関数プログラミングによる構造を表す関数

2014年1月30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、(Microsoft .NET Framework のジェネリックコレクションのような)コレクション(たとえばList、Dictionary)を容易に扱うことができることが考えられます。このようなコレクションのクラスがオブジェクト指向のライブラリにあって、オブジェクト指向言語から使えるようになっています。オブジェクト指向のプログラミングに現れるものの中で、木のような再帰的な構造は再帰的なプログラムのプログラム上の位置を指定することによって実現されると考えることができます。

関数プログラミングではそのような構造は関数で定義されています。「関数による構造」はオブジェクト指向的な構造を表すものとして現れるものなのですが、ウェブ上で見ることができる文献などではとくに名前などはなく独立して扱われていないようです。この文章では関数型言語を拡張して、オブジェクト指向的な構造を定義する方法を考察します。

H/Fシステム

Vを可算集合、Cを有限集合とします(Vの元を変数、Cの元を定数と呼びます)。$T = V + C \times T^*$ と再帰的にTを定義します(Tの元を項と呼びます。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、$\times$は集合の直積、$1$は$1$個の元からなる集合を表します)。

$E = T + T \times E + E \times E$と再帰的に$E$を定義します($E$の元を式と呼びます。)式は項に$\lambda(t, e)$と$\alpha(e, f)$という記法を付け加えたものとします。($t$は項、$e$と$f$は式で、$\lambda(t, e)$は$f(t)=e$という関数$f$のことを表し、$\alpha(e, f)$は$f(e)$のことを表します。$\lambda(t, e)$はラムダ計算のラムダ抽象に対応するものですが、ここでは再帰的関数にも使うものとします。$\alpha(e, f)$はラムダ計算の関数適用に対応するものです。)

$P = E$とおきます(Pの元はを論理プログラムと呼ぶことにします)。

$s : V \to E$をCの元とλとαを保存するように$s : E \to E$に拡張することができます。このような定数とλとαを保存する$s : E \to E$を代入と呼び、代入全体の集合をSとおきます。代入s_1とs_2が$s_2 = s' \bullet s_1$を満たす代入s'が存在するとき$s_1 \leq s_2$と書くことにします。Sは$\leq$によって順序集合になります。

式に含まれる$\alpha(x, \lambda(t, e))$の部分を簡約可能部分式と呼ぶことにします。関数プログラムpに対してpに含まれる簡約可能部分式$\alpha(x, \lambda(t, e))$をxとtが項として一致するときe、そうでないときは未定義に置き換えることを簡約と呼ぶことにします。$\lambda(t, e)$はeが未定義のときは未定義、$\alpha(e, f)$はeかfかのどちらかが未定義のときは未定義とします。以下ではEとTは未定義を含むものとします。

関数プログラムpに対して簡約ができる限りすべて簡約を行ったものを正規形と呼び、$\rho(p)$と書くことにします。関数プログラムpに対して、代入sが存在して正規形$\rho(s(p))$が未定義ではなく項(またはλを含む項)になるとき、その項をpの結果と呼ぶことにします。結果は複数存在することもあります。

再帰的定義が行われている簡約可能部分式を再帰的簡約可能部分式(関数型言語のletrecに対応。letrecが使われる理由はいろいろあるようですが、ここではそれとは関係なく単にプログラムの構造を表すために分類します)と呼び、再帰的定義が行われていない簡約可能部分式を非再帰的簡約可能部分式と呼ぶことにします。再帰的簡約可能部分式を$\mathrm{letrec}(x, t, e)$、非再帰的簡約可能部分式を$\mathrm{letnonrec}(x, t, e)$、(それらをあわせた)簡約可能部分式を

letany(x, t, e)と書くことにします。

評価空間

以下では評価の順序について考えます。

式eがある（複数でもよい）letany(x, t, e)と定数からなる式であるとき、eを簡約可能ということにします。そうでないとき簡約不可能ということにします。

式eが簡約不可能であるとき結果はeであるとします。式eが簡約可能であるして、簡約可能部分式をletany(x_1, t_1, e_1)とします。代入s_1を$s_1(x_1)$が項になるものとします。$s_1(x_1)$が$s_1(t_1)$に一致しないときは結果は未定義であるとします。

$s_1(x_1)$が$s_1(t_1)$に一致するとき、$s_1(e_1)$が簡約不可能であるとき結果は$s_1(e_1)$であるとします。$s_1(e_1)$が簡約可能であるとして、簡約可能部分式をletany(x_2, t_2, e_2)とします。代入s_2を$s_1 \leq s_2$かつ$s_2(x_2)$が項になるものとします。$s_2(x_2)$が$s_2(t_2)$に一致しないときは結果は未定義であるとします。

$s_2(x_2)$が$s_2(t_2)$に一致するとき、$s_2(e_2)$が簡約不可能であるとき結果は$s_2(e_2)$であるとします。$s_2(e_2)$が簡約可能であるとき、結果が得られるまで同様のことを繰り返すものとします。（以下s_nで結果が得られるものとします。無限に繰り返すことも考えることができます。）この代入の列$s_1 \leq s_2 \leq \ldots \leq s_n$によって評価の順序が決まっていると考えます。

評価には時間がかかるものとして、ある式の評価が行われる時間の範囲を評価期間と呼ぶことにします。代入$s_1, s_2, \ldots, s_n$の順に代入が決まるとして、隣り合う代入の「差分」の評価期間をそれぞれ$z_1, z_2, \ldots, z_n$とおきます。$x_1, x_2, \ldots, x_n$をそれぞれ$z_1, z_2, \ldots, z_n$に対応する式と呼ぶことにします。

式e全体の評価期間をzとして$z = z_1 \gg z_2 \gg \ldots \gg z_n$と表すことにします。これは全体の評価期間が$z_1, z_2, \ldots, z_n$に分割されて、時間的な順序は$z_1, z_2, \ldots, z_n$という順になっていることを表します。このとき評価期間には$z_1 \leq z_2 \leq \ldots \leq z_n$という順序$\leq$があるものとします。

式に簡約可能部分式が複数ある場合や、代入の「差分」が複数ある場合、それらの「差分」に対応する評価期間をそれぞれ$z_1, z_2, \ldots, z_n$とすると$z = z_1 \vee z_2 \vee \ldots \vee z_n$と表すことにします。$z$は$z_1, z_2, \ldots, z_n$に分岐すると呼ぶことにします。$z_1, z_2, \ldots, z_n$の時間的な順序は決まっていないものとします。

ブレークポイント空間

C#のyield return文のようなものとして、項の後で出力変数を見ることができるようにすることができるものとします。このように設定された項に対応する評価期間をブレークポイントと呼ぶことにします。ブレークポイントは評価期間と評価期間の間と考えることもできます。ブレークポイントと評価期間の対を考えることによって、ブレークポイントと評価期間を同一視することもできます。

以下、非再帰的簡約可能部分式と再帰的簡約可能部分式に分けて考えます。$e \in E$の非再帰的簡約可能部分式をすべて簡約した式を$\nu(e)$、$e \in E$の再帰的簡約可能部分式をすべて1回だけ簡約した式を$\mu(e)$と書くことにします。

$e \in E$のブレークポイントをb_0とおきます。$B_0 = \{b_0\}$をeのブレークポイント空間と呼ぶことにします。$\nu(e)$のブレークポイントが分割と分岐によって

$b_0 = (b_{11} \gg b_{12} \gg \ldots \gg b_{1n}) \vee (b_{21} \gg b_{22} \gg \ldots \gg b_{2n}) \vee \ldots \vee (b_{m1} \gg b_{m2} \gg \ldots \gg b_{mn})$

と表されたとします。このとき$\nu(e)$のブレークポイントは

$B'_0 = \{b_{11}, b_{12}, \ldots, b_{1n}, b_{21}, b_{22}, \ldots, b_{2n}, \ldots, b_{m1}, b_{m2}, \ldots, b_{mn}\}$

とします。この1回の変換で得られるブレークポイントの集合

$\{b_{11}, b_{12}, \ldots, b_{1n}, b_{21}, b_{22}, \ldots, b_{2n}, \ldots, b_{m1}, b_{m2}, \ldots, b_{mn}\}$

をブレークポイント平面、b_0をその頂点と呼ぶことにします。

次にB'_0から始めて$\mu(\nu(e))$のブレークポイントについて考えます。上記の方法と同様に、$\nu(e)$の各ブレークポイントを頂点とするブレークポイント平面を考えることができます。同様に$\mu^n(\nu(e))$に対応するブレークポイント平面B_nを考えることができます。

ブレークポイント$b \in B_n$に対して$B_n(b)=\{x \in B_n \mid x \leq b\}$とおきます。これを$b$のブレークポイントパスと呼ぶことにします。ブレークポイントパスは$\leq$によって全順序集合になっています。$B_n(b)=\{b_1, b_2, \ldots, b_m\}, b_1 \leq b_2 \leq \ldots \leq b_m$とし、$b_1, b_2, \ldots, b_m$に対応する式をそれぞれ$x_1, x_2, \ldots, x_m$とおきます。$b_1, b_2, \ldots, b_m$にそれぞれ$x_1, x_2, \ldots, x_m$を対応させる写像を$\tau$とおきます。写像$\tau: B_n \to E$によってリストのような構造を構築すると考えることができます。（C#でyield returnを含む関数がリストを構築すると考えることができますがそれと同様に考えます。図はイメージです。）

$$\begin{array}{ccccccc} b_1 & \xrightarrow{z_1} & b_2 & \ldots\ldots & b_{n-1} & \xrightarrow{z_{n-1}} & b_n \\ \downarrow\tau & & \downarrow\tau & \ldots\ldots & \downarrow\tau & & \downarrow\tau \\ x_1 & \longrightarrow & x_2 & \ldots\ldots & x_{n-1} & \longrightarrow & x_n \end{array}$$

ズームインパス

この構造の構築が有限の時間で終わるとすると、さらにこの構造の時間的な変化を考えることができます。これをこの構造に対する評価期間と考えることにより、複数の評価期間を持つ構造を考えることができます。（順序集合の直積として順序集合$B_1 \times B_2 \times \ldots \times B_m$とを考えます。図はイメージです。）

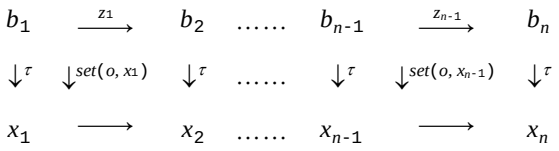
$$\begin{array}{ccc} B_1 \times B_2 \times \ldots \times B_m & \xrightarrow{b_1 \times b_2 \times \ldots \times b_m} & B_1 \times B_2 \times \ldots \times B_m \\ \downarrow{\scriptstyle \tau_1 \times \tau_2 \times \ldots \times \tau_m} & & \downarrow{\scriptstyle \tau_1 \times \tau_2 \times \ldots \times \tau_m} \\ E_1 \times E_2 \times \ldots \times E_m & \longrightarrow & E_1 \times E_2 \times \ldots \times E_m \end{array}$$

式eに対して$\nu(e)$のブレークポイント平面B'_0から始めて、$\mu^n(\nu(e))$に対応するブレークポイント平面B_nを考えます。B_nのあるブレークポイントb_1からその頂点b_2、b_2の頂点b_3、のように次々に頂点をたどっていくと、最終的にB'_0のブレークポイントb_mに到達します。このとき$b_m, b_{m-1}, \ldots, b_2, b_1$の列をブレークポイント$b_0$のズームインパスと呼ぶことにします。このズームインパス$b_1, b_2, \ldots, b_m$（添数を付け直します）は直積$B_1 \times B_2 \times \ldots \times B_m$の元と考えることもできます。

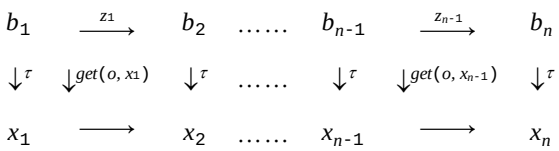
ある式eに対する$B_1, B_2, \ldots, B_n$のズームインパスの全体を$ZP(e)$とおきます。ズームインパスに式を対応させる写像（$ZP(e)$からEへの写像）を考えることにより、木のような構造を表すことができます。ズームインパスにはブレークポイントが対応しているので、ズームインパスの構造を無視してブレークポイントと考えることにより、リストのような構造を表すことができます。

H#/Fシステム

H/Fシステムのブレークポイントに対応する式xに対して、xのかわりに「$set(o, x)$」という記述を書くことができるようにします。oは変数のようなもので、$set(o, x)$は$\mathrm{letany}(x, o, \mathrm{letany}(x, t, e))$のようなものですが、$o$は変数ではなくてズームインパスを記録できるものである必要があります。このようなoを関数Fオブジェクトと呼ぶことにします。（これはオブジェクト指向的なオブジェクトを表すものと考えることができます。図はイメージです。）



関数Fオブジェクトを使うために x のかわりに「$get(o, x)$」という記述も書くことができるようにします。$get(o, x)$ は $\mathrm{letany}(o, x, \mathrm{letany}(x, t, e))$ のようなものとなります。（図はイメージです。）



順序集合の直積を考えることにより関数Fオブジェクトは1つにすることができます。

このようにH/Fシステムを拡張し、setがプログラム上の位置を記録できるもの、getがプログラム上の位置を参照できるものと考えることにより、木のような構造を表すことができます。このように拡張したものを$H^{\#}$/Fシステムと呼ぶことにします。ズームインパスを無視することにより、リストのような構造を表すことができます。

結論

関数プログラミングにプログラム上の位置を記録する機能を追加することにより、ある再帰的な構造を表すことができます。

今後の課題

プログラム上の位置を記録する方法については、この文章では厳密に定義されていません。関数によってオブジェクトを定義するとこのようになるのですが、この方法はいろいろな方法があるかもしれません。今後実現する方法を考えていく必要があります。

このような構造を関数型言語的機能から利用するためのパターンを考える必要があります。ウェブアプリケーションのクライアント側の記述がパターンによって表すことができるようにすることが目標となります。

参考文献

Haskell
本物のプログラマはHaskellを使う
http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248215/

F#
F#入門
http://winterradish.web.fc2.com/

履歴

2014年1月30日 Revision 1.00