B-4

システムの横断的性質のための型代数

Type Algebra for Crosscutting Concerns

山口陽平† 世木博久†
Youhei Yamaguchi Hirohisa Seki

背景

近年のシステム開発ではプログラミング言語をコードジュネレータやメタオブジェクトとともに利用することが増えている。典型的な分散システムではメソッドの遠隔呼出やエンティティのデータベース格納のためのプログラムを生成するためにコードジュネレータやメタオブジェクトが利用される。メソッドの遠隔呼出やエンティティのデータベース格納のような性質は Crosscutting Concerns と呼ばれ、多くのクラスを横断するため、既存のプログラミング言語で上手く記述することが難しいことが分かっている。Crosscutting Concerns を上手く記述し、プログラムの保守性・再利用性・堅牢性をたかめるために AOP[1]・Composition Filters[2]・Object Extension Framework[3]・Mixin Layers[4]など多くの提案が行われている。

しかし、従来の提案では横断されるクラスの型を変化させる Crosscutting Concerns について制限された議論しか行われていなかった。例えば、遠隔呼出などの性質では、それが横断するメソッドの呼出はネットワークを超えて行われる。そのためネットワークの状態によって呼出が失敗する可能性がある。それにも関わらず、従来の議論では、そのメソッドの型は失敗のための例外をスローするように変化されず曖昧にされてきた。その原因は継承・総称に限定された枠組の上で Crosscutting Concerns が議論されていたことにある。つまり、従来の継承や総称による型の合成では横断されるクラスの型の変化を表現できないことが根本的な原因である。

そこで従来の継承の枠組を包含する型の合成方法として型代数を提案する。さらに横断するクラスの型を変化させる Crosscutting Concerns を型代数によって上手く表現できることを例によって示す。

型代数

型とは項(変数など)の種類である。項は同じ型の変数にだけ代入できる。さらに型には包含関係があり下位の型の項は上位の型の項でもある。項 x が型 X であることを命題 $x:X$ で表現する。命題を論理演算子で結合することで複雑な型をとる項を作ることができる。例えば、命題 $x:X \wedge x:Y \vee \neg x:Z$ は項 x が型 X と型 Y を同時にとるか型 Z をとらないことを意味している。また、この命題を $x:X \times Y + -Z$ と項 x についてまとめて記述することもある。$\times + -$ は型の集合 S_T 上の演算子で、$\times$ は $\wedge$、$+$ は $\vee$、$-$ は $\neg$ に対応している。また S_T の単位元 I_T は *true*、零元 O_T は *false* に対応している。このように対応付けることでブール代数系 $(S_T;\times + -)$ を構成することができる。さらに、このブール代数上の順序関係 $\leq$ によって型の包含関係を表現する。以降では型の集合上のブール代数系のことを型代数系と呼ぶ。

†名古屋工業大学世木研究室

S_T は演算をそれ以上計算できない原始型の集合 S_P や関数型の集合 S_F やモジュール型の集合 S_M などで構成される。どの型の集合も S_T の部分集合である。

ここでこれらの型の集合を Java 言語に対応させてみると、S_P は char や int などの原始型の集合、S_F はクラスやインターフェイスに定義された関数の型の集合、S_M はクラスやインターフェイスの型の集合に対応させることができる。以降では $(S_T;\times + -)$ の部分型代数系 $(S_F;\times + -)$ と部分型代数系 $(S_L;\times + -)$ について考える。

関数型代数

関数型は関数の入力の型 a と出力の型 b の組 $a \Rightarrow b$ で表現された型である。次に関数型代数系 $(S_F;\times + -)$ の定義を示す。次のように定義することによって $(S_F;\times + -)$ はブール代数系となる。

定義：関数型代数系 $(S_F;\times + -)$

任意の型 $\forall a,b,c,d \in S_T$ に対して定義される関数型 $a \Rightarrow b, c \Rightarrow d \in S_F$ に対して次を定義する。

順序 $(a \Rightarrow b) \leq (c \Rightarrow d) \equiv (c \leq a) \wedge (b \leq d)$

単位元 $I_F \equiv (O_T \Rightarrow I_T)$

零元 $O_F \equiv (I_T \Rightarrow O_T)$

積 $(a \Rightarrow b) \times (c \Rightarrow d) \equiv ((a+c) \Rightarrow (b \times d))$

和 $(a \Rightarrow b) + (c \Rightarrow d) \equiv ((a \times c) \Rightarrow (b+d))$

否定 $-(a \Rightarrow b) \equiv (-a) \Rightarrow (-b)$

順序の意味

順序の定義は、任意の関数はより上位の入力型でより下位の出力型の関数に置換できることを意味している。より上位の入力型は元の関数への入力を全て代入できる。また、より下位の出力型の出力は元の出力を代入できる変数には必ず代入できる。

単位元と零元の意味

単位元の定義は、型が単位元である関数には任意の関数を代入できることを意味し、また零元の定義は、型が零元である関数は任意の関数に代入できることを意味している。

積・和・否定の意味

積の結果は元の型の下位になるという関係から積の定義を導出できる。同様に、和の結果は元の型の上位になるという関係から和の定義を導出できる。否定の結果と元の型の積は零元になり、否定の結果と元の型の和は単位元になるという関係から否定の定義を導出できる。本論文では導出の過程は省略する。

モジュール型代数

モジュール型は型付けされた項の組合せを表現する型である。既に紹介したように型付けされた項は $x:X$ のような命題で表現できるので、その組合せは命題を論理演算子で結合することで表現できる。次にモジュール型代数系

$(S_M;\times+-)$ の定義を示す。$(S_M;\times+-)$ は $\times+-$ を $\wedge\vee\neg$、I_R を $true$、O_R を $false$ に対応付けただけなので元のブール代数系の性質をそのまま継承している。

定義：モジュール型代数系 $(S_M;\times+-)$

型付けされた項で構成された任意の命題 $\forall a,b$ に対して定義されるモジュール型 $[a],[b]\in S_M$ に対して次を定義する。

順序　$[a]\le[b]\equiv a\to b$

単位元　$I_R\equiv[true]$

零元　$O_R\equiv[false]$

積　$[a]\times[b]\equiv[a\wedge b]$

和　$[a]+[b]\equiv[a\vee b]$

否定　$-[a]\equiv[\neg a]$

単純なモジュール型

例えばモジュール型 $[foo:A\Rightarrow B\wedge goo:B]$ は関数型 $A\Rightarrow B$ の関数 foo と型 B の項 goo を同時に持つ型という意味になる。Java 言語を例に説明すると、このモジュール型は入力型 A と出力型 B を持つ関数 foo と型 B の変数 goo を定義したクラスの型に相当する。

状態を持つモジュール型

Java 言語など従来のプログラミング言語ではメンバの直積でしかクラスを表現できなかったが、モジュール型代数系では論理演算によって更に柔軟にメンバを組合せることができる。例えば、モジュール型代数では次のように状態によってインターフェイスが変化する型を表現できる。

$$Server=\left[\begin{pmatrix}start:O_F\oplus\\ \begin{pmatrix}stop:O_F\wedge\\ setPort:\mathrm{int}\Rightarrow O_T\end{pmatrix}\end{pmatrix}\wedge\right.$$
$$\left.getPort:O_T\Rightarrow\mathrm{int}\right]$$

モジュール型 $Server$ はいつでも呼び出すことができる関数 $getPort$ を持っている。また、排他的な 2 つの状態（実行状態と停止状態）を持ち、各状態で呼び出すことができる関数が制限されている。

複数の入出力や例外をスローする関数型

これまでは関数型の入出力は 1 つだけであったが、モジュール型を組合せることで、次のように複数の入出力や例外を出力する関数型を表現できる。

$$hoo:[a:A\wedge b:B]\Rightarrow\begin{bmatrix}result:[c:C\wedge d:D]\vee\\ exception:E+F\end{bmatrix}$$

関数 hoo は 2 つの入力と 2 つの出力を持ち、2 種類の例外型をスローする可能性がある。hoo の出力がモジュール型になっていて、そのモジュール型のメンバとして本当の出力と例外の型を定義することで、例外の型を関数型に導入することができる。

横断的性質による型の変化

横断するクラスの型を変化させる Crosscutting Concerns を型代数によって上手く扱えることを例で説明する。まず、純粋に距離を求めることだけを想定して作られた関数型 $distance$ と純粋に遠隔呼出機能だけを想定して作られた関数型 $remote$ が次のように定義されているとする。

$$distance=\begin{bmatrix}from:double\wedge\\ to:double\end{bmatrix}\Rightarrow\begin{bmatrix}result:double\vee\\ exception:O_T\end{bmatrix}$$

$$remote=I_T\Rightarrow[result:O_T\vee exception:RemoteException]$$

次にこれらの関数を関数型代数の演算＋によって合成する。

$remoteDistance=distance+remote$

$$=\left(\begin{bmatrix}from:double\wedge\\ to:double\end{bmatrix}\Rightarrow\begin{bmatrix}result:double\vee\\ exception:O_T\end{bmatrix}\right)+$$
$$(I_T\Rightarrow[result:O_T\vee exception:RemoteException])$$
$$=([from:double\wedge to:double]\times I_T)\Rightarrow$$
$$\left(\begin{bmatrix}result:double\vee\\ exception:O_T\end{bmatrix}+\begin{bmatrix}result:O_T\vee\\ exception:RemoteException\end{bmatrix}\right)$$
$$=\begin{bmatrix}from:double\wedge\\ to:double\end{bmatrix}\Rightarrow\begin{bmatrix}result:double\vee exception:O_T\vee\\ result:O_T\vee exception:RemoteException\end{bmatrix}$$
$$=\begin{bmatrix}from:double\wedge\\ to:double\end{bmatrix}\Rightarrow\begin{bmatrix}result:double+O_T\vee\\ exception:O_T+RemoteException\end{bmatrix}$$
$$=\begin{bmatrix}from:double\wedge\\ to:double\end{bmatrix}\Rightarrow\begin{bmatrix}result:double\vee\\ exception:RemoteException\end{bmatrix}$$

以上のように型代数上の演算を繰り返し適用し型を計算することで $distance$ の型を変化させた新しい型 $remoteDistance$ を生成することができる。この性質によって遠隔呼出のように横断するクラスの型を変化させる Crosscutting Concerns を扱うことができるようになる。

$distance$ と $remoteDistance$ をよく見比べてみるとそれらの間には $distance\le remoteDistance$ という関係が成り立っていることが分かる。つまり演算＋によって型を汎化できる。従来のプログラム言語では特化だけが提供されてきたが、この例のように型の汎化も役に立つ。

まとめ

横断する型を変化させる Crosscutting Concerns を扱えるようにするために型代数を提案し、それが本当に有効かを例によって確認した。また関数型とモジュール型を組合せることで従来から議論されてきている型や状態によって変化するインターフェイスをもつ型などを表現できた。また型代数の演算で型を計算し型を導出できることを示した。

現在の型代数は総称を考慮していないので、総称を導入する必要がある。総称を導入することで型代数の演算は述語論理の制約を計算できるようになる。

参考文献

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, Aspect-Oriented Programming, ECOOP, 1997

[2] Lodewijk Bergmans and Mehmet Aksit, Composing Multiple Concerns Using Composition Filters, *Communications of the ACM*, October 2001

[3] Youhei Yamaguchi and Nobuhiro Inuzuka, A distributed object system with Object Extension Framework, SNPD, 2001

[4] Richard Cardone and Calvin Lin, Comparing Frameworks and Layered Refinement, ICSE, 2001

[5] Luca Cardelli, Peter Wegner, On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys Volume 17, Number 4, December 1985