B」、「C」、「Ctrl-H」、「Ctrl-D」です。「A」、「B」、「C」が入力されるとテキストボックスの文字列にそれぞれ「A」、「B」、「C」の文字が追加されます。「Ctrl-H」が入力されるテキストボックスに文字列の最後の文字がもしあれば最後の文字が削除されます。「Ctrl-D」が入力されると終了します。

G を（上記の文字を含む）文字列または終了状態を表すある集合とし、D を（上記の文字を含む）文字のある集合とします。ある時点 t において、「A」を入力、「B」を入力、「C」を入力、「Ctrl-H」を入力、「Ctrl-D」を入力、の5種類のイベントがあります。イベント e に対して、$data(e) \in D$ は e に対応する文字を表します。イベント処理関数 $f : G \times D \to G$ は文字列 $s \in G$ と文字 $c \in D$ に対して、c が「A」、「B」、「C」のどれかであれば、$f(s, c) = \mathrm{concat}(s, c)$、$c$ が「Ctrl-H」であれば、$f(s, c) = \mathrm{backspace}(s, c)$、$c$ が「Ctrl-D」であれば、$f(s, c) = \mathrm{exit}$ とします。ここでconcatは文字列を連結する関数、backspaceは文字列の最後の文字を削除する関数、exitは終了コードを返す関数となります。終了コードが返されると状態は終了状態となり、その後のどのイベントに対してもイベント後は終了状態であるとします（終了状態はウィンドウは表示されず、入力は受け付けないとしてもよい）。この関数 f から前節のプログラミング言語Hによる関数 h^* を作ります。関数 h^* によってGUIを記述することができます。（図は前節と同じ）

$$
\begin{array}{ccccccc}
t_1 & \xrightarrow{e_1} & t_2 & \cdots\cdots & t_{n-1} & \xrightarrow{e_{n-1}} & t_n \\
\downarrow g & & \downarrow g & \cdots\cdots & \downarrow g & & \downarrow g \\
g(t_1) & \xrightarrow[h^*(d_1)]{} & g(t_2) & \cdots\cdots & g(t_{n-1}) & \xrightarrow[h^*(d_{n-1})]{} & g(t_n)
\end{array}
$$

前節のプログラミング言語H^+の疑似関数でGUIを記述することもできます。$return(x)$ で x が文字列のときはテキストボックスに表示された文字列が更新され、x が終了コードのときはウィンドウが非表示になるとします。（図は前節と同じ）

$$
\begin{array}{ccccccc}
t_1 & \xrightarrow{e_1} & \longrightarrow & \longrightarrow & t_2 & \longrightarrow & \cdots\cdots \\
\downarrow g & \downarrow enter & & \uparrow return & \downarrow g & & \cdots\cdots \\
g(t_1) & \longrightarrow & \xrightarrow[f]{} & \longrightarrow & g(t_2) & \longrightarrow & \cdots\cdots
\end{array}
$$

結論

プログラミング言語F#などでは、関数の定義によりGUIの処理が実現されています。プログラミング言語F#などの関数は数学の用語の写像とは異なるため、この処理と同様のものを写像を使って表しました。

状態を持たないプログラミング言語の関数は写像とほぼ同じものと考えられます。この関数の定義を拡張することによってGUIの処理を実現することができます。

今後の課題

Haskellのような言語（状態を持たない、関数の理論に基づいたプログラミング言語）でプログラミング言語H^+の場合と同様の議論ができるのかどうかはまだわかりません。また、状態を持たないプログラミング言語でイベント処理登録関数はどのように定義するのかは、言語の外部のシステムとの関係に依存するものとなるので、この関係を検討する必要があります。

Haskellは実行順序を持たないとも、場合によっては実行順序を持つとも考えられます。プログラミング言語Hは実行順序を持たない関数型言語、プログラミング言語H^+は実行順序を持つ関数型言語に対応するものですが、機能に違いがあるのかどうかは不明です。またプログラミング言語Hとプログラミング言語H^+の機能の違いも不明です。

参考文献

Haskell
http://www.haskell.org/haskellwiki/Haskell

F#
F# 言語リファレンス
http://msdn.microsoft.com/ja-jp/library/vstudio/dd233181.aspx

Microsoft Windows
http://windows.microsoft.com/ja-jp/windows/home

履歴

2013年6月28日 Revision 1.00