ソフトウェア基礎論　25－2
（1988. 5. 26）

型継承の表示的意味論

A Denotational Model of Type Inheritance

立木 秀樹
Hideki Tsuiki

京都大学数理解析研究所
Research Institute for Mathematical Sciences,
Kyoto University

あらまし　オブジェクト指向プログラミングが広まるに従い、型継承とgenericな関数の重要度は高まりつつある。しかし、これらの厳格な数学的基礎は今の所与えられているとはいえない。本論文は、型継承とgenericな関数の数学的基礎として、それらに表示的意味を与えることを目指している。まず、階層的な型構造とgenericな関数をもった言語の表示的意味論を考えるための領域を構成する。この領域は、値と階層的な型のなす領域の数学的モデルである I-domain (domain with type inheritance) がgenericな関数に対応するものを射としてなすカテゴリの中で構成される。このカテゴリは cartesian closed category であり、M = MB + [M-->M] といった領域方程式はこのカテゴリの中で解を持つ。その後に、この等式の解となるI-domain の上で、ラムダ式を拡張した構文を持つ型付き言語の表示的意味を与える。

Abstract　Type inheritance and generics are considered to be very important in connection with the recent wide spread use of object-oriented programming. However, they are not yet given rigorous mathematical foundations. The purpose of this paper is to give, as a mathematical foundation, a denotational semantics to type inheritance and generics. A domain is constructed on which one can give denotational semantics to languages with hierarchical type structures and generics. This domain is constructed in the category DTI whose objects are I-domains (domain with type inheritance) which are mathematical models of hierarchical type and value, and whose morphisms correspond to generic functions. This category is cartesian closed and such domain equations as M = MB + [M-->M] are solvable in this category. On the solution of this equation, a denotational semantics is given to a typed language with lambda-calculus-like syntax.

1. 序

型継承

多くのプログラミング言語は型構造を持っており、値は計算機上での表現の仕方、および、適用可能な操作の種類に応じて型に分類されている。幾つかの言語は、さらに、型の間の継承の機構を持っている。

型継承については幾つかの考え方が存在する。[Cardelli'85] は、イデアル・モデル [MacQueen '86] に基づいた型継承の表示的モデルをあたえているが、そこでは、型継承の意味はイデアル間の包含関係であたえられる。このモデルでは、型の継承関係を反影するような構造は値領域上に存在せず、よって、値同士の間に関係が存在しないような均質的な値領域を階層的に分類するような型構造のモデルを与えている。

この論文で構成する表示的モデルは、 Cardelli のものとは基本的に異なる。このモデルでも、型に対応して値空間上の集合が存在しているが、それらは共通部分を持たず、よって、それぞれの値は唯一の型に属している。そして、型の継承関係は、それらの集合の間に型変換を行う関数が存在することにより定義される。

そのような型継承の例として、int と real の間の継承関係を考える。多くの言語では、型 int に属する値 1 と、型real に属する値 1.0 は別の値であるが、real に対して定義されている関数を int である 1 に対して適用した場合、1 はint から realに自動的に型変換され、1.0 にその関数が適用される。この場合、この型変換の関数によって int はreal を継承している。

もう一つ例を挙げる。各スロットが強く型付けされた構造体の型を持った言語を考える。構造体は、リスプにオブジェクト指向の機能を付け加えたいくつかの言語でクラスとして使われており、型継承を考える上で重要な型である。次の型定義を考える。

```
deftype person <age->integer,
                name->string,
                sex->boole>

deftype female-student <age->integer,
                        name->string,
                        sex->true,
                        school->string>
```

ここで、true は boole を継承している型で値 t のみが属しており、この例では t は女を表わしているとする。その時、型 female-student に属する値に対して型 person に属する値が、school のスロットの値を忘れ、sex のスロットの値を true からboole へ型変換することによって対応している。この対応を型変換の関数として、female-studentはpersonを継承している。

この場合、 female-student から person への型変換は多対一である。((注) 全てのスロットの値が等しい時に二つの値は等しいとする。)また、female-studentと同様に male-student

を定義した時、 male-student も person を継承しており、型 person に型変換した結果が同じ値になる値がfemale-student、male-studentの両方の型に存在している。このように、型変換が多対一である場合や、異なる型の値が型変換をした結果同じ型の同じ値になる場合も取り扱えるモデルを考える。

さらに、

```
deftype female    <age->integer,
                   name->string,
                   sex->true>

deftype student   <age->integer,
                   name->string,
                   school->string>
```

という定義があった時、female-studentは female と student の両方を継承している。 このような多重継承 （ multiple inheritance ）も取り扱えるモデルを考える。

これらの例では、型 A が型 B を継承している時に A ≧ B と定義することにより、型全体は ≧ を順序とする半順序集合(以下では poset と呼ぶ)となっている。((註) この論文では、二つの型が相互に型変換可能であるという状況は考えない。また、型の間の順序は、より一般的な型の方が小さくなるようにつけることにする。)また、値について考えれば、値 a が値 b に型変換可能な時に a > b と定義することにより、 値全体も > を順序とする poset となっている。さらに、後者の例で、female-student は student、female の両方の型を継承している型のうちで最も一般的なものである。すなわち、型の成す順序集合の上で、 female-student はstudent とfemaleの上限である。ここでは、幾つかの型を同時に継承している型が存在すれば、そのような型の内で最も一般的なものが存在するような型構造を考える。順序集合の言葉に直せば、これは、上界を持つ部分集合は上限をもつということである。順序集合において、そのような性質をsemi-coherent と呼ぶことにする。また、値全体のなす poset も、semicoherent であるとする。

この論文では、型継承のモデルを構築するための意味領域として、値のなす集合 D と、型のなす集合 T と、それぞれの値に対してその属する型を返すDから T への関数τの三つ組を考える。D は semi-coherent なcpo、Tはsemi-coherentな poset、τはある値のある型への型変換が一意に定まるための条件を満たす、単調な全射関数である。このような三つ組 (D, T, τ) を I-domainと呼ぶ。D のことを値領域、T のことを型領域と呼ぶ。τ(a) = A の時、値 aは型 Aに属すると呼ぶことにする。

Generic function

generic function とは、複数の型の引き数に対して適用可能で、その動作が引き数の型に応じて異なる関数である。型継承を持つ幾つかの言語は generic function を持っている。例えば、幾つかの言語では、足し算を表わす関数 + は引き数の型がintであっても real であっても適用可能であり、引き数が共にintに属する時は int の値を返し、どちらかの引き数の型が real の時はreal の値を返す generic function である。また、オブジェクト指向の機構を LISPに取り入れる標準化案として考えられている言語 CLOS (Common Lisp Object System) は、message passing の機能を実現するためにgeneric function を採用している[Bobrow, et al.'87] 。

ここで構成するモデル (D, T, τ) では、値領域 D の間の関数としてgeneric function のみを考えることにする。多くの実用的な関数は、全ての型の全ての値に対して適用可能ではない。そのような関数を D から D への関数としてモデル化するために、T には全ての型を継承している型 nullが存在し、D には型 null に属する唯一の値 nil が存在するとし、引き数として適当でない値に対して generic functionは nil を返すことにする。

generic functionはプログラムを書く上で非常に有用な機能である。しかし、型の間に継承がある場合に、型の継承関係と無関係に関数が定義可能であっては、プログラムの動作を追うのが困難となることがある。実際、継承関係にある型に対して無関係な動作を割り当てて作られたgeneric functionに、一つの関数として意味を与えるのは難しい。そこで、I-domain間のgeneric functionとして、型変換を保つもののみを考える事にする。すなわち、同じ型に属する値に対する像は同じ型に属し、二つの値が型変換可能な時にそれらに関数を適用した結果も型変換可能であるとする。 ここから以降では、 単に generic function と呼べば、 I-domain 間の generic functionでこの性質を満たすものを指すものとする。

generic functionに関して、カテゴリを用いて自然変換によって意味を与える研究がなされている[Reynolds'81]。そこでは、generic function はオペレータとして値以外のものとして扱われている。ここで構成するモデル (D, T,τ) は、D から D への generic function がまた D の要素であり、他の generic function の引き数となり得るものである。 そのためには、generic function に型が与えられなくてはならない。Typed lambda calculus のモデルでは、関数の型は、引き数の型と結果の型の組で与える。generic function の型は型の単純なペアではなく、それぞれの型に対して、その型に属する引き数に対する結果の型を返す T から T への単調な関数として与えられる。

この論文では、まず、基本となる型のなすsemi-coherent な poset TB と、TB の型に属する値のなす semi-coherentな CPO DB と、DB から TB への値に型を対応させる関数 τBからなるI-domain MB = (DB, TB, τB) が与えられたして、意味領域 M = (D,T,τ) を構成する。D は DB およびD からD へのgeneric functionからなり、T は TB およびgeneric functionの型となるような T からT への関数からなる。すなわち、M は、次の等式を満たす。

$$
\begin{array}{ccccccc}
D & = & DB & + & [D \dashrightarrow D]_{T\tau} & & \\
\downarrow{\scriptstyle \tau} & & \downarrow{\scriptstyle \tau B} & & \downarrow{\scriptstyle \tau *} & & \\
T & = & TB & + & \langle T \dashrightarrow T\rangle^{\sim} & & (*)
\end{array}
$$

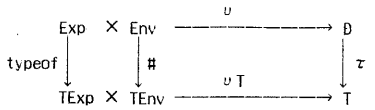
ここで、[D --> D]Tτは、D から D への T、τに関するgeneric function の集合であり、<T --> T>~ は、generic function の型となるような T から T への単調関数の集合である。τ*は、generic functionに型を対応させる [D-->D]Tτから<T --> T>~ への関数である。

モデルの構成、LTI、および、その意味論 --- 本論文の構成

(*)を満たす I-domain を構成するために、まず、I-domain全体がカテゴリをなしていることを述べる。このカテゴリの射は、値領域間のgeneric function と、その型となる型領域間の関数の組である。その後に、 [M --> M] = ([D --> D]Tτ, <T --> T>~, τ*)の三つ組が、I-domainをなしていることを証明し、[-->] を exponent としてI-domainのなすカテゴリがcartesian closed category となっていることを述べる。その後に、M = MB + [M --> M]という領域方程式 ---これは、(*)と等価である --- がこのカテゴリの中で解を持つことを証明する。

この方程式の解となる M を構成した後に、M の上で、言語 LTI (Language with Type Inheritance) に表示的意味を与える。LTI の構文は、通常の式の集合 Exp と、type-式の集合 TExpと、それぞれの式に対して対応する type-式 を返す Expから TExp への関数 typeof の三つ組からなる。これらはそれぞれ、意味を与えた時に、D, T, τ に対応する。Exp は定数の加えられたラムダ式であり、TExp は定数の加えられたラムダ式に、+ などの generic function の型を表現できるように拡張を行ったものである。

環境の集合を Env，型の環境の集合を TEnv とすると、LTIの意味は Exp に対する意味関数 υ:Exp --> (Env --> D)とTExp に対する意味関数 υT: TExp --> (TEnv --> T) の組で与えられる。((註) この表現は厳密には正しくない所がある。詳しくは本文を参照のこと。) υ と υT は、次の図式を可換にする。ここで、# は環境に対して対応する型の環境を返す関数である。



この論文の最後に、二つ以上のラムダ式を組み合わせて関数を表現する機能を付け加えた言語 LTI1 について述べる。これは、CLOS の method combination に対応する機能である。

2. 式

LTI の構文を定める。この言語は、通常の式の集合 Exp (expression)、type-式 の集合 TExp (type expression)、及び ExpからTExpへの関数 typeof からなる。

記号の集合を次のように定義する。

C :定数の集合
V :変数の集合
TC :型定数の集合
FTC:有限型定数の集合
TV :型変数の集合

ここで、変数と型変数は一対一に対応しており、その対応関係はV からTV への関数typeofv で与えられているとする。Cには特別な定数nile が含まれているとする。FTC はTCの部分集合であって、特別な型定数 nulle が含まれているとする。nile、nulleには、それぞれ値領域、型領域の最小限である nil、null の意味が与えられる。

最初に、式の集合 Exp を次のように定義する。

$c \in C$
$x \in V$
$e, e_1, e_2 \in Exp$

$e ::= \quad c \mid x \mid \lambda x.\, e \mid e_1\ e_2$

式及び type-式では、結合の順番をはっきりさせるために適当に () を使うことにする。

次に type-式の集合 TExp を定義する。 TExp を定義するために、open-set-式の集合 OExp、finite-type-式 の集合FTExp を同時に定義する。意味を与えた時に、OExpは型のなす順序集合の単項生成な閉集合を表わす。また、FTExp はOExpの表わす集合を生成する型を表わす。

$tc \in TC$
$tx \in TV$
$ftc \in FTC$
$te, te_1, te_2 \in TExp$
$o, A_i \in OExp\ (i = 0,1,\ldots,m)$
$fte, B_i \in FTExp\ (i = 0,1,\ldots,m)$

$te ::= \quad tc \mid tx \mid \Lambda tx{:}o.te \mid te_1\ te_2 \mid te_1 \vee te_2$

$fte ::= \quad ftc \mid (A_0 \text{-->} B_0, A_1 \text{-->} B_1, \ldots, A_m \text{-->} B_m)$

$o ::= \quad fte\uparrow$

$t_1 \vee t_2$ は、二つの型の上限となる型を表わすtype-式である。$\Lambda tx{:}o.te$ は関数の型を表わす type-式 で、引き数の型がo の表わしている集合に含まれる時、te の表わす型に属する値を返し、それ以外の引き数に対してはnile が表わしている値nil を返す関数の型を表わしている。$\Lambda tx{:}nulle\uparrow.te$ のことを簡単に $\Lambda tx.te$ と書くことにする。また、$(A_0 \text{-->} B_0, A_1 \text{-->} B_1, \ldots, A_m \text{-->} B_m)$ は、type-式

$\Lambda tx{:}A_1.B_1 \vee \Lambda tx{:}A_2.B_2 \vee \ldots \vee \Lambda tx{:}A_m.B_m$

の略である。これによって、FTExp は TExp の部分集合である。式、type-式に対して通常のラムダ計算の時と同様に閉じた式、閉じたtype-式が定義される。

最後に、typeofを定義する。 C から閉じた type-式の集合へ、それぞれの定数の型を対応させる関数 typeofc が与えられているとする。typeofcを拡張してExp から TExp への関数 typeof を定義する。

```
typeof(c) = typeofc(c)  where c ∈ C
typeof(x) = typeofv(x)  where x ∈ V
typeof(λx. e) = Λtypeofv(x). typeof(e)
typeof(e1 e2) = typeof(e1) typeof(e2)
```

type-式 typeof(e) のことを、式 e の型と呼ぶことにする。

式、及びtype-式の例を挙げる。LTI の定数を、

```
V  = (x)
C  = INTE + REALE + (nile) + (sqrte, trunce, +e)
TV = (tx)
TC = FTC =  (inte, reale, nulle)
```

とする。ただし、INTE は -2e, -1e, 0e, 1e, 2e,...といった整数を表わす記号の集合、 REALE は 1.0e, 2.0e,2.45e,... といった実数（の部分集合）を表わす記号の集合とする。数学的な概念と区別するために、構文領域の定数は数学的概念の名前に e を付けた名前を与えてある。

変数に対して typeofv、定数に対して typeofc が次のように与えられているとする。

```
typeofv(x) = tx

typeofc(e) = inte (e が INTE に属する時)
typeofc(e) = reale (e が REALE に属する時)
typeofc(sqrte) = (reale↑ --> reale)
typeofc(trunce) = (reale↑ --> inte)
typeofc(+e) = (inte↑ --> (inte↑ --> inte)
                reale↑ --> (reale↑ --> reale))
```

その時、次の式はそれぞれ Exp に属している。

```
((+e 2e) 3e)
λx. x
λx. (x x)
λx. ((+e (trunce (sqrte x))) x)
```

これらの式の型は次のようになる。

```
typeof((+e 2e) 3e)      = (('+e inte)inte)
typeof(λx. x)           = Λtx. tx
typeof(λx. (x x))       = Λtx. (tx tx)
typeof(λx. ((+e (trunce (sqrte x))) x)
       = Λtx. ((+e' (trunce' (sqrte' x))) x)
```

ここで、次の略記を用いた。

```
sqrte' = (reale↑ --> reale)
trunce' = (reale↑ --> inte)
+e' = (inte↑ --> (inte↑ --> inte)
       reale↑ --> (reale↑ --> reale))
```

3. I-domain

この章では、I-domainを定義し、I-domain および I-domainのなすカテゴリの性質を調べる。

3.1 I-domain の定義

I-domain (domain with type inheritance) は、継承関係を持った型、及び値の数学的モデルである。I-domain が型継承のモデルとして自然なものであることを示すために、型に関する直感により即した定義を最初に説明する。

I-domain は、型のなす集合 T と、値のなす集合 D 及びそれぞれの値に対してその属している型を対応させる D から T への全射 τ からなる。D を値領域、T を型領域、D の要素を値、Tの要素を型と呼ぶ。値 x と型 A が $\tau(x) = A$ を満たす時 x は A に属すると呼ぶ。型 A に対して A に属している値の集合を Val(A) と書くことにする。Val(A) は A の τ に関する逆像に等しい。

型の間には継承という関係が定義されており、型 A が型 B を継承している時に $A \geqq B$ と書くことにすると、$(T, \geqq)$ は半順序集合 (poset) となっているとする。つまり、A、B、C を型とした時、

$A \geqq A$
$A \geqq B$ かつ $B \geqq C$ ならば、$A \geqq C$
$A \geqq B$ かつ $B \geqq A$ ならば、$A = B$

が成り立つ。T はこの順序に関して最小限をもつとする。Tにはさらにsemi-coherentという条件を設ける。

DEFINITION　順序集合 O が semi-coherent であるとは、Oの中に上界を持つ全ての O の部分集合が O の中に上限を持つことである。

T が semi-coherent であることは、型の集合に対してそれに属する型を全て継承している型が存在する時には、そのような型の中で最も一般的な型が存在するということを意味している。

型の間に継承関係がある時、それらの型に属する値の集合の間で型変換(coercion) の関数が定義されているとする。$A \geqq B$ の時、Val(A)から Val(B) への型変換の関数を $\delta_{A,B}$ と書く。型変換の関数は、

$\delta_{A,A} = id_A$ (恒等関数)
$A \geqq B$ かつ $B \geqq C$ ならば、$\delta_{B,C} \cdot \delta_{A,B} = \delta_{A,C}$　(*)

を満たすとする。この時、D の上に関係 $>$ を、

$A = \tau(a)$、$B = \tau(b)$ とした時、
$A \geqq B$ かつ $\delta_{A,B}(a) = b$ ならば $a > b$

と定義すると、$(D, >)$ は順序集合となる。さらに、$(D, >)$ は、semi-coherent な cpo であるという条件を設ける。

以上の性質を持つ (D, T, τ) は、次の性質を満たす。

$\tau(a) = A$、$A \geqq B$ ならば、B に属し $a > b$ となる値 b が唯一つ存在する。　(**)

ここまでは、型 A、B に対して $\delta_{A,B}$ が与えられているとして、それを基に D の順序を定義したが、逆に、Dを semi-coherent な cpo、T をsemi-coherentな poset、τ をD からTへの単調な全射で(**)を満たすものとする。その時、(**)でaに対して唯一存在するb を $\delta_{A,B}(a)$ と定義すると、δは(*)を満たすことがわかる。よって、次のようにI-domain を定義する。

DEFINITION　$(D, >)$ を semi-coherent な cpo 、$(T, \geqq)$を semi-coherent な順序集合、τ を D から T への単調な全射で (**) を満たしているものとする。その時、(D, T, τ) の組をI-domain (Domain with Type Inheritance) と呼ぶ。(**)の b を $\delta_{A,B}(a)$ と書き、a の型 B への型変換 (coercion) と呼ぶ。

$\delta_{A,B}(a)$ は、$a\!\upharpoonright_B$ と書くこともある。

PROPOSITION 1
(D, T, τ) を I-domain とした時次のことが成り立つ。
1) D の最小元 nilに対し null = τ(nil) とおくと、null はT の最小元である。
2) $\tau(a) = \tau(b) = A$ かつ $a > b$ ならば、$a = b$
3) U を T の任意の部分集合とする。U に上限が存在し、$a \in D$ に対して $\tau(a) = A = \bigvee t$ ならば、$a = \bigvee a\!\upharpoonright_t$である。
$(t \in U)$　　$(t \in U)$
4) τは存在する全ての上限を保存する。すなわち、X を D の部分集合で $\bigvee X$ が存在するとした時、$\bigvee \tau(X)$ が存在し $\tau(\bigvee X) = \bigvee \tau(X)$ が成り立つ。

ここでは、紙面の都合上、証明は基本として省略する。

順序集合における部分集合 U の下限は、もし存在すれば $\bigvee\{y \mid y < \forall u \in U\}$に等しい。 semi-coherent な順序集合においては、空集合以外の U に対して$\{y \mid y < \forall u \in U\}$は Uの任意の元を上界とする。よって上限が存在し、semi-coherent な順序集合で、空集合以外の部分集合は下限を持つことがわかる。集合の下限を $\bigwedge$ で表わすことにする。

3．2　generic function

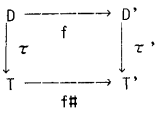
DEFINITION　(D, T, τ)、(D', T', τ')を I-domain とする。その時、D から D'への T、τ、T'、τ'に関する generic function とは、D から D' への連続関数で型を保存するものである。ここで、型を保存するとは f を generic functionとした時、Dの要素 x、y に対し、$\tau(x) = \tau(y)$ ならば $\tau'(f(x)) = \tau'(f(y))$ が成り立つことである。

定義より、f が generic function の時、f は同じ型に属する引き数に対しては同じ型に属する値を返す。また、f は連続より特に単調である。よって、型$A \geqq$型B の時、Val(A)$\ni$xに対し、$f(\delta_{A,B}(x)) = \delta_{A',B'}(f(x))$である。(ここで、$A' = \tau'(f(x))$、$B' = \tau'(f(\delta_{A,B}(x)))$である。) 即ち、fは値の間の型変換を保つ。これらの性質は、序章で generic function の性質として述べたものである。

$M = (D, T, \tau)$、$M' = (D', T', \tau')$をI-domain、f を D から D' への T、τ、T'、τ'に関するgeneric function とした時、T から T' へ関数 f# が

$f\#(t) = \tau'(f(x))$ where $\tau(x) = t$

と定義される。ここで、τ が全射であることよりこのような x は存在し、 fが generic function であることより f#(t) の値は x の取り方によらない。f# は、次の図式を可換とする。



逆に、容易に分かるように、generic function f に対してこの図式を可換とする T から T' への関数は f# のみである。そこで、 I-domain 間の準同型を次のように定義する。

DEFINITION $M = (D, T, \tau)$、$M' = (D', T', \tau')$ をI-domain とする。その時、M から M' への準同型とは、D から D' へのcpo 間の連続関数 ϕD とT から T' への関数 ϕT の組で、$\phi T \cdot \tau = \tau' \cdot \phi D$ を満たすものである。

f が generic function の時、(f, f#) は準同型である。また容易に分かるように、$(\phi D, \phi T)$が準同型の時、ϕDはgeneric function である。よって、これからは、generic function を考える代わりに準同型を考える事にする。準同型の性質として、次のことが成り立つ。

PROPOSITION 2
$(\phi D, \phi T)$ を $M = (D, T, \tau)$ から $M' = (D', T', \tau')$への準同型とする。
1) ϕT は有向集合の極限を保つ。すなわち、U が T の有向集合で $\bigvee U$ を持つ時、$\bigvee \phi T(U)$ が存在して $\phi T(\bigvee U) = \bigvee \phi T(U)$ である。
2) $a \in D$ 、$u \in T$、$\tau(a) > u$、$v = \phi T(u)$ とする。$\phi D(a\!\upharpoonright_u) = \phi D(a)\!\upharpoonright_v$ である。

PROPOSITION 3
I-domain 全体は、準同型の結合を$(\phi D, \phi T) \cdot (\phi D', \phi T') = (\phi D \cdot \phi D', \phi T \cdot \phi T')$と定義し、$M = (D, T, \tau)$に対して M から M への恒等射 id_M を (id_D, id_T) と定義することにより、準同型を射としてカテゴリーをなしている。このカテゴリを DTI (category of Domains with Type Inheritance) と呼ぶことにする。

DEFINITION 準同型 $(\phi D, \phi T)$ で、ϕD がボトムを保存するものを I-domain 間の自然な準同型と呼ぶ。

$(\phi D, \phi T)$ が自然な時、ϕT もボトムを保存する。

DEFINITION 準同型 $(\phi D, \phi T)$ で、ϕD が存在する全ての上限を保存するものを I-domain間の加法的準同型と呼ぶ。

$(\phi D, \phi T)$ が加法的な時、ϕT も存在する全ての上限を保存する。また、加法的準同型は自然である。

3．3 DTI の性質

ここでは、DTI が cartesian closed category であることを証明する。まず、最初に、DTI における和と積について述べる。

M = (D, T, τ)、M' = (D',T',τ') を I-domain とした時、これらの積 M × M'を M × M' = (D × D',T ×T', $\tau \times \tau$')と定義する。ここで、D × D'、T × T'はcpo,posetの積、$\tau \times \tau$' はposet 間の関数の積である。M×M' はI-domainとなり、Dp1: D ×D' --> D、Tp1: T×T' --> T を cpo 、poset における射影とすると、(Dp1,Tp1): (M × M' --> M)が積から成分への射影となる。M× M' --> M' の射影も同様である。この積は、DTI のカテゴリとしての積と一致する。

また、I-domain の和 M + M' を M + M' = (D + D',T + T', τ + τ')と定義する。ここで、D + D'、T + T'は、cpo、poset のボトムを同一視した和で、τ + τ' はposet 間の関数の和である。(τ、τ'はボトムを保つので和が定義される。) M + M' は I-domain となる。 f_1: M_1 --> M_1'、f_2: M_2 --> M_2'が自然な関数の時 $f_1 + f_2$: $M_1 + M_2$ --> $M_1' + M_2'$ が定義される。この和は、射として自然な準同型のみを取る DTI の部分カテゴリを考えた時、そのカテゴリの和と一致する。

$\perp_D$、$\perp_T$ を一つの要素からなる poset、$\perp_\tau$ をその間に一つだけ存在する関数とする。($\perp_D$,$\perp_T$,$\perp_\tau$) はDTI の終対象となっている。この I-domain を $\perp_{DTI}$ と書くことにする。任意のI-domain M から $\perp_{DTI}$ へ存在している唯一の準同型を $\perp_M$ と書くことにする。

次に、 I-domain 間の準同型全体が I-domain をなすことを証明する。M = (D, T, τ)、M' = (D',T',τ') をI-domain とする。D から D' への T、τ、T'、τ' に関するgeneric functionのなす集合を $[D\text{-->}D']_{T\tau T'\tau'}$ と書くことにする。特に、D = D', T = T', τ = τ' の時には$[D\text{-->}D]_{T\tau}$と書くことにする。$[D\text{-->}D']_{T\tau T'\tau'}$ の上に順序 $\lhd$ を

$$f \lhd g \Leftrightarrow f(x) \lhd g(x)\ (\forall x \in D)$$

と定義する。$\lhd$ は順序の公理を満たしている。また、M から M' への準同型の上に順序を、 D 成分の順序によって定義する。

PROPOSITION 4
$[D\text{-->}D']_{T\tau T'\tau'}$ は cpo をなす。

<T-->T'> を T から T' への単調関数全体のなす posetとする。$f \in [D\text{-->}D']_{T\tau T'\tau'}$ の時、f を $f^\sharp$ に対応させる$[D\text{-->}D']_{T\tau T'\tau'}$から <T-->T'> への関数が存在する。この関数を $\tau\tau'^*$ とおく。特に、τ = τ'の時には τ^* と書くことにする。$\tau\tau'^*$ は単調関数であるが一般に全射ではない。全射にならない例は、紙面の関係上割愛する。$\tau\tau'^*$の像を$\langle T\text{-->}T'\rangle^\sim$とおく。

LEMMA
$f \in [D\text{-->}D']_{T\tau T'\tau'}$、$u \in \langle T\text{-->}T'\rangle^\sim$、$\tau\tau'^*(f) \geq u$ とする。この時、$f\natural_u$ を、$f\natural_u(x) = f(x)\natural_{u(\tau(x))}$と定義すると、$f\natural_u$ は generic function である。

PROPOSITION 5
$\langle T\text{-->}T'\rangle^\sim$ は semi-coherent である。

$\langle T\text{-->}T'\rangle^\sim$ は、T、T' が cpo であっても一般には cpo になるとは限らない。cpo にならない例は割愛する。

THEOREM M = (D, T, τ)、M' = (D',T',τ') をI-domain とした時、$([D\text{-->}D']_{T\tau T'\tau'}, \langle T\text{-->}T'\rangle^\sim, \tau\tau'^*)$はI-domainをなしている。これを、[M-->M'] と書くことにする。

THEOREM 2
DTI は、cartesian closed category である。

DTI は、concrete category である。すなわち、集合としての構造を持つ。最後にそのことについて述べる。

DEFINITION カテゴリ C で、終対象から対象 A への射を$\natural_A$ の点と呼び、A の点の集合を $\natural A\natural_C$ (或いは簡単に $\natural A\natural$) と書く。全ての f, g ∈ Hom(A, B) に対し、

$$f \neq g \Rightarrow \exists x \in \natural A\natural,\ f \cdot x \neq g \cdot x$$

が成り立つ時、A は十分点を持つと呼ぶ。

M = (D, T, τ) に対して $\natural M\natural_{DTI} = \natural D\natural_{CPO}$ である。Dのボトムに対応するM の点を $\perp'_M$ と書くことにする。DTI において、任意の I-domain は十分点を持つ。DTI からCPO への関手 F を

$$F(M) = D$$

$$f = (fD, fT) \in \mathrm{Hom}(M_1, M_2) \text{ に対して、} F(f) = fD$$

とおくと、F は忠実な関手となっている。F は、積や ev なども保存する。CPOは concrete category なので、DTIはconcrete category であることがわかる。

3．4 DTI における帰納的極限

ここでは、DTI において、帰納系が limit を持つことを述べる。

PROPOSITION 6
$\Delta_R = (M_i, f_i)$ $(M_i = (D_i, T_i, \tau_i), f_i = (fD_i, fT_i) : M_{i+1} \text{-->} M_i)$ を帰納系とする。その時、Δ_R の帰納的極限 (M, g_i)が存在する。

DTI から CPO への関手 F は、帰納的極限も保存する。

DEFINITION M_1、M_2 を I-domain、$f: M_1 \text{-->} M_2$、$g: M_2 \text{-->} M_1$を準同型とする。$g \cdot f = id_{M1}$ かつ $f \cdot g \lhd id_{M2}$の時、(f,g) を M_1 から M_2 への projective pair、fをembedding、g を projection と呼ぶ。(f, g) がprojective pairの時、f、g は自然である。特に f は加法的である。g も加法的な時、(f, g)を加法的なprojective pair と呼ぶ。

上の定義の中で、f、g が自然で f が加法的であることは、f、g の D 成分がcpo の projective pair をなすことより明らかである。

f が embedding の時、対応する projection を f_R と書くことにする。DTI_E を、射として加法的な embedding のみをとる DTI の部分カテゴリとする。

PROPOSITION 7
$\Delta = (M_i, f_i)$ を、DTI_E における射影系とする。その時、$\Delta_R = (M_i, f_{iR})$はDTI における帰納系となる。その帰納的極限を (M, g_i) とすると、g_i はprojectionであり、対応する embedding を g_{iE} とすると、(M, g_{iE})はDTI_Eにおける Δ の射影的極限となる。

3．5 有限な要素、開集合

最後に、順序集合に対して有限な要素、及び開集合という概念を導入する。これらは、LTI の意味論を考える時に必要となる。

DEFINITION 順序集合 (T, >) の要素 t が有限であるとは、上限が存在する任意の T の有向集合 U に対して、$t \lhd \bigvee U$ ならば、ある u ∈ U に対して $t \lhd u$ となることである。(T, >)の有限な要素全体の集合を F(T) と書く。

DEFINITION (T, >) について、T の部分集合 V が開集合であるとは、次の (1)、(2) を満たすことである。
(1) V ∋ tならば、u > t となる全ての u についてV ∋u 。

(2) 上限が存在する任意の T の有向集合 U に対して、$\vee U \in V$ ならば、ある $u \in U$ に対して $u \in V$。
(T, >) の開集合全体の集合を、O(T) と書く。

PROPOSITION 8
(T, >) を順序集合とする。
1) $t_1, t_2 \in F(T)$ かつ $t_1 \vee t_2$ が存在するならば、$t_1 \vee t_2 \in F(T)$ である。
2) $f \in F(T)$ の時、$\{x \in T \mid x > f\}$ は (T, >) の開集合をなす。この開集合を、$f\triangle$ と書くことにする。

DEFINITION O(T) のなかで、F(T) に属する f に対して$f\triangle$ と書き表わされるものを、f によって定められる単項生成開集合と呼ぶ。T の中で、単項生成開集合及び空集合のなす集合を P(T) と書く。

PROPOSITION 9
(T, >)、(T', >) を順序集合とする。$O \in P(T)$、$F \in F(T')$ とする。その時、Tから T'への関数 $\{O \rightarrow F\}$ を、

$\{O \rightarrow F\}$ = if (t ∈ O) then F else null
(where null は T' のボトム)

と定義する。これは $\langle T \rightarrow T' \rangle$ の有限な要素となる。

PROPOSITION 10
M = (D, T, τ)、M' = (D',T',τ') を I-domain とする。nil、null をそれぞれ D'、T' のボトムとする。
(1) $A \in F(T)$、$B \in T'$、f: Val(A) --> Val(B) を(集合間の)関数とする。その時、$fD\blacktriangle : D \rightarrow D'$、$fT\blacktriangle : T \rightarrow T'$ を、

$fD\blacktriangle(x)$ = if ($\tau(x) \cong A$) then $f(x|_A)$ else nil
$fT\blacktriangle(t)$ = if ($t \cong A$) then B else null

とおく。$f\blacktriangle = (fD\blacktriangle, fT\blacktriangle)$ は M から M' への準同型となる。
(2) f = (fD, fT) を M から M' への準同型、$O \in O(T)$ とする。

$fD!O(x)$ = if ($\tau(x) \in O$) then fD(x) else nil
$fT!O(t)$ = if ($t \in O$) then fT(t) else null

とおくと、$f!O = (fD!O, fT!O)$ は M から M' への準同型となる。

M =(D,T,τ)、M' =(D',T',τ')を I-domain とし、$O \in P(T)$、$F \in F(T')$ とする。$\tau'(z) = F$ となる $z \in D'$ を一つ定め、fD, fT それぞれを恒等関数 lambda(x).z、lambda(t).F とおくと、(fD, fT) は Mから M'への準同型なので、PROPOSITION 10 より f!O は、M から M'への準同型となる。この時、fT!O は $\{O \rightarrow F\}$ と一致するので、$\{O \rightarrow F\}$ は$\langle T \rightarrow T \rangle^\sim$ に含まれることがわかる。

4. reflective な I-domain

この章ではは、I-domain MB = (DB, TB, τB) が与えられたとして

$$M = MB + [M \rightarrow M]$$

を満たす I-domain M を構成する。[Smyth, Plotokin] は、一般にカテゴリの中でこのような等式が解けるための条件を与えている。DTI はその中でで述べられている O-categoryであり、それらの条件を満たしている。ここではその証明を示すのではなく、直接この等式を満たす M を構成する。なお、[Smyth, Plotokin]に従えば、この他に積などを加えた方程式もDTI において解を持つことがわかる。このことは、LTI を拡張して様々なデータ型を持つ言語とした時に、その意味論を与える領域を構成するのに必要となる。

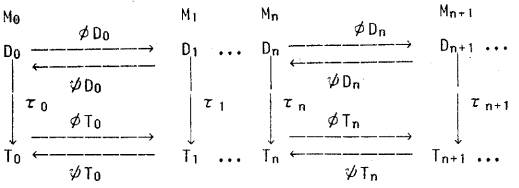
I-domain $M_n = (D_n, T_n, \tau_n)$ (n = 0,1,2,...) 及び $MF_n = (DF_n, TF_n, \tau F_n)$ (n = 1,2,...) を、

$M_0 = \perp_{DTI}$

$MF_{n+1} = [M_n \rightarrow M_n]$,
$M_{n+1} = MB + MF_{n+1}$ (n = 0,1,2,...)

と定義する。

M_nから M_{n+1}への加法的な projective pair ($\phi_n = (\phi D_n, \phi T_n)$, $\psi_n = (\psi D_n, \psi T_n)$) (n = 0,1,2,...)を順に構成する。



n = 0 の時

$\phi_0 = \perp'_{M1}$: $\perp_{DTI} \rightarrow M_1$
$\psi_0 = \perp_{M1}$: $M_1 \rightarrow \perp_{DTI}$

(ϕ_0, ψ_0) は、M_0 から M_1 への加法的な projective pairである。

n > 0 の時

ϕDF_n: $DF_n \rightarrow DF_{n+1}$、$\psi DF_n$: $DF_{n+1} \rightarrow DF_n$、
ϕTF_n: $TF_n \rightarrow TF_{n+1}$、$\psi TF_n$: $TF_{n+1} \rightarrow TF_n$
(n = 1,2,...) を、

$\phi DF_n(x:DF_n) = \phi D_{n-1} \cdot x \cdot \psi D_{n-1}$
$\phi TF_n(t:TF_n) = \phi T_{n-1} \cdot t \cdot \psi T_{n-1}$
$\psi DF_n(f:DF_{n+1}) = \psi D_{n-1} \cdot f \cdot \phi D_{n-1}$
$\psi TF_n(u:TF_{n+1}) = \psi T_{n-1} \cdot u \cdot \phi T_{n-1}$

と定義する。

PROPOSITION 11
1)$\phi F_n = (\phi DF_n, \phi TF_n)$ (n = 1,2,...) は MF_n からMF_{n+1} への準同型である。
2)$\psi F_n = (\psi DF_n, \psi TF_n)$ (n = 1,2,...) は MF_{n+1}から MF_n への準同型である。
3)($\phi F_n, \psi F_n$) (n = 1,2,...) は、MF_n からMF_{n+1}への加法的な projective pair である。ϕF_n、ψF_n、は自然なので、和をとることができる。$\phi_n = id_{MB} + \phi F_n$, $\psi_n = id_{MB} + \psi F_n$ とおく。
4)ϕ_n (n = 1,2,...) は M_n からM_{n+1}への準同型である。
5)ψ_n (n = 1,2,...) は M_{n+1}からM_n への準同型である。
6)(ϕ_n, ψ_n) (n = 1,2,...) は、M_nから M_{n+1} への加法的な projective pair である。

(proof) 1) から6) までを同時に帰納法によって証明する。1)から6) までが、n-1 の時に成り立っているとする。
1)ϕDF_n が連続であることは容易にわかる。
$\tau F_{n+1} \cdot \phi DF_n = \phi TF_n \cdot \tau F_n$
を証明すればよい。
$d \in DF_n$ 、$t \in T_n$ とする。

$(\tau F_{n+1} \cdot \phi DF_n)(d)(t)$
$= \tau_n(\phi DF_n(d)(x))$
where $x \in D_n$、$\tau_n(x) = t$;;τF_{n+1} の定義
$= \tau_n(\phi D_{n-1} \cdot d \cdot \psi D_{n-1}(x))$;;ϕDF_n の定義
$= (\phi T_{n-1} \cdot \tau_{n-1} \cdot d \cdot \psi D_{n-1})(x)$;;帰納法の仮定

$(\phi TF_n \cdot \tau F_n)(d)(t)$
$= (\phi T_{n-1} \cdot \tau F_n(d) \cdot \psi T_{n-1})(t)$;;ϕTF_n の定義
$= \phi T_{n-1} \cdot \tau F_n(d) \cdot \tau_{n-1}(\psi D_{n-1}(x))$
where $x \in D_n$、$\tau_n(x) = t$;;帰納法の仮定
$= \phi T_{n-1} \cdot \tau_{n-1}(d(\psi D_{n-1}(x)))$;;τF_n の定義

2)から 6) の証明は省略する。

(M_n, ϕ_n) (n = 0,1,2,...) はDTI$_E$ において射影系をなしている。この射影的極限を$(M = (D,T^\sim,\tau), i_n=(iD_n,iT_n))$とおく。$i_n$と対応する projectionを$j_n = (jD_n,jT_n)$とおく。また、($MF_n$, ϕF_n) (n = 1,2,...)も射影系をなしている。この射影的極限を $(MF = (DF,TF^\sim,\tau F), i_n' = (iD_n',iT_n'))$とおく。$i_n'$に対応するprojection を、$j_n' =(jD_n',jT_n')$とおく。

DTI$_E$ において射影的極限は + を保存するので、次の proposition が成り立つ。

PROPOSITION 12
M = MB + MF である。

次に、ΦF = (ΦDF,ΦTF): MF --> [M-->M]、
ΨF = (ΨDF,ΨTF): [M-->M] --> MF を

$$\Phi DF(d:DF) = \bigvee_n iD_n \cdot jD_{n+1}'(d) \cdot jD_n$$

$$\Phi TF(t:TF) = \bigvee_n iT_n \cdot jT_{n+1}'(t) \cdot jT_n$$

$$\Psi DF(f:[D\text{-->}D]_{T\tau}) = \langle y_1,\ldots,y_n,\ldots\rangle \in DF \quad \text{where } y_{n+1} = jD_n \cdot f \cdot iD_n$$

$$\Psi TF(v:\langle T\text{-->}T\rangle^{\sim}) = \langle u_1,\ldots,u_n,\ldots\rangle \in TF \quad \text{where } u_{n+1} = jT_n \cdot v \cdot iT_n$$

と定義する。

PROPOSITION 13
1) ΦF は準同型である。
2) ΨF は準同型である。
3) (ΦF,ΨF) は加法的な projective pair である。
さらに、ΨF、ΦF は、
$\Psi F \cdot \Phi F = id_{MF}$
$\Phi F \cdot \Psi F = id_{[M\text{-->}M]}$
を満たす。

$\Phi TF \cdot \Psi TF = id_{\langle T\text{-->}T\rangle^{\sim}}$ となることは一見直感に反するようにに思えるが、定義域が <T-->T>の中で τ*の像に制限されているために成り立つ。
ΦF、ΨFは自然なので、id_{MB}と和をとることができる。

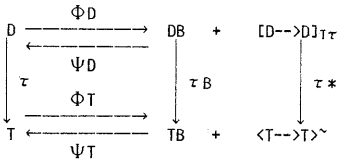
$$\Phi = id_{MB} + \Phi F$$
$$\Psi = id_{MB} + \Psi F$$

とおく。Φ、Ψ は、次の I-domain 間の準同型である。

$$\Phi = (\phi D, \phi T): M \text{ --> } MB + [M\text{-->}M]$$
$$\Psi = (\Psi D, \Psi T): MB + [M\text{-->}M] \text{ --> } M$$

THEOREM 3
Φ、Ψは加法的で、
$\Phi \cdot \Psi = id_{(MB+[M\text{-->}M])}$
$\Psi \cdot \Phi = id_M$
をみたす。

Φ と Ψ の関係を (D, T, τ) の関係で表すと、下図のようになる。



$\tau = \tau B + \tau *$

5. 意味論

基本となる値の集合 DB、基本となる型の集合 TB 及び、基本となる値に対しその型を返す関数 τB からなるI-domain MB = (DB, TB, τB) が与えられたとする。WRONG を、型領域が (TWrong,null)からなる poset, 値領域が (DWrong,nil) からなる poset である I-domain として、

$$M = MB + WRONG + [M \text{ --> } M]$$

を満たす I-domain M = (D、T、τ) が前章の方法で構成される。この章では、LTI の定数の集合 C から D へ定数に対する意味関数 υC 及び、TC から T へ型定数に対する意味関数 υTC が与えられたとして、M の上で LTI の表示的意味を与える。

((註) 6 章　　でとりあげる例では、υTCの像は TB に含まれている。) LTI の意味関数は、式に対する意味関数 υ と、type-式に対する意味関数 υT の組からなる。

TWrong, DWrong は、TB, DB の元を関数として実行しようとした時に返される型、及び値である。一方、二つの type-式の表わす型の上限を表わすtype-式($t_1 \vee t_2$)が LTI には存在するが、これは、上限が存在しない二つの型に対しては意味を与えられない。その時には、υT は、Errorを返すものとする。Error とTWrongの違いに注意されたい。

変数の集合 V から D への割り付け(環境)の集合を Env、型変数の集合 TV から T への割り付け(型環境)の集合をTEnv とする。υ、υTは次の値域を持つ。

```
υ  :Exp  --> (Env --> D)
υT :TExp --> (TEnv --> T + (Error))
```

以下で意味関数を定義するのに用いる記法を説明する。意味関数の引き数は ()ではなくて [] で書き表わす。 D のボトムをnil、 T のボトムを null とおく。[M --> M] を MFとおき、MF = (DF, TF, τF) とする。

まず、式に対する意味関数 υ を定義する。定数に対する意味関数υC: C --> Dが与えられているとする。その時、υ: Exp --> (Env --> D) を、式の構成に合わせて以下のように定義する。

$$\upsilon[c](\rho) = \upsilon C[c]$$

$$\upsilon[v](\rho) = \rho(v)$$

$$\upsilon[e_1\ e_2](\rho) = \text{if } (\Phi D(\upsilon[e_1](\rho)) \in DF) \text{ then } \Phi D(\upsilon[e_1](\rho))(\upsilon[e_2](\rho)) \text{ else DWrong}$$

$$\upsilon[\lambda x.e](\rho) = \Psi D(f) \quad \text{where } f = \text{lambda(d).}\upsilon[e](\rho[d/x])$$

ただし、$\rho[d/x]$ は、環境 ρ の x に対する値を d に置き換えて得られる環境である。これが定義されるためには、fが DF = $[D\text{-->}D]_{T\tau}$ に属してなくてはならない。

LEMMA
lambda(d).υ[e](ρ[d/x]) は、D から D への generic function となっている。

次に、type-式に対する意味関数 υT を定義する。型定数に対する意味関数υTC: TC --> T で FTC の要素に対してはT の有限な要素(F(T) に属する要素)を返すものが与えられているとする。υT を定義する前に、FTExp の意味関数 υF、OExp の意味関数 υO を定義する。

```
υF: FTExp --> F(T) + (Error)
υO: OExp --> P(T)
```

である。

$$\upsilon F[ftc] = \upsilon TC[ftc] \quad \text{(where ftc: 有限型定数)}$$

$$\upsilon F[(A_0\text{-->}B_0, A_1\text{-->}B_1, \ldots, A_m\text{-->}B_m)] = \text{if } (\exists i.\ \upsilon F[B_i] = \text{Error}) \text{ then Error else if } (\bigvee_i \upsilon F(\{\upsilon O[A_i] \rightarrow \upsilon F[B_i]\}) \text{ exists}) \text{ then } \bigvee_i \upsilon F(\{\upsilon O[A_i] \rightarrow \upsilon F[B_i]\}) \text{ else Error}$$
$$\text{where } A_i \in OExp,\ B_i \in FTExp$$

$$\upsilon O[fte\uparrow] = \text{if } (\upsilon F[fte] = \text{Error}) \text{ then 空集合 else } \upsilon F[fte]^{\triangle}$$

type-式に対する意味関数 υT: TExp --> (TEnv --> T + (Error)) を、type-式の構成に合わせて定義する。

$$\upsilon T[tc](\rho t) = \upsilon TC[tc]$$

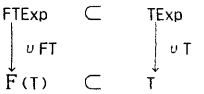
$$\upsilon T[tv](\rho t) = \rho t(tv)$$

```
υT[t1 t2](ρt) =
      if( υT[t1](ρt) = Error or υT[t2](ρt) =Error)
            then Error
      else if (ΦT(υT[t1](ρt)) ∈ TF )
            then ΦT(υT[t1](ρt))(υT[t2](ρt))
      else TWrong

υT[Λtx:o.te](ρt) =
      if ( υT[te](ρt[u/tx]) = Error for ∃u ∈ T )
            then Error
      else let f = (lambda (u). υT[te](ρt[u/tx]))
      if (f ! υO[o] ∈ TF )  then ΨT(f ! υO[o])
      else Error

υT[t1 ∨ t2](ρt) =
      if( υT[t1](ρt) = Error or υT[t2](ρt) =Error)
            then Error
      else if ( υT[t1](ρt) ⋁ υT[t2](ρt) exists)
            then υT[t1](ρt) ⋁ υT[t2](ρt)
      else Error
```

PROPOSITION 14
υF と υT は、FTExp から TExp、F(T) から T への埋め込みに対して可換である。



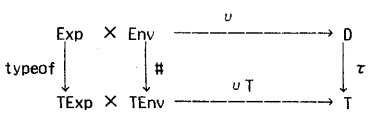
環境 ρ が与えられたとする。ρ に対して型環境 ρ# を、

```
ρ#(tx) = τ(ρ(x))   where typeofv(x) = tx
```

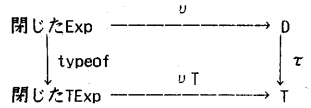
とおく。
type-式の意味は、T の要素とはならずに Error となることがあったが、そのtype-式 を型とする式が存在する時には、type-式 の意味は Error にならないことが次の定理でわかる。

THEOREM 4
定数に対して、υT[typeofc(c)] = τ(υC[c]) が成り立っているとする。その時、任意の式 e に対し、
υT[typeof(e)](ρ#) = τ(υ[e](ρ))
である。特に、υT[typeof(e)](ρ#) はError にはならない。

この定理は、υC、υTC が υT・typeofc = τ・υC を満たすように与えられた時、(υ、υT) が次の図式を可換にすることを意味している。



特に、閉じた式 に対しては、次の図式が可換となる。



この図式は、(υ、υT) が LTI の意味を与えていることを意味している。

6. 例

3章で与えた式の例に対して、式の意味及び対応するtype-式の意味を考える。LTI の記号及びその型は、3 章で与えたとおりである。
意味領域 MB = (DB, TB, τB) を

```
¦TB¦ = (int, real, null)
       int ≧ real、real ≧ null

¦DB¦ = INT + REAL + (nil)
       1 > 1.0、2 > 2.0、...、
       x > nil  where x ∈ REAL

τB(x) = int  where x ∈ INT
τB(x) = real  where x ∈ REAL
τB(nil) = null
```

と定義する。ここで INT は整数のなす集合、REAL は実数のなす集合である。 MB が I-domain となることは容易に確かめられる。
MB = (DB, TB, τB) に対して 4章の方法で M = (D, T, τ)が構成されたとする。ここでは、記述を簡略化するために、ΦD、ΨD を通じて DB の元を D の元と同一視する。また、ΦT、ΨT を通じて TB の元を T の元と同一視する。
υC: C --> D を、INTE に対してはその記号が表わしている整数を返し、REALE に対してはその記号が表わしている実数を返し、それ以外の定数に対しては、次のように定義されている関数とする。

```
υC[nile] = nil υC[sqrte] = ΨD(sqrt▲)
   where sqrt: REAL --> REAL, 平方根を計算する関数
υC[trunce] = ΨD(trunc▲)
   where trunc: REAL --> INT, truncateを計算する関数
υC[+e] = ΨD(f ⋁ g)  where
    f = lambda(x). if (τ(x) = int )
          then ΨD(lambda(y). if (τ(y) = int )
                                then x +i y  else nil)
           else nil
    g = lambda(x). if (τ(x) ≧ real )
          then ΨD(lambda(y). if (τ(y) ≧ real )
                   then x↑real +r y↑real else nil)
          else nil
```

ここで、 +r は実数と実数の足し算をして結果の実数をREALとして埋め込まれた D の要素として返す関数であり、 +i は整数と整数の足し算をして結果の整数を INT として埋め込まれた D の要素として返す関数である。
また、 υTC: TC --> T を

```
υTC[inte] = int
υTC[reale] = real
υTC[nulle] = null
```

と定義する。υC、υTC を基に作られた υ、υT は、全ての定数 c に対して υT[typeofc(c)] = τ(υC[c]) を満たしている。以降の例で、閉じた式、type-式に対しては意味関数の環境引き数を書かないことにする。

●例1 ((+e 2e) 3e)

```
    typeof((+e 2e)3e)= ((+e' inte)inte)
       where +e' = (inte↑ --> (inte↑ --> inte)
                    reale↑--> (reale↑--> reale))

υ[(+e 2e)] = ΦD(υ[+e])(υ[2e])
           = (f⋁g)(2) = f(2)⋁ g(2)
           = h1 ⋁ h2 where
      h1 = lambda(y). if (τ(y) = int )
             then 2 +i y        else nil
      h2 = lambda(y). if (τ(y) ≧ real )
             then 2.0 +r y↑real         else nil
υ[((+e 2e)3e)] = h1(3) ⋁ h2(3)
               = 5 ⋁ 5.0  =  5

υT[+e'] =   (int△ → (int△ → int) )
          ⋁ (real△ → (real△ → real) )
υT[(+e' inte)] = (int△ → int) ⋁ (real△ → real)
                    ;;int > int and int > real
υT[((+e' inte)inte)] = int ⋁ real  = int
```

ここで、ΦT を通してTF の元を T の元と同一視した。

●例2 λx.x

```
    typeof(λx.x) = Λtx.tx

υ[(λx.x)]   = ΦD( lambda(d).d)
```

```
υT[tx](ρt)  = ρt(tx)
υT[(Λtx.tx)]= if(υT[tx](ρt[tx/u]) = Error for ∃u)
              then Error
              else let f=(lambda(u). υT[te](ρt[tx/u]))
                     if ((f ! υO[nulle↑]) in TF )
                     then ΨT(f ! υO[nulle↑])
                     else Error
              = ΨT(f ! null△)  where f = lambda(u).u
              = ΨT(lambda(u).u)
```

●例3　λx. (x x)

```
typeof(λx. (x x)) = Λtx.(tx tx)

υ[(λx. (x x))] = ΦD(lambda(d). ΦD(d)(d))

υT[(Λtx. (tx tx))] = ΦT(lambda(u). ΦT(u)(u))
```

●例4　λx.((+e (trunce (sqrte x))) x)

```
typeof(λx.((+e (trunce (sqrte x))) x)
  = Λtx. ((+e' (trunce' (sqrte' tx))) tx)
```

ここで、

```
sqrte'  = (reale↑ --> reale)
trunce' = (reale↑ --> inte)
+e'     = (inte↑ --> (inte↑ --> inte)
          reale↑--> (reale↑--> reale))
```

である。ここでは、表記を簡単にするために、D と DF、T と TF を同一視する。

```
υ[(trunce (sqrte x))](ρ)    = trunc▲(sqrt▲(ρ(x)))
  ( if (τ(ρ(x)) > real )
    then  τ(trunc▲(sqrt▲(ρ(x))) = int
    else  τ(trunc▲(sqrt▲(ρ(x))) = null を用いて)
υ[(+e (trunce (sqrte x)))](ρ)
     = if (τ(ρ(x)) > real )
       then lambda(y). if (τ(y) = int )
                then trunc▲(sqrt▲(ρ(x))) +i y
                else nil
            ∨
            lambda(y). if (τ(y) ≧ real )
               then trunc▲(sqrt▲(ρ(x)))↓real +r y↓real
               else nil
       else nil
     = if (τ(ρ(x)) > real )
       then lambda(y).if (τ(y) = int )
                  then trunc▲(sqrt▲(ρ(x))) +i y
                  else if (τ(y) = real )
                  then trunc▲(sqrt▲(ρ(x)))↓real +r y
                  else nil
       else nil
.......
.......
υ[(λx.((+e(trunce (sqrte x)))x))] =
        lambda(d).if (τ(d) = int )
                  then trunc(sqrt(d↓real)) +i d
                  else if (τ(d) = real )
                  then trunc(sqrt(d))↓real +r d
                  else nil .

υT[sqrte'] =  (real△ → real)
υT[trunce'] = (real△ → int)
υT[+e'] =     (int△ →  (int△ → int)
          ∨ (real△ →  (real△ → real)

υT[tx](ρt)     = ρt(tx)
υT[(sqrte' tx)](ρt) = if (ρt(tx) > real )
                  then real    else null
.......
.......
υT[(Λtx.((+e' (trunce' (sqrte' tx))) tx))] =
                  lambda(t). if (t = int )   then int
                        else if (t = real ) then real
                        else null
```

この f は、(int△ → int) ∨ (real△ → real) に等しい。よって、 υT[Λtx. ((+e' (trunce' (sqrte' tx))) tx)] は、υT[(inte↑--> inte) ∨ (reale↑--> reale)] に等しいことがわかる。(Λtx.((+e' (trunce' (sqrte' tx))) tx))という type-式 を (inte↑--> inte) ∨ (reale↑--> reale)に変形するようなreduction をtype-式上に定義出来れば、generic functionに対する型チェックに応用できると考えられる。このことは、今後の課題である。

7.　method combination

LTI では、+r▲、+i▲ を意味とする式が存在する時に、それらから generic な + を意味とする関数を定義する方法がなかった。ここでは、幾つかの式の上限を取る式をLTI に追加することによって generic function を定義できるようにした言語 LTI1 を定義する。

まず、LTI1 の構文を定める。 LTI1 の構文は、LTI の構文の Exp の定義にe_1 & e_2 および λx:o.e を追加したものである。

```
c ∈ C
x ∈ V
e, e1, e2 ∈ Exp
o ∈ OExp

e ::=   c | x | λx:o. e | e1 e2 | e1 & e2  |
```

また、式の型の定義を次のように拡張する。

```
typeof(e1 & e2) = typeof(e1) ∨ typeof(e2)
typeof(λx:o.e) = Λ typeofv(x): o. typeof(e)
```

次に、LTI1 の意味を定義する。 type-式 の意味関数 υT は、LTI の時と同じである。 LTI では、全ての式が全ての環境において意味を持ったが、 LTI1 では、環境によって意味を与えることのできなくなる式が存在する。LTI1 の意味関数 υ は次の値域を持つ。

```
υ   :Exp  --> (Env --> D + (Error))
```

υ は、LTI の意味関数を次のように拡張したものである。

```
υ[c](ρ) = υC[c]
υ[v](ρ) = ρ(v)
υ[e1 e2](ρ) =
       if(υ[e1](ρ)=Error or υ[e2](ρ)=Error)
       then Error
       else  if (ΦD(υ[e1](ρ)) ∈ DF )
             then ΦD(υ[e1](ρ))(υ[e2](ρ))
             else DWrong
υ[λx:I.e](ρ) =
       if (υ[e](ρ[d/x]) = Error for ∃d ∈ D )
       then Error
       else (ΨD(lambda (d). υ[e](ρ[d/x])) ! υO[I])
υ[e1 & e2](ρ) =
       if (υ[e1](ρ) = Error or υ[e2](ρ) = Error )
       then Error
       else if (υ[e1](ρ) ∨ υ[e2](ρ) exists )
            then υ[e1](ρ) ∨ υ[e2](ρ)
            else Error
```

e_1 & e_2 は、υ[e_1]、υ[e_2] が関数を表わしている時、関数空間の上限の定義から、それぞれの関数を実行してその結果の上限を取るという semantics が与えている。これは、二つ以上の関数を合成して型変換を保つ関数を構成する一つの方法である。

THEOREM 5
e ∈ Exp に対し、υ[e](ρ) が Error でないならば、
υT[typeof(e)](ρ#)はError ではなく、τ(υ(e)(ρ))
と等しい。

ただし、この定理の逆は成り立たない。すなわち、
υT[typeof(e)] (ρ#) が値を持っても、υ[e](ρ) は Errorとなることがある。例えば、υ[1e & 2e] =υ[1e] ∨ υ[2e] = 1∨ 2 = Error であるが、υT[typeof(1e & 2e)] =υT[typeof(1e) ∨ typeof(2e)] = υT[typeof(1e)] ∨υT[typeof(2e)] = int ∨int = int である。これは、式に対する型の情報だけからでは、ある式の意味が Error になるかどうかは判断できないことを意味している。

8. 問題点、および将来の課題

●関数の連続性

I-domain (D,T,τ) において D が cpo であるという条件は、値のなす集合が型変換可能性に関して持つ性質として本質的ではなく、generic function がcpo 間の連続関数であるという条件も、複数の型の値に適用可能で型変換を保存する関数の性質として本質的ではないと思われる。D が cpo であり、generic function が cpo 間の連続関数であるという条件は、領域方程式を解いて reflective な領域を作るために付けたものである。連続でない関数も扱うことのできる領域を構築することは、将来の(非常に困難な)課題である。また、LTI では再帰的関数がうまく扱えない。Error, Wrong の取り扱いについても、改良の余地があると思われる。これも課題である。

●LTI における簡約規則

LTI は、構文と意味を与えただけで、計算という概念が定義されていなかった。式に対しては通常のlambda-caliculus と同様に、β-reduction を考えることができ、β-reduction が式の意味を変えないことも証明される。((註)LTI1 では、式の意味がError になるような式をβ-reduction すると Error でなくなることがあり、成り立たない。) type-式に対する reduction は、ラムダ式の引き数に型の制限が付いているので式の場合と同様にはいかないが、 reductionの体系を作ることは不可能ではないと思われる。type-式の reduction は、generic function の静的な型チェックなどに応用可能であると考えられ、reduction の体系を整えることは今後の最優先の課題である。

●LTI における型の拡張性

LTI では、型として関数型しか定義していなかった。LTIに積の型を加え、積を作る動作に関して閉じているI-domainを構成し、その上で意味を与えることは可能である。現在の所、オブジェクト指向の class に近いような型構造をLTI の構文に加え、それに対応した意味領域を構成してその上で意味を与えることを研究中である。そのような拡張を型構造に加えることにより型の間の継承関係が豊かになり、generic functionの有効性が発揮されると考えられる。とくに、LTI1で示したような generic function を定義する方法は、複雑な継承関係を持つ型構造の中で始めて実用的になると考えられる。また、LTI には副作用が存在しなかった。副作用を起こす関数の間の継承関係を考えるのは決して容易ではないが、オブジェクト指向言語など実際的な言語の意味を考える場合には必要である。

●型理論との関係

型理論では、ΣやΠといった依存型を取り扱う。Πは引き数の値によって結果の型が異なる関数の型であるのに対して、generic function は引き数型によって結果の型が異なる関数である。この両者には、類似点があるように思われる。また、Σに対応する概念がどういうものになるのか興味深い問題である。その他にも、再帰的に定義されるような無限型や、let 構成子の取り扱いなど、型の間に継承がある場合には困難とされている問題が色々とある [Reynolds 85]。この論文で与えた意味論は、それらの問題を解く基礎となる可能性があると思われる。

謝辞

この研究を行うに当たって適切な助言を与えていただいた京都大学数理解析研究所の中島玲二助教授、本研究の指導をして頂き多くの助言を頂いた同研究所の萩谷昌己先生に心から感謝します。

参考文献

H. Ait-Kaci and R. Nasr (1986), LOGIN: A logic programming language with built-in inheritance, in "J. Logic Programming, 3,(3),185-215"

H. Ait-Kaci (1986), An algebraic semantics approach to the effective resolution of type equations, in "Theoretical Computer Science, 45, 293-351"

H. P. Barendregt (1981), "The Lambda Calculus: Its Syntax and Semantics",North-Holland, Amsterdam.

D. G. Bobrow, K. Kahn and G. Kiczales (1986), Commonloops: Merging Common Lisp and Object-Oriented Programming, in "OOPSLA 86"

D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel,S. Keene, G. Kiczales and D. A. Moon (Feb. 1987) "Common Lisp Object System Specification",ANSI X3 Draft

J. P. Briot and P. Cointe, A Uniform Model for Object-Oriented Languages Using the Class Abstraction, "IJCAI'87"

L. Cardelli (1985), A Semantics of Multiple Inheritance, in "LNCS., vol. 173, 51-68"

L. Cardelli and P. Wegner (1985), On understanding Types, Data Abstraction, and Polymorphism, in "ACM Computing Survey, vol. 17, No. 4"

D. B. MacQueen, G. D. Plotokin and R. Sethi (1986), An Ideal Model for Recursive Polymorphic Types, in "Information and Control,71,95-130"

A. R. Meyer (1982), What is a Model of the Lambda Calculus?, in "Information and Control 32,87-122."

R. Milner (1978), A Theory of Type Polymorphism in Programming, in "J. Comput. System. Sci.,17, NO.3, 348-375."

中島玲二 (1982), "数理情報学入門:スコット・プログラム理論",朝倉書店

J. C. Reynolds (1985), Three approaches to type structures, in "LNCS., vol. 185, 97-138"

J. C. Reynolds (1981), Using Category Theory to Design Implicit Conversions and Generic Operators, in "LNCS. vol. 94,211-258"

M. B. Smyth and G. D. Plotokin (1982), The category-theoretic solution of recursive domain equations, in "SIAM J. Comput.,11,No.4, 761-783."