数学入門公開講座

平成6年8月8日（月）から8月12日（金）まで

京都大学数理解析研究所

講師及び内容

1．代数曲線の幾何（6時間15分）

京都大学数理解析研究所・教授　森　　重　文

代数幾何は代数的に定義された図形（代数多様体）を研究する学問である。19世紀に始まり、現在までに目覚ましい発展を遂げている。図形には次元という概念があり、現在は1次元（曲線）、2次元（曲面）そして3次元までが大まかに分類されてきている。

ここでは、曲線について、その形そして分類などを中心に入門的な話をする。

2．プログラミング言語の数理モデル（6時間15分）

京都大学数理解析研究所・助教授　大　堀　　淳

プログラミング言語は、単なる計算の手順を記述する手段であるばかりでなく、複雑なプログラムを構築する上で必要な抽象化の概念と構造化の機構を提供するものです。ここでは、プログラミング言語の持つべき種々の望ましい性質の分析や新しいプログラミング言語の設計などを行なう基礎となる数学的モデルの概略を解説した後、プログラミング言語研究における最近の話題を幾つか紹介します。

3．楕円曲線と整数論（6時間15分）

京都大学数理解析研究所・助手　玉　川　安騎男

楕円曲線は、代数幾何学的には、（有理点を1つ与えられた）種数1の代数曲線として特徴づけられる比較的易しい代数多様体であるが、整数論的には、多くの重要な問題（しかもその多くは現在もなお未解決）と関連した、たいへん豊かな対象である。この講義では、予備知識の解説を含む楕円曲線についての入門的な話と並行して、いくつかのより進んだ「夢のある」話も折り込んでいく予定である。

時　間　割

日 / 時間	8月8日(月)	9日(火)	10日(水)	11日(木)	12日(金)
10：30～11：45	森	森	森	森	森
11：45～13：00	休　憩				
13：00～14：15	大堀	大堀	大堀	大堀	大堀
14：15～14：45	休　憩				
14：45～16：00	玉川	玉川	玉川	玉川	玉川

平成6年度(第16回)数学入門公開講座テキスト(京都大学数理解析研究所, 平成6年8月8日～8月12日開催)

プログラミング言語の数理モデル

京都大学数理解析研究所・助教授　大堀　淳

1994, AUGUST 8, 9, 10, 11, 12,　　13:00 ~ 14:15

プログラミング言語の数理モデル

1994 年度 数理解析研究所 数学入門公開講座 ノート

大堀 淳
京都大学 数理解析研究所

1 はじめに

プログラミング言語は，単にコンピュータの動作を指示するのにとどまらず，ソフトウェアを構築するための枠組みを与える重要なものである．堅牢な理論的基礎をもつ高水準プログラミング言語は，規模と複雑さを増しつつあるソフトウエアの開発体系を作り上げていく上で欠かせないものである．本講座では，プログラミング言語の設計や分析の基礎となる数学的なモデルを解説し，さらに時間が許せば，プログラミング言語に関する最近の研究動向の幾つかを紹介する．

本稿は，講座で説明する種々のシステムの定義やその諸性質をまとめたものである．講義では，本ノートの内容を中心に，例を多く用いて，分かりやすく説明する．

2 プログラミング言語のモデル

コンピュータの原理は，基本的な算術演算や条件判定等の単純な命令を実行し続けるという簡単なものであるが，その問題解決能力は極めて大きく，「航空機を自動的に着陸させる」といった複雑で大規模な問題も，その問題の解決手順を表すプログラムをコンピュータの命令列として用意しさえすれば解くことができる．しかしながら，人間が，複雑なプログラムを単純な命令の列として記述することは極めて困難である．プログラミング言語は，プログラムのより高水準の記述を，コンピュータが実行可能な命令列に翻訳するプログラム記述システムである．その様な記述システムであるプログラミング言語の主な役割は，複雑なソフトウェアを設計・構築するために必要な抽象化の概念と構造化の機構を提供することである．望ましいプログラミング言語を設計する為には，問題解決により適した高水準のプログラム記述が可能な計算のモデル (または抽象機械) を構築すること，並びにその計算モデルが表現する計算を現実のコンピュータの命令列として実現するための翻訳技術を開発することが必要である．本講座では前者のプログラミング言語のための計算モデル及びその基本的な性質に焦点を当てる．

今日のプログラミング言語の基礎となっている計算モデルの代表的なものは以下の 3 つである．

1) 手続き型計算モデル
記憶領域を変更することを計算の実行と考えるモデルである．このモデルに基づく言語では，必要な抽象化の概念および構造化の機構は，記憶領域の操作に関するよく現れるパターンを一まとまりの手続きとして提供することによって実現される．

このモデルは現実のコンピュータの動作原理をほぼ忠実に反映したものであり，従ってこのモデルに基づく言語の実装は比較的簡単である．しかしながらこのモデルでは，コンピュータの実装上の概念である記憶領域をプログラムが明示的に操作するため，プログラムの表現する計算の数学的な意味が必要以上に複雑となり，プログラムの性質の分析が困難となる．

2) 関数型計算モデル
関数の値を求めることを計算の実行と考えるモデルである．このモデルに基づく言語においては，必要な抽象化の概念および構造化の機構は，関数の定義機構を通じて実現される．とくに，関数を引数として受け取る関数や，関数を値として返す関数を使用することにより，種々のプログラム構造を簡潔に表現可能である．

関数は抽象度が高くかつ数学的な分析にも適していることから，このモデルは，プログラミング言語の研究の基礎として広く用いられている．しかも最近の研究によれば，記憶領域のなどの手続き型プログラミング言語における概念の多くも，関数型計算のモデルで表現可能であり，その性質も関数型プログラミングの研究を通じて発達した理論を通じて調べることができるようになって来ている．

3) 論理型計算モデル
定理の証明をサーチすることを計算の実行と考えるモデルである．このモデルを基礎とする言語では，必要な抽象化の概念および構造化の機構は，種々の定理や推論規則の定義機構として提供される．

定理の証明は論理学の枠組みで厳密に扱うことが可能な抽象度の高い概念であり，関数型プログラミングと同様の利点をもっており，このモデルに基づくプログラミング言語の理論も広く研究されている．

手続き型計算モデル及び関数型計算モデルは逐次的アルゴリズムを表現するのに適しており，それぞれPASCALやC，あるいはLispやMLなどの汎用のプログラミング言語の基礎となっている．これに対して論理型計算モデルは，探索を主な要素とする問題解決に適しており，人工知能の分野で主に使用されている．代表的な論理型プログラミング言語にはProlog言語がある．

本講座では，これららの中で，関数型プログラミング言語のモデルを解説し，最近の研究動向を紹介する．まず第3節で，関数型プログラミング言語における関数定義とその評価機構の抽象的なモデルである，単純な型付きラムダ計算について解説し，次に第4節で，単純な型付きラムダ計算を，再帰的な関数定義機構と再帰的データ型を扱えるように拡張する．この拡張によって，静的型システムをもつ関数型プログラミング言語の機能のほとんどを表現することが可能である．次に第5節で，最近注目を集めているプログラミング言語MLに採用されている多相型と型推論について解説する．最後に第6節において現在の研究トピックを幾つか紹介する．

3 型付きラムダ計算：関数定義とその評価機構のモデル

関数型プログラミング言語は，プログラムを関数として捕らえプログラムの行う計算を関数の値を求めることで表現しようとするものである．しかし数学における関数の概念には，入力から出力を得るために必要な計算過程が表現されていない．コンピュータで実行可能なプログラミング言語をモデル化するためには，単なるグラフとして静的に捉えられた関数の概念を，入力から出力を計算する過程をも表現した動的な「計算可能な関数」へと洗練する必要がある． そのような計算可能な関数の概念の代表的ものには，**帰納的関数**と**ラムダ計算**がある．この二つの中で，ラムダ計算がよりプログラミング言語に近く，プログラミング言語の基礎として広く研究されている．

ラムダ計算は，型なしのシステムと型付きのシステムに大別される．型なしラムダ計算は，コンピュータが表現可能な計算の理論的な分析等に便利である．しかし，ほとんどすべてのプログラミング言語は種々のデータ型を持っており，さらに多くのプログラミング言語では型システムがソフトウェア構築上の抽象化および，複雑なプログラムを構造化する上で重要な役割を果たしている．従って，プログラミング言語のモデルとしては型付きラムダ計算がより適している．型付きラムダ計算には種々のシステムがあるが，そのなかでも最も基本的なシステムが，単純な型付きラムダ計算(the simply typed lambda calculus)である．このシステムは，関数型言語の中枢である関数の定義とその評価機構のモデルとして有用である．以下本節では，単純な型付きラムダ計算の定義とその諸性質に付いて解説する．

種々の現実の関数型プログラミング言語は，この単純な型付きラムダ計算を，再帰的関数や再帰的データ型の定義機構を導入して拡張したシステムに対応している．この拡張に付いては第4節で説明する．

3.1 ラムダ式の集合

ラムダ計算においては，プログラムはラムダ式(lambda expression)でモデル化される． 単純な型付きラムダ計算のラムダ式の集合は，メタ変数Mを用いて以下の疑似分脈自由文法で与えられる．

$$
\begin{aligned}
M \quad ::= \quad & c^{\tau} \mid x \mid \lambda x:\tau.M \mid (MM) \\
& \mid (M,M) \mid M.1 \mid M.2 \\
& \mid inl_{\tau_1+\tau_2}(M) \mid inr_{\tau_1+\tau_2}(M) \mid case\ M\ of\ inl => M, inr => M\ end
\end{aligned}
$$

$$\mid \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M$$

"|" は「または」の意味であり，"|" で区切られた各式は，メタ変数 M で代表される集合を決定する生成規則と解釈する．例えば，$\cdots \mid (M,M)$ は，「もし M_1, M_2 がともに式なら，(M_1, M_2) も式である」という生成規則を意味する．この例から分かるように，M はこの文法で定義される集合の任意の要素を代表する．M が同一の式に2回現れてた場合でも，同一の式を表しているわけではない．

c^{τ} は型 τ を持つ定数を表す．型についてはは後の 3.3節で詳しく述べる．x は加算無限個ある変数を表す．(M_1, M_2) は M_1 と M_2 からなる組を表す．M が組であるとき，$M.1$ と $M.2$ はそれぞれ，M の第一要素および第二要素を取り出す演算を表す．$inl_{\tau_1+\tau_2}(M)$，$inr_{\tau_1+\tau_2}(M)$ は，型 τ_1 で表される集合と型 τ_2 で表される集合の直和をあらわす．直和とは種類の違った二つの集合の和集合のことであり，その要素は，(τ_1 か τ_2 かの) どちらの集合に入っていたかを示す印 (プログラミング言語の用語ではタグと呼ぶ) のついた要素である．プログラミング言語でのバリアントレコードやタグ付きユニオン型に相当する．$\mathit{case}\ M_1\ \mathit{of}\ \mathit{inl} => M_2, \mathit{inr} => M_3\ \mathit{end}$ は，直積の要素 M_1 がどちらの集合の要素かを分析し，それに応じて処理を行なう構文であり，プログラミング言語の $case$ 文に相当する．$\lambda x : \tau.M$ は，型 τ の値をもつ引き数を x として受け取り，x を含む式 M の値を計算する関数を表す．記法 $\lambda x : \tau.M$ における x は，数学における $f(x) = x + 1$ や $\int (x+1)dx$ などの記法における x と同様に，種々の値を表す仮の名前であり，名前自身は重要ではない．すなわち，$\int f(x)dx$ が $\int f(y)dy$ と同じ意味を持つのと同様，M の中の x を別の新しい名前 y で置き換えて得られる式を M' とすると，$\lambda x : \tau.M$ は $\lambda y : \tau.M'$ と同じ意味をもつ．このような仮の変数を束縛変数といい，束縛変数を導入する記法 $\lambda x : \tau.$ をラムダ束縛と呼ぶ．ラムダ計算の大きな特徴は，このラムダ束縛によって，関数の名前を用いることなく関数を定義する記法を導入した点にある．例えば通常数学において $f(x) = x + 1$ と表現される関数は，$\lambda x : int.x + 1$ と表現される．ラムダ抽象による関数の定義は，$f(x) = \cdots$ といった記法に比べて，関数に名前を付けることを要求しないため概念的により単純であり，また関数を値として取扱う上でも便利である．$(M_1\ M_2)$ は，関数 M_1 を値 M_2 に適用する関数適用を表す．以下関数適用は左結合するものと約束し，可能な限り括弧を省略することにする．例えば，$(((M_1\ M_2)\ M_3)\ M_4)$ のかわりに単に $M_1\ M_2\ M_3 M_4$ と書く．構文 $\mathit{if}\ M_1\ \mathit{then}\ M_2\ \mathit{else} M_3$ は，まず M_1 を評価し，結果が $true$ なら M_1 を，$false$ なら M_3 を評価する条件構文である．

ラムダ式は以上の通り極めて簡潔な構文を持つが，その表現力は高く，プログラム言語の基礎的な構造を表現することができる．たとえば次のように表現された値の束縛とその束縛のもとでの式の評価

```
begin x1 = M1;
      x2 = M2;
         ⋮
      xn-1 = Mn1
in
   M;
end
```

は，以下のラムダ式で表現できる．

$$((\cdots((\lambda x_n : \tau_n.(\lambda x_{n-1} : \tau_{n-1}.\cdots(\lambda x_2 : \tau_2.(\lambda x_1 : \tau_1.M))\cdots))\ M_n)M_{n-1})\cdots M_2)M_1)$$

ここで型 $\tau_1, \ldots, \tau_n$ はそれぞれ $M_1, \ldots, M_n$ の型であるが，これらは後に説明する型システムによって自動的に計算可能である．名前付きの関数定義と，以下の文脈でのその使用

```
function f (x:τ) =
   begin
      M
   end;
   ⋮
```

は以上の変数束縛機構を用いて以下のように表現できる．

```
begin f = λx:τ.M in
   ⋮
end
```

$$
\begin{aligned}
FV(c^{\tau}) &= \emptyset \\
FV(x) &= \{x\} \\
FV(\lambda x:\tau.M) &= FV(M)\setminus\{x\} \\
FV((M_1\ M_2)) &= FV(M_1)\cup FV(M_2) \\
FV((M_1,M_2)) &= FV(M_1)\cup FV(M_2) \\
FV(M.1) &= FV(M) \\
FV(M.2) &= FV(M) \\
FV(inl_{\tau_1+\tau_2}(M)) &= FV(M) \\
FV(inr_{\tau_1+\tau_2}(M)) &= FV(M) \\
FV(case\ M_1\ of\ inl => M_2, inr => M_2\ end) &= FV(M_1)\cup FV(M_2)\cup FV(M_3) \\
FV(if\ M_1\ then\ M_2\ else\ M_3) &= FV(M_1)\cup FV(M_2)\cup FV(M_3)
\end{aligned}
$$

図 1: 自由変数の定義

3.2 ラムダ式の表現する計算

ラムダ計算における計算の基本的な考えは，ラムダ式の一部を別のラムダ式で書き換えることを計算の実行の単位と見なすことである．複雑な計算は，書き換えを繰り返し行なうことによって実行される．この考えに基づき，ラムダ式の表現する計算を厳密に定義するのが，ラムダ計算の操作的意味論である．操作的意味論を定義するためには，それぞれのラムダ式構成子の意図する意味に従って，ラムダ式の書き換え規則を定義しなければならない．最も重要な書き換え規則は，関数適用 $(\lambda x:\tau.M)\ N$ のためのものである．$\lambda x:\tau.M$ における x は実際の引数を表す仮の名前であるから，もとめる規則は，M の中の x を N で置き換える規則である．この規則を厳密に定義するために，変数の置き換え規則をまず定義する．

置き換えの対象となる変数は，ラムダ束縛されていない変数，すなわち自由変数である．式 M の中の自由変数の集合 $FV(M)$ の定義は図1の様に与えられる．式 M の中の自由変数 x を式 N で置き換えて得られる式を $M[N/x]$ と書くことにする．$M[N/x]$ の定義を図2に示す．$\lambda y:\tau.M$ の置き換えで，$x=y$ のときは M の中の x は置き換えられないことに注意．(この式に関する置き換え規則の3番目の場合は，自由変数が，意図せずに，束縛変数に変換されてしまう事態を防ぐためのものである．詳しくは講義で説明する．) この自由変数の置き換え規則を用いれば，関数適用 $((\lambda x:\tau.M)\ N)$ の書き換えの結果は $M[N/x]$ と定義できる．この規則 $((\beta)$- 規則) を含めたラムダ項の書き換え規則の集合を図3に示す．(π) は直積の要素を取り出す為の規則，(σ) は直和の要素を判定しその属していた集合に従った処理を行う為の規則，(if) は条件文の評価規則である．これ以外に，例えば算術演算などを行なう関数を定数として導入する場合はそれぞれの対応する書き換え規則を仮定する．例えば自然数の加算演算を表現する定数 $plus$ を導入する場合は，$n_3=n_1+n_2$ なる任意の自然数 n_1,n_2,n_3 に対して以下の書き換え規則を導入する．

$$(\delta)\qquad plus(n_1,n_2)\stackrel{\delta}{\Longrightarrow} n_3$$

これら定数式との間に導入される書き換え規則を総称して (δ)- 規則と呼ぶ．

ラムダ式 M の一部に，以上の書き換え規則のいずれかを1回適用した結果式 N がえられるとき，

$$M \longrightarrow N$$

と書き，$\longrightarrow$ の反射的，推移的閉包を $\longrightarrow\!\!\!\!\!\rightarrow$ とかく．すなわち $M \longrightarrow\!\!\!\!\!\rightarrow N$ となるのは，M の一部に0回以上書き換え規則を適用した結果 N がえられるときである．この関係に関して以下の重要な性質が成り立つことが知られている．

定理 1 (Church-Rosser の定理) 任意のラムダ式 M に対して，もし $M \longrightarrow\!\!\!\!\!\rightarrow N_1$，$M \longrightarrow\!\!\!\!\!\rightarrow N_2$ なる二つのラム

$$
\begin{aligned}
c[N/x] &= c \\
y[N/x] &= \begin{cases} N & \text{if } x = y \\ y & \text{if } x \neq y \end{cases} \\
(\lambda y:\tau.M)[N/x] &= \begin{cases} \lambda y:\tau.M & \text{if } x = y \\ \lambda y:\tau.M[N/x] & \text{if } y \neq x \text{ and } y \notin FV(N) \\ \lambda z.(M[z/y])[N/x] & \text{if } y \neq x \text{ and } y \in FV(N) \text{ (choosing fresh } z) \end{cases} \\
(M_1\ M_2)[N/x] &= (M_1[N/x]\ M_2[N/x]) \\
(M_1, M_2)[N/x] &= (M_1[N/x], M_2[N/x]) \\
M.1[N/x] &= M[N/x].1 \\
M.2[N/x] &= M[N/x].2 \\
(inl_{\tau_1+\tau_2}(M))[N/x] &= inl_{\tau_1+\tau_2}(M[N/x]) \\
(inr_{\tau_1+\tau_2}(M))[N/x] &= inr_{\tau_1+\tau_2}(M[N/x]) \\
(case\ M_1\ of\ inl => M_2, inr => M_2)[N/x] &= case\ M_1[N/x]\ of\ inl => M_2[N/x], inr => M_2[N/x] \\
(if\ M_1\ then\ M_2\ else\ M_3)[N/x] &= if\ M_1[N/x]\ then\ M_2[N/x]\ else\ M_3[N/x]
\end{aligned}
$$

図 2: 変数の代入規則

$$
\begin{aligned}
&(\beta) && (\lambda x:\tau.M)\ N \stackrel{\beta}{\Longrightarrow} M[N/x] \\
&(\pi_1) && (M_1, M_2).1 \stackrel{\pi_1}{\Longrightarrow} M_1 \\
&(\pi_2) && (M_1, M_2).2 \stackrel{\pi_2}{\Longrightarrow} M_2 \\
&(\sigma_l) && (case\ inl_{\tau_1+\tau_2}(M)\ of\ inl => M_1, inr => M_2) \stackrel{\sigma_1}{\Longrightarrow} M_1 M \\
&(\sigma_r) && (case\ inr_{\tau_1+\tau_2}(M)\ of\ inl => M_1, inr => M_2) \stackrel{\sigma_2}{\Longrightarrow} M_2 M \\
&(if_1) && if\ true\ then\ M_1\ elseM_2 \stackrel{if}{\Longrightarrow} M_1 \\
&(if_2) && if\ false\ then\ M_1\ elseM_2 \stackrel{if}{\Longrightarrow} M_2
\end{aligned}
$$

図 3: ラムダ式の書き換え規則

$$(\beta^v) \qquad E[(\lambda x:\tau.M)\ V] \stackrel{\beta}{\Longrightarrow} E[M[V/x]]$$

$$(\pi_1^v) \qquad E[(V_1,V_2).1] \stackrel{\pi_1}{\Longrightarrow} E[V_1]$$

$$(\pi_2^v) \qquad E[(V_1,V_2).2] \stackrel{\pi_2}{\Longrightarrow} E[V_2]$$

$$(\sigma_l^v) \qquad E[(case\ inl_{\tau_1+\tau_2}(V)\ of\ inl => M_1, inr => M_2)] \stackrel{\sigma_1}{\Longrightarrow} E[M_1V]$$

$$(\sigma_r^v) \qquad E[(case\ inr_{\tau_1+\tau_2}(V)\ of\ inl => M_1, inr => M_2)] \stackrel{\sigma_2}{\Longrightarrow} E[M_2V]$$

$$(if_1^v) \qquad E[if\ true\ then\ M_1\ elseM_2] \stackrel{if}{\Longrightarrow} E[M_1]$$

$$(if_2^v) \qquad E[if\ false\ then\ M_1\ elseM_2] \stackrel{if}{\Longrightarrow} E[M_2]$$

図 4: call-by-value 戦略でのラムダ式の文脈書き換え規則

ダ式 N_1, N_2 が存在すれば，あるラムダ式 N_3 が存在して $N_1 \longrightarrow\!\!\!\!\!\rightarrow N_3$ かつ $N_2 \longrightarrow\!\!\!\!\!\rightarrow N_3$ となる． ∎

この性質は，書き換えの結果最終的に得られる値は書き換え規則の適用順序に依存しないことを意味する．したがって，ラムダ式間の関係 $\longrightarrow\!\!\!\!\!\rightarrow$ は，ラムダ式 M に，$M \longrightarrow\!\!\!\!\!\rightarrow N$ かつ 高$N'.N \longrightarrow N'$ であるラムダ式 N を対応させる (部分) 関数を定義していると見なすことができる．

しかしながら関係 $\longrightarrow\!\!\!\!\!\rightarrow$ は書き換えの順序を規定していないため，この操作的意味論は非決定的であり，計算のモデルとしては不十分である．例えばラムダ式 $((\lambda x : int.plus(x,x))(plus(3,3))$ には以下の二つの書き換え列が存在する．

$$((\lambda x : int.plus(x,1))(plus(3,3)) \longrightarrow plus(plus(3,3),1) \longrightarrow plus(6,1) \longrightarrow 7$$

$$((\lambda x : int.plus(x,1))(plus(3,3)) \longrightarrow ((\lambda x : int.plus(x,1))6) \longrightarrow plus(6,1) \longrightarrow 7$$

プログラミング言語のモデルとしては，書き換えの順序をも規定した計算のモデルを定義する必要がある．通常のプログラミングでは，上記のような複数の書き換えの可能性がある場合には，(1) 関数適用の前に関数の引き数を書き換え，(2) 同一のレベルの式については左の式を先に書き換えるという書き換え戦略をとる．この書き換え戦略のことを call-by-value 戦略と呼ぶ．

call-by-value 戦略基づく計算を定義するためにまず，値の集合を V をメタ変数として以下の文法で定義する．

$$V ::= c \mid \lambda x:\tau.M \mid (V,V) \mid inl_{\tau_1+\tau_2}(V) \mid inr_{\tau_1+\tau_2}(V)$$

ここで $\lambda x:\tau.M$ の中の M は値でなくてもよい．値の集合は書き換え済みのラムダ式の集合に対応する．この値の概念を用いて，ラムダ式の中に複数存在する書き換え可能な部分式の書き換え順序を定義する．その方法は幾つかあるがここでは「評価文脈」(evaluation context) の概念を用いた定義を紹介する．ラムダ式の文脈とは，直観的には，別のラムダ式が埋め込まれる穴を含むラムダ式のことである．評価文脈とは，「ラムダ式の中の次に変換すべき部分式」を文脈の穴によって示した特殊な文脈である．ラムダ式が埋め込まれるべき穴を [ ] で表し，評価文脈を $E[\ ]$ で表すことにする．ラムダ式 M が評価分脈 $E[\ ]$ によって $M = E[M_0]$ と表されるとき，M の中の部分式 M_0 が最初に変換される部分式である．ラムダ式の計算を定義するために，書き換え可能な任意のラムダ式 M に対してつねに，$M = E[M_0]$ となる評価分脈 $E[\ ]$ と書き換え可能なラムダ式 M_0 が一意に定まるように評価分脈を定義する．call-by-value 戦略に対応する評価分脈の定義は以下の通りである．

$$\begin{aligned} E \ ::= \ & [\ ] \mid VE[\ ] \mid E[\ ]\ M \mid (E[\ ], M) \\ & \mid (V, E[\ ]) \mid E[\ ].1 \mid E[\ ].2 \\ & \mid inl_{\tau_1+\tau_2}(E[\ ]) \mid inr_{\tau_1+\tau_2}(E[\ ]) \mid case\ E[\ ]\ of\ inl => M, inr => M \\ & \mid if\ E[\ ]\ then\ M\ elseM \end{aligned}$$

次に，call-by-value 戦略での文脈書き換え規則を図 4の様に定義する．これらを用いて，戦略でのラムダ式の書き換え規則を，以下のように分脈書き換え規則として定義する．

$$M \longrightarrow_v N \ \text{ if } M = E[M_0], N = E[N_0], E[M_0] \Longrightarrow^v E[N_0]$$

以前同様に $\longrightarrow\!\!\!\!\!\rightarrow_v$ を $\longrightarrow_v$ の反射的推移的閉包とする．この定義は決定的な書き換え規則となっている．すなわち，与えられた M に対して，$M = E[M_0]$，$E[M_0] \Longrightarrow E[N_0]$ となる $E[\ ]$，M_0，N_0 は高々一組しか存在しない．例えば前に例をあげたラムダ式 $(\lambda x : int.plus(x,1))plus(3,3)$ では，$E[\ ] = (\lambda x : int.plus(x,1))[\ ]$，$M_0 = plus(3,3)$ となり，$plus(3,3)$ がまず書き換えられる．この決定的なラムダ式間の書き換え関係 $\longrightarrow\!\!\!\!\!\rightarrow_v$ をラムダ式をプログラムと考えた場合のプログラムの行なう計算と考えることにする．

3.3 ラムダ計算の型システム

文法的に正しいラムダ式がすべて意味あるプログラムに対応しているとは限らない．例えば $plus(true,3)$ などは意味ある計算を表現していない．ラムダ計算の型システムは，ラムダ式をその表現する計算の性質に応じて型に分類することによって，$plus(true,3)$ などの意味を持たない式を排除するシステムである．

単純な型付きラムダ計算の型の集合は，τ をメタ変数とする以下の文法で与えられる．

$$\tau ::= b \mid \tau \times \tau \mid \tau + \tau \mid \tau \rightarrow \tau$$

ここで b は，int(整数型)，$bool$(真理値型)，$string$(文字列型) などの種々の原子型 (atomic type) を表す．$\tau_1 \times \tau_2$ は τ_1 と τ_2 の直積型，$\tau_1 + \tau_2$ は τ_1 と τ_2 の直和型，$\tau_1 \rightarrow \tau_2$ は τ_1 から τ_2 への関数型である．

式 M が型 τ を持つ条件を厳密に規定するシステムが，型付きラムダ計算の型システムである．一般に式 M は自由変数を含むから，式 M の型はその自由変数の型を与えなければ決定出来ない．そこで，型環境 $\mathcal{A}$ を変数の集合から型への関数として，型システムを「式 M が型環境 $\mathcal{A}$ のもとで型 τ をもつ」という性質を表現する以下の形の型判定を導出するシステムとして定義する．

$$\mathcal{A} \rhd e : \tau$$

導出システムは以下の形をした推論規則の集合からなる．

$$(\text{RULE}) \qquad \frac{\mathcal{A}_1 \rhd e_1 : \tau_1, \ldots, \mathcal{A}_n \rhd e_n : \tau_n}{\mathcal{A} \rhd e : \tau}$$

この規則は「各型判定 $\mathcal{A}_i \rhd e_i : \tau_i$ がすべて成り立つならば，型判定 $\mathcal{A} \rhd e : \tau$ が成り立つ」という推論規則を表す．仮定が空であり結論のみからなる

$$\mathcal{A} \rhd e : \tau$$

の形をした規則は常に成り立つ型判定をあらわし，導出システムの公理に相当する．

単純な型付きラムダ計算の型付け規則の集合を図5に示す．公理 (CONST) は，型つき定数はあらかじめ定められた型を持つラムダ式として使用できることを示す．公理 (VAR) は，変数 x がすでに型環境の中で $x : \tau$ と定義されていれば，型 τ を持つラムダ式として使用できることを示す．例えば型判定

$$\{x : int\} \rhd x : int$$

は常に成り立つ．規則 (ABS) における記法 $\mathcal{A}\{x : \tau\}$ は以下が成り立つ型環境 $\mathcal{A}'$ を表す．$domain(\mathcal{A}') = domain(\mathcal{A}) \cup \{x\}$，$\mathcal{A}'(x) = \tau$，かつ $\mathcal{A}'(y) = \mathcal{A}(y), y \neq x$．規則 (ABS) は，ラムダ式 $\lambda x : \tau_1.M_1$ が，型環境 $\mathcal{A}$ のもとで型 $\tau_1 \rightarrow \tau_2$ を持つのは，ラムダ式 M_1 が型環境 $\mathcal{A}\{x : \tau_1\}$ のもとで型 τ_2 を持つ時であることを示している．これは，「環境 $\mathcal{A}$ で定義されている変数を含む関数定義 $\lambda x : \tau.M_1$ を型チェックするためには，型環境 $\mathcal{A}$ に変数 x の型付け $x : \tau_1$ を追加（もし $\mathcal{A}$ ですでに x の型付けがされていれば，それを τ_1 に変更）した型環境のもとで，関数の本体 M_1 を型チェックすればよい」という，通常コンパイラーが関数をコンパイルする際に行なっている型チェック手続きを抽象的な規則で表現したものと理解できる．表記 $\mathcal{A}\{x : \tau_1\}$ によって，通常のプログラム言語における変数の静的な可視性に関する規則 (static scope rule) が表現されていることに注意．これら規則は，前に定義したラムダ式の書き換え規則に対応している．例えば書き換え規則 (σ) によれば，$\tau_1 + \tau_2$ の型の直和型データ M は，$case\ M\ of\ inl => M_1, inr => M_2$ の形の $case$ 文で処理される．この際，もし M が $inl(M')$ の形をしていれば，$M_1\ M'$ に，またもし M が $inr(M')$ の形をしていれば，$M_2\ M'$ に変換される．しかるに規則 (INL) と (INR) によれば，$\tau_1 + \tau_2$ の型の直和型データ M が $inl(M')$ の形をしていれば M' の型は τ_1 であり，$inr(M')$ の形をしていればその型は τ_2 である．そこで規則 (σ) の変換後の型が τ であるためには，$case$ 文の中 M_2, M_3 の型はそれぞれ $\tau_1 \rightarrow \tau$ と $\tau_2 \rightarrow \tau$ の型を持たねばならない．型付け規則 (CASE) は以上の条件を正確に表現している．

(CONST) $\mathcal{A} \triangleright c^{\tau} : \tau$

(VAR) $\mathcal{A} \triangleright x : \tau \quad$ if $\mathcal{A}(x) = \tau$

(ABS) $$\frac{\mathcal{A}\{x : \tau_1\} \triangleright M_1 : \tau_2}{\mathcal{A} \triangleright \lambda x : \tau_1 . M_1 : \tau_1 \to \tau_2}$$

(APP) $$\frac{\mathcal{A} \triangleright M_1 : \tau_1 \to \tau_2 \quad \mathcal{A} \triangleright M_2 : \tau_1}{\mathcal{A} \triangleright M_1\ M_2 : \tau_2}$$

(PAIR) $$\frac{\mathcal{A} \triangleright M_1 : \tau_1 \quad \mathcal{A} \triangleright M_2 : \tau_2}{\mathcal{A} \triangleright (M_1, M_2) : \tau_1 \times \tau_2}$$

(PROJ1) $$\frac{\mathcal{A} \triangleright M : \tau_1 \times \tau_2}{\mathcal{A} \triangleright M_1.1 : \tau_1}$$

(PROJ2) $$\frac{\mathcal{A} \triangleright M : \tau_1 \times \tau_2}{\mathcal{A} \triangleright M_1.2 : \tau_2}$$

(INL) $$\frac{\mathcal{A} \triangleright M : \tau_1}{\mathcal{A} \triangleright inl_{\tau_1+\tau_2}(M) : \tau_1 + \tau_2}$$

(INR) $$\frac{\mathcal{A} \triangleright M : \tau_2}{\mathcal{A} \triangleright inr_{\tau_1+\tau_2}(M) : \tau_1 + \tau_2}$$

(CASE) $$\frac{\mathcal{A} \triangleright M_1 : \tau_1 + \tau_2 \quad \mathcal{A} \triangleright M_2 : \tau_1 \to \tau \quad \mathcal{A} \triangleright M_3 : \tau_2 \to \tau}{\mathcal{A} \triangleright case\ M_1\ of\ inl => M_2, inr => M_3 : \tau}$$

(IF) $$\frac{\mathcal{A} \triangleright M_1 : bool \quad \mathcal{A} \triangleright M_2 : \tau \quad \mathcal{A} \triangleright M_3 : \tau}{\mathcal{A} \triangleright if\ M_1\ then\ M_2\ else\ M_3 : \tau}$$

図 5: 単純な型付きラムダ計算の型付け規則

$$\dfrac{\dfrac{\{x:int\}\rhd mul:(int\times int)\to int \qquad \dfrac{\{x:int\}\rhd x:int \quad \{x:int\}\rhd x:int}{\{x:int\}\rhd (x,x):int\times int}}{\{x:int\}\rhd mul(x,x):int}}{\emptyset\rhd\lambda x:int.mul(x,x):int\to int}$$

図 6: 型判定の導出の例

ラムダ式 M が型環境 $\mathcal{A}$ のもとで型 τ を持つのは，型判定 $\mathcal{A}\rhd M:\tau$ が，上記ルールを有限回用いて導出できる時である．図 6に型判定の導出の例を示す．

以上定義した型システムに従い，与えられたラムダ式の型が正しいか，また正しいならどのような型を持つかを計算するアルゴリズムを書くことができる．これは，型付け規則を型判定問題を部分問題に分解する規則と見なし，それらを再帰的に適用することにより行なうことができる．図 7に，この戦略に基づき型チェックを行なう再帰的な関数の定義を示す．関数 $type$ は，型環境 $\mathcal{A}$ の下でラムダ式 M が型を持てばそれを返し，型を持たなければ型エラーを報告する．

命題 1 任意のラムダ式 M と型環境 $\mathcal{A}$ について，もし $type(M,\mathcal{A})=\tau$ であれば $\mathcal{A}\rhd M:\tau$ が成り立ちかつ，もし $type(M,\mathcal{A})=Type\_Eror$ であれば $\mathcal{A}\rhd M:\tau$ となる τ は存在しない． ∎

証明は式 M の構造に関する帰納法による． $type(\mathcal{A},M)$ を計算することは， M に対する型判定の導出を根本から再構築することを試みることに相当する．

静的型システムが有用であるためには，それが“正しく”なければならない．すなわち，型システムによって定められたプログラムの型が，実際にそのプログラムを実行することによって得られる結果の型と一致しなければならない．型理論では，この性質を型システムの健全性 (soundness) と呼ぶ．健全性はすべての静的型システムが満たすべき最も基本的な性質である．ラムダ計算の型システムの健全性を示す一つの方法は，ラムダ式の変換関係 $\longrightarrow\!\!\!\!\!\rightarrow$ が型判定を保存すること，すなわち以下の性質を示すことである．

定理 2 M を与えられた任意の式とする．もし， $\mathcal{A}\rhd M:\tau$ が成り立ちかつ $M\longrightarrow\!\!\!\!\!\rightarrow N$ であれば， $\mathcal{A}\rhd N:\tau$ も成り立つ． ∎

$\longrightarrow\!\!\!\!\!\rightarrow_v$ は $\longrightarrow\!\!\!\!\!\rightarrow$ の特殊な場合であるから，上記の定理はプログラムの実行のモデルである $\longrightarrow\!\!\!\!\!\rightarrow_v$ についても成立する．この性質は「静的に型チェックされたプログラムは，実行時に型エラーを起こさない」という型システムの望ましい性質を保証している．例えば以下の型判定を考えてみよう．

$$\emptyset\rhd(\lambda x:int.mul(x,x))\ 3:int$$

上記定理から，このラムダ式の演算の結果は int 型であることが保証されている．実際，このラムダ式の演算は $(\lambda x:int.mul(x,x))\ 3\longrightarrow\!\!\!\!\!\rightarrow_v 9$ であらわされ，その結果は確かに int 型である．

4 単純な型付きラムダ計算の拡張

以上紹介した単純な型付きラムダ計算を，より表現能力を大きくするための拡張が幾つか提案されている．本講座では，それらの中でプログラミング言語にとって重要な再帰的関数定義と再帰的型について説明する．

4.1 再帰的関数定義機構の導入

単純な型付きラムダ計算では型をもつラムダ式の計算は必ず停止することが知られている．

定理 3 任意のラムダ式 M に付いて， $\mathcal{A}\rhd M:\tau$ となる $\mathcal{A}$ と τ が存在すれば， M から始ま変換の系列 $M\longrightarrow M_1\longrightarrow M_2\longrightarrow\cdots$ はすべて有限である． ∎

$type(\mathcal{A}, M) =$
  $case\ M$ of
    c^τ $\Longrightarrow \tau$
    x $\Longrightarrow$ if $x \in domain(\mathcal{A})$ then $\mathcal{A}(x)$ else Type_Error
    $\lambda x : \tau_1 . M_1$ $\Longrightarrow \tau_1 \to type(\mathcal{A}\{x : \tau_1\}, M_1)$
    $M_1\ M_2$ $\Longrightarrow$ let $\tau_1 = type(\mathcal{A}, M_1)$
      $\tau_2 = type(\mathcal{A}, M_2)$
      in if $\tau_1 = \tau_2 \to \tau_3$ then τ_3 else Type_Error
    (M_1, M_2) $\Longrightarrow$ let $\tau_1 = type(\mathcal{A}, M_1)$
      $\tau_2 = type(\mathcal{A}, M_2)$
      in $\tau_1 \times \tau_2$.
    $M_1.1$ $\Longrightarrow$ let $\tau_1 \times \tau_2 = type(\mathcal{A}, M_1)$
      in τ_1.
    $M_1.2$ $\Longrightarrow$ let $\tau_1 \times \tau_2 = type(\mathcal{A}, M_1)$
      in τ_2.
    $inl_{\tau_1+\tau_2}(M_1)$ $\Longrightarrow$ let $\tau_1' = type(\mathcal{A}, M_1)$
      in if $\tau_1 = \tau_1'$ then $\tau_1 + \tau_2$ else Type_Error
    $inr_{\tau_1+\tau_2}(M_1)$ $\Longrightarrow$ let $\tau_2' = type(\mathcal{A}, M_1)$
      in if $\tau_2 = \tau_2'$ then $\tau_1 + \tau_2$ else Type_Error
    $case\ M_1\ of\ inl => M_2, inr => M_2$
      $\Longrightarrow$ let $\tau_1 = type(\mathcal{A}, M_1)$
        $\tau_2 = type(\mathcal{A}, M_2)$
        $\tau_3 = type(\mathcal{A}, M_3)$
      in if $\tau_1 = \tau_1^1 + \tau_1^2$ and
        $\tau_2 = \tau_1^1 \to \tau$ and
        $\tau_3 = \tau_1^2 \to \tau$ for some τ_1^1, τ_2^1, τ
      then τ
      else Type_Error
  $endcase$

図 7: 単純な型付きラムダ計算の型チェックアルゴリズム

従って，ラムダ式をプログラムのモデルと考えるならば，この定理は，すべてのプログラムがすべての入力に対して停止することを意味している．プログラムが停止するという性質は多くの場合望ましい性質ではあるが，実際我々が書くプログラムの多くは，入力の値によっては停止しないことがあるのがふつうである．さらに，すべての入力に対して停止するプログラムとしては実現出来ない有用なアルゴリズムが多数存在することが知られている．このことは，単純な型付きラムダ計算のモデルでは，有用なプログラムのすべてを表現出来ないことを意味する．

この問題は，関数の再帰的定義機構を導入することによって解決できる．そのための一つの方法は，ラムダ式の定義を以下のように拡張することである．

$$M ::= \cdots \mid fix\ f(x : \tau).M$$

直感的には，$fix\ f(x : \tau).M$ は，「現在定義中のラムダ抽象そのもの」を，変数名 f を通じて定義本体の M の中で使用することが許されるラムダ抽象 $\lambda x : \tau.M$ と考えることができ，プログラミング言語における再帰的な関数定義に対応している．例えば疑似コードで書かれた以下の再帰的関数：

```
function factorial(n:int) =
        begin
          if eq(n,0) then return(1)
          else return(mul(n, factorial(sub(n,1))))
        end
```

$$
\begin{aligned}
Factorial\ 2 &= ((fix\ f(n:int).if\ eq(n,0)\ then\ 1\ else\ mul(n,f\ (sub(n,1))))\ 2)\\
&\longrightarrow (\lambda n:int.if\ eq(n,0)\ then\ 1\ else\ mul(n,Factorial\ sub(n,1)))2)\\
&\longrightarrow if\ eq(2,0)\ then\ 1\ else\ mul(2,Factorial\ sub(2,1))\\
&\longrightarrow if\ false\ then\ 1\ else\ mul(2,Factorial\ sub(2,1))\\
&\longrightarrow mul(2,Factorial\ sub(2,1))\\
&= mul(2,((fix\ f(n:int).if\ eq(n,0)\ then\ 1\ else\ mul(n,f\ (sub(n,1))))\ (sub(2,1))))\\
&\longrightarrow mul(2,(\lambda n:int.if\ eq(n,0)\ then\ 1\ else\ mul(2,Factorial\ sub(n,1)))sub(2,1)\\
&\longrightarrow mul(2,(\lambda n:int.if\ eq(n,0)\ then\ 1\ else\ mul(n,Factorial\ sub(n,1)))1\\
&\longrightarrow mul(2,if\ eq(1,0)\ then\ 1\ else\ mul(1,Factorial\ sub(n,1)))\\
&\longrightarrow mul(2,if\ true\ then\ 1\ else\ mul(1,Factorial\ sub(n,1)))\\
&\longrightarrow mul(2,1)\\
&\longrightarrow 2
\end{aligned}
$$

図 8: 再帰的関数を含んだラムダ式の評価の例

は以下の以下のように表現される.

$$Factorial = fix\ f(n:int).if\ eq(n,0)\ then\ 1\ else\ mul(n,f\ (sub(n,1)))$$

ここで eq, mul, sub はそれぞれ整数の比較, 整数の積, 整数の減算を表す定数関数とする.

fix 構成子に対する変換規則は以下のように定義される.

$$(fix) \qquad fix\ f(x:\tau).M \stackrel{fix}{\Longrightarrow} \lambda x:\tau.M[(fix\ f(x:\tau).M)/f]$$

この変換規則を以前のラムダ計算の評価の規則の定義と統合すれば, 再帰的関数の定義を含んだラムダ計算の計算モデルがえられる. 評価分脈の定義は以前と同一でよい. 図8に再帰的関数を含んだ評価の例を示す.

この拡張によって, すべての計算可能な関数がラムダ式として表現できるととが確かめられる. それればかりか, 関数の再帰的定義機構によって, プログラミング言語の種々の制御機構を表現できる.

fix 構文に対する型付け規則は以下の通りである.

$$(\text{FIX}) \qquad \frac{\mathcal{A}\{f:\tau_1\rightarrow\tau_2, x:\tau_1\}\rhd M\ :\ \tau_2}{\mathcal{A}\rhd fix\ f(x:\tau_1).M\ :\ \tau_1\rightarrow\tau_2}$$

この拡張は型システムの安全性, すなわち定理2を保存する.

4.2 再帰的データ型の導入

型付きラムダ計算では, 原子型以外の型構成子として直積型 $(\tau\times\tau)$ と直和型 $(\tau+\tau)$ のみを含んでいた. しかしながら, 実際のプログラミングでは, リストや木構造などの種々のデータ型を扱う必要がある. これら種々のデータ型を定義する一般的な構成法として, 再帰的データ型の定義機構を導入する.

まず, ただ一つのデータ $*$ を値として含む型である $unit$ 型を導入する. そのために, ラムダ式の集合と型の集合を

$$M ::= \cdots \mid *$$

$$\tau ::= \cdots \mid unit$$

と拡張し, 以下の型付け規則を追加する.

$$(\text{UNIT}) \qquad \mathcal{A}\rhd * : unit$$

ラムダ式 $*$ に対する変換規則はない. $unit$ 型は種々の再帰的データ型を定義する上でのいわば "出発点" の役割を果

たす.

再帰的データ型とは，自分自身を部分構造として含むような型である．そのような型を定義するための一つの方法は，定義のなかで“定義された結果の自分自身”を指し示す名前を使用することである．今 t をそのような名前，τ を t を含む型とし，自分自身を部分として含むような再帰的な型を $\mu t.\tau$ と書くことにする．再帰的に定義された型を導入するために，型の集合を“仮に”以下のように拡張する．

$$\tau ::= \cdots \mid t \mid \mu t.\tau$$

ただし τ は t 自身ではないものとする．型は，以上の文法で与えられるものの中で，自由な型変数を含まないものと定義する．例えば $\mu t.t \to t$ は型であるが，$int \times t$ や $\mu t.t \to t'$ は型ではない.

型 $\mu t.\tau$ は直感的には

$$t = \tau$$

のような等式の解に対応する．しかし型が有限の大きさであれば，τ が t を含まない特殊な場合を除き，このような等式の解は存在せず，従ってこの等式を実現するようなラムダ式とその変換規則を導入するのは困難である．この問題を回避する一つの方法は，上等式を実現する代わりに，同型関係

$$\mu t.\tau \cong \tau[(\mu t.\tau)/x]$$

を実現するラムダ式構成子を導入することである．ここで $\tau[(\mu t.\tau)/x]$ は τ の中の t を $\mu t.\tau$ で置き換えて得られる型である．この同型射像を実現するために，ラムダ式の集合を以下の様に拡張する.

$$M ::= \cdots \mid up(M) \mid dn(M)$$

これら二つの構成子が実際に同型写像として振舞うことを保証するために，以下の書き換え規則を定める.

$$(\mu_1) \qquad uu(dn(M)) \stackrel{\mu_1}{\Longrightarrow} M$$

$$(\mu_1) \qquad dn(up(M)) \stackrel{\mu_1}{\Longrightarrow} M$$

さらに評価分脈の定義を以下の様に拡張する.

$$E[\,] ::= \cdots \mid up(E[\,]) \mid dn(E[\,])$$

この二つの構成子を利用すれば，$\mu t.\tau$ 型のデータは，実質的には，$t = \tau$ が成り立つデータとして扱うことができる.

これら構成子に対する型付け規則は以下の通りである.

$$(\text{UP}) \qquad \frac{\mathcal{A} \rhd M : \tau[(\mu t.\tau)/t]}{\mathcal{A} \rhd up(M) : \mu t.\tau}$$

$$(\text{DN}) \qquad \frac{\mathcal{A} \rhd M : \mu t.\tau}{\mathcal{A} \rhd dn(M) : \tau[(\mu t.\tau/t]}$$

以上の拡張を直和型及び直積型と組み合わせることによって，ほぼすべてのプログラミング言語のデータ構造が定義可能である．例えば整数 (int 型データ) を要素とするリスト型は以下の様に定義できる.

$$list(int) = \mu t.unit + (int \times t)$$

この定義は，整数のリストは「単純な定数 $*$ か，整数とリストの組である」ことを表しており，通常のリストの定義と一致する．以降の説明では $list(int)$ を $\mu t.unit + (int \times t)$ の略として用いる．次に，空リスト nil 及びリスト処理関数 car，cdr，$cons$ を定義することを考える．このうち car と cdr は，引き数が空リストの時はかえす値が無い．通常のプログラミング言語ではエラーを通知するが，ラムダ計算の理論でのエラーの取扱いは本講座の範囲を越えるので，ここではそのような場合は計算停止しないものとする．型 $\tau \to \tau$ を持つ停止しない関数は以下の様に書ける.

$$Undef_\tau = fix\ f(x:\tau).f\ x$$

$$
\begin{aligned}
car(cons(1,nil)) &= car((\lambda x : int \times list(int).up(inr(x)))(1, up(inl(*)))) \\
&\longrightarrow car(up(inr(1, up(inl(*))))) \\
&= (\lambda x : list(int).case\ dn(x)\ of\ inl => \lambda x : int + list(int).Undef_{int}\ 0, \\
&\qquad\qquad inr => \lambda y : int \times list(int).y.1) \\
&\quad (up(inr((1, up(inl(*)))))) \\
&\longrightarrow case\ dn(up(inr(1, up(inl(*)))))\ of\ inl => \lambda x : int + list(int).Undef_{int}\ 0, \\
&\qquad\qquad inr => \lambda y : int \times list(int).y.1 \\
&\longrightarrow case\ inr(1, up(inl(*)))\ of\ inl => \lambda x : int + list(int).Undef_{int}\ 0, \\
&\qquad\qquad inr => \lambda y : int \times list(int).y.1 \\
&\longrightarrow (\lambda y : int \times list(int).y.1)(1, up(inl(*))) \\
&\longrightarrow (1, up(inr(*))).1 \\
&\longrightarrow 1
\end{aligned}
$$

(inr,inl の型の添字は省略した.)

図 9: 再帰的データ型を含む関数評価の例

するとこれら 3 つの関数は以下の様に定義できる.

$$
\begin{aligned}
nil &= up(inl_{unit+list(int)}(*)) \\
car &= \lambda x : list(int).case\ dn(x)\ of\ inl => \lambda x : int + list(int).Undef_{int}\ 0, inr => \lambda y : int \times list(int).y.1 \\
cdr &= \lambda x : list(int).case\ dn(x)\ of\ inl => \lambda x : int + list(int).Undef_{int}\ 0, inr => \lambda y : int \times list(int).y.2 \\
cons &= \lambda x : int \times list(int).up(inr(x))
\end{aligned}
$$

以上の定義に対して, 以下の型判定が導出できることが確かめられる.

$$
\begin{aligned}
\emptyset &\ \triangleright\ nil : list(int) \\
\emptyset &\ \triangleright\ car : list(int) \rightarrow int \\
\emptyset &\ \triangleright\ cdr : list(int) \rightarrow list(int) \\
\emptyset &\ \triangleright\ cons : int \times list(int) \rightarrow list(int)
\end{aligned}
$$

さらに, ラムダ式の評価に関する拡張された定義により, car, cdr, $cons$ は通常のプログラミングで実装されているこれら関数と同じ動作をすることを確かめることができる. 簡単な評価の例を図 9に示す. この関数と再帰的関数定義を使えば任意のリスト処理関数を書くことができる. 例えば整数のリストの長さを求める関数は以下のように定義できる.

$$length = fix\ l(x : list(int)).case\ (dn(x))\ of\ inl => \lambda x : unit.0, inr => \lambda x : list(int).plus(1, l(x))$$

リストに限らず, スタックやキュー, 種々の木構造等のデータ構造を以上の再帰的データ構造の定義機構を使って定義することができる.

5 多相型と型推論

以上のように拡張された型付きラムダ計算は, Pascal などの静的型システムを持つ言語のモデルと考えることができる. 静的な型システムを持つ言語は, プログラムの型エラーのすべてをコンパイル時に取り除くことにより, プログラムの信頼性の向上が期待でき, また, 実行時の型チェックが不要になるため効率のよいコードへのコンパイルを可能にするといった利点がある. しかし, 静的に型付けられた言語は柔軟性に欠け扱いにくいとの批判があるのも事実である. 特に問題とされる点は, 関数の型が静的に決定されているため, Lisp などの型なし言語で普通に使

われている汎用の手続きが書けないということである．さらに，型システムが要求する変数の型宣言は，プログラムのドキュメンテーションとしては有用ではあろうが，プログラマにとっては自明なことが多く繁雑である．この二つの問題の克服が，プログラミング言語の型システムの研究の重要なテーマであった．それらの研究を通じて達成された成果の中で特に重要なものは，プログラムの多相性 (polymorphism) の理論と 静的型推論 (static type inference) の技術である．

多相性とは，一つのプログラムが種々の型に適用可能であるという性質を表す型理論における用語である．例えば，リストの長さを計算する Lisp プログラム

```
(defun length (l)
  (if (null l) 0
    (+ 1 (length (cdr l)))))
```

は，("CRU" "Haskel" "Miranda" "ML") や (() (1 2 3) (1)) などの型の異なる種々のリストに適用可能であり，この意味で多相的である．プログラムの多相性を表現し得る型を，多相型 (polymorphic types) という．一方，静的型推論は，型宣言を含まないプログラムからその型を自動的に推論する技術である．多相型と型推論の二つの技術によって，Lisp などの型無し言語に近い柔軟性と，静的型システムの利点とを合わせ持った型システムの設計の基礎が築かれた．これらの理論はプログラミング言語 ML で実用化されている．ML では，上記の `length` のような，型宣言を含まない関数の定義から，その正確な多相型が自動的に推論される．例えば `length` 関数は ML では

```
fun length l = if null(l) then 0
               else 1 + length(cdr(l))
```

のように書かれるが，この定義に対して型システムは以下のような型情報を推論する．

$$\texttt{length} \ : \ \forall t.list(t) \rightarrow int$$

この型情報は，`length` が，任意の型のリストに対して整数型のデータを計算する多相関数であることを表現している．この技術の有用性は広く認識され，他のいくつかの言語でも採用されている．

以下簡単のためにデータ型や再帰的関数定義の取り扱いは省略し，以下の文法で与えられる ML の核言語を用いて，ML の型システムの基本原理を説明する．

$$e ::= c^{\tau} \mid x \mid \lambda x.e \mid e\ e \mid let\ x = e\ in\ e$$

単純な型付きラムダ計算との違いは，ラムダ抽象が型指定を含まないことと，多相型を持つプログラムの定義機構である let 式が導入されていることである．ラムダ抽象および let による名前の束縛は，関数定義ばかりでなくプログラム言語での種々の名前の定義のモデルでもあるため，これらの束縛機構が型指定を含まないという特徴は，一切の型指定無しにプログラムが書けることを意味する．型指定を含まないにもかかわらず，ML の型システムは，後に述べる型推論機構により，任意のプログラムを完全に型チェックすることができる．ラムダ抽象が型指定を含まないことはまた，ラムダ式が複数の型を持ち得ることを意味する．`let` 構文は，そのような複数の型を持つラムダ式を，文脈に応じた種々の型のプログラムとして，すなわち多相型を持つプログラムとして使うためのものである．例えば，

$$let\ id = \lambda x.x\ in\ id(1)\ \cdots\ id(\text{"}John\text{"})\ \cdots$$

において，2番めと3番めの id はそれぞれ $int \rightarrow int$ および $string \rightarrow string$ 型の関数として使われている．

以上定義した式の型システムを定義する．理解を容易にするために，let 構文を含まない式の型システムをまず定義し，後にそれを let 構文に拡張する．let 構文を含まない式の取り得る型の集合は，単純な型付きラムダ計算の型の集合に，メタ変数 t で表される型変数を加えてものである．本説明では，直積型と直和型は省略しているので，その集合は以下の文法で与えられる．

$$\tau ::= t \mid b \mid \tau \rightarrow \tau$$

型付け規則も単純な型付きラムダ計算の場合と同様である．let 式を除く ML の核言語の型付け規則を図 10に示す．

$$\text{(CONST)} \quad \mathcal{A} \rhd c^{\tau} : \tau$$

$$\text{(VAR)} \quad \mathcal{A} \rhd x : \tau \quad \text{if } \mathcal{A}(x) = \tau$$

$$\text{(ABS)} \quad \frac{\mathcal{A}\{x : \tau_1\} \rhd e_1 : \tau_2}{\mathcal{A} \rhd \lambda x.e_1 : \tau_1 \to \tau_2}$$

$$\text{(APP)} \quad \frac{\mathcal{A} \rhd e_1 : \tau_1 \to \tau_2 \quad \mathcal{A} \rhd e_2 : \tau_1}{\mathcal{A} \rhd e_1\ e_2 : \tau_2}$$

図 10: ML の核言語の型システム (1)

ラムダ式関数式 $\lambda x.e$ の変数の型が指定されていないため，与えられた式は一般に複数の型を持ち得る．例えば任意の型 τ に対して型判定 $\{x : \tau\} \rhd x : \tau$ がなり立つから，規則 (ABS) によって，任意の型 τ について，

$$\emptyset \rhd \lambda x.x : \tau \to \tau$$

が成り立つ．与えられた式に対して複数存在する型判定の中で，主要な型判定 (principal typing) と呼ばれる特に重要な型判定が存在する．

定理 4 与えられた式 e が型を持てば，主要な型判定 $\mathcal{A} \rhd e : \tau$ が存在し，e に対するその他の可能な型判定はすべてその主要な型判定の型変数を適当な型で置き換え，必要に応じて型環境に前提を付け加えることによって得ることができる． ∎

ある式の主要な型判定は，その式が持ち得るすべての型判定を代表する型判定である．例えば，

$$\emptyset \rhd \lambda x.x : t \to t$$

は式 $\lambda x.x$ の主要な型判定であり，この式の持つすべての型判定は型変数 t を適当な型で置き換え，(不必要な) 型宣言を型環境に追加することによって得ることができる．さらに重要なことに，主要な型判定を自動的に計算するアルゴリズムが存在する．

定理 5 任意の式 e に対して，もし e が型を持てばその主要な型判定を計算し，もし型を持たなければエラーを報告するアルゴリズムが存在する． ∎

アルゴリズムのおおよその構造は以下の通りである．

1) 自分自身を再帰的に用いて，式を構成する部分式の型判定を計算する．
2) 式の構造に対応する型付け規則に従い，部分式の型判定の間に成り立つべき条件を等式の集合として求める．
3) 単一化アルゴリズムを用いて，等式集合を満たす最も一般的な解を，型変数の置換として求め，その置換を適用し型判定を求める．

例として，式 $f\ x$ の主要な型判定を計算する過程を考えてみよう．まず，部分式 f と x の主要な型判定を計算する．

$$\{f : t_1\} \rhd f : t_1$$
$$\{x : t_2\} \rhd x : t_2$$

求める式 $f\ x$ は関数適用であり，その型判定を導出する規則は (APP) である．そこで，以上 2 つの型判定から，規則 (APP) の二つの前提に当てはまる最も一般的な型判定を作ることを考える．まず型環境が共通でなければならないことから，以下の型判定を得る．

$$\{f : t_1, x : t_2\} \rhd f : t_1$$
$$\{f : t_1, x : t_2\} \rhd x : t_2$$

$$(\text{GEN}) \qquad \frac{\mathcal{A} \rhd e : \sigma}{\mathcal{A} \rhd e : \forall t.\sigma} \quad t \text{ not free in } \mathcal{A}$$

$$(\text{INST}) \qquad \frac{\mathcal{A} \rhd e : \forall t.\sigma}{\mathcal{A} \rhd e : \sigma[\tau/t]}$$

$$(\text{LET}) \qquad \frac{\mathcal{A} \rhd e_1 : \sigma \quad \mathcal{A}\{x : \sigma\} \rhd e_2 : \tau}{\mathcal{A} \rhd let\ x = e_1\ in\ e_2 : \tau}$$

図 11: ML の核言語の型システム (2)

さらに，規則 (APP) の前提となる型判定の型の関係から，t_1, t_2 の間には，ある型 τ に対して

$$t_1 = t_2 \to \tau$$

の関係が成り立たなければならない．この条件のもとで，式 f の結果の型は τ となる．この条件を満たす最も一般的な解は，t_3 を新しい型変数として，t_1 を $t_2 \to t_3$ に置き換える置換である．以上から，式 f の主要な型判定が

$$\{f : t_2 \to t_3, x : t_2\} \rhd f\ x : t_3$$

と求まる．

以上の性質は Hindley によってラムダ計算と等価なコンビネータ論理の分野で発見されていた．ML の型システムの基礎となった Milner の型推論システムは，以上の型推論システムを，多相型と，多相型を持つ式の利用機構である `let` 宣言を導入し拡張したものである．その型理論的な定義は Damas と Milner によって与えられた．彼らはまず，今まで使用した型集合に加えて，以下の生成規則で与えられる多相型の集合を定義した．

$$\sigma ::= \tau \mid \forall t.\sigma$$

多相型 $\forall t.\sigma$ は，σ の中に現れる型変数 t を適当な型で置き換えて得られる種々の型の式として使用可能な汎用性ある式の型である．多相型を持つ式の定義とその利用に関する規則は図 11で与えられる．規則 (GEN) は，型環境の中で特に仮定されていない型変数を抽象化して，多相型を持つ式を作る規則である．規則 (INST) は，式の持つ多相型の中の型変数を適当な型で置き換え，種々の型の式として使用可能にする規則である．規則 (LET) は，多相型を持つ式 e に名前 x をつけ，それを種々の型の式として使用するための規則である．

以上の拡張によって，多相型を持つプログラムの定義と使用が可能になる．例えば

$$let\ ID = \lambda x.x\ in\ \cdots\ (ID\ 3)\ \cdots\ (ID\ "ML")\cdots$$

の形をした式において，ID の型は式 $\lambda x.x$ の持つ主要な型 $t \to t$ の型変数を抽象して得られる多相型 $\forall t.t \to t$ となり，$ID\ 3$ では t を int で置き換えた $int \to int$ 型の関数として，また $ID\ "ML"$ では t を $string$ で置き換えた $string \to string$ 型の関数として使われている．

$let\ x = e_1\ in\ e_2$ 式の型推論は以下の様に行なえば良い．

1) まず e_1 の主要な τ 型を求め，その中の型変数の中で型環境に現れないものを $\forall$ で束縛し，それを x の型 σ として記録する．

2) e_2 の型推論を以前同様に行なう．ただし，e_2 の中で x が現れたら，その型は，ステップ 1) で記録した型 σ の中の型変数の内，$\forall$ で束縛された型変数をすべて新しい型変数で置き換えてえられる型として推論を進める．

先に説明した型推論アルゴリズムに，以上の let 式の型推論を加えたものが，Milner による ML の型推論アルゴリズム $\mathcal{W}$ である．そのアルゴリズムは，本節で定義した Damas と Milner による多相型システムに関して完全であること，すなわち，アルゴリズムは常にプログラムの持つ主要な型を推論することが証明されている．

6 現在の研究動向

より高水準でしかも安全なプログラミング言語の設計と実装の理論を確立する為に，現在ラムダ計算及びその型システムを，より柔軟で強力な計算を表現できるように拡張する研究が盛んに行われている．最近よく研究されているトピックには以下のようなものが含まれる．

- 値の更新の概念の導入．
  式の置き換えに基づくラムダ式の計算のモデルでは，通常の手続き型言語で行われている変数の更新やポインターデータなどは表現できない．これらの機能の無際限な使用はプログラムの理解しやすさを損なうとの批判があるが，アルゴリズムによってはそれらの機能があるとプログラムの理解しやすさやその効率が大幅に向上する場合がある．現在ラムダ計算の基本的な枠組みのなかで，これら非関数的な機能をうまく導入する研究が行われている．

- 大域的な制御構文の導入．
  プログラムの制御の流れが関数の呼出によって制御されるラムダ計算のモデルでは，大域的なジャンプなどを直接表現することはできない．しかしながら，前項の場合同様，例外的な事態の処理等，大域的な制御機構があるとプログラムの理解しやすさやその効率が大幅に向上する場合がある．現在ラムダ計算の基本的な枠組みのなかで，これら非関数的な機能をうまく導入する研究が行われている．

- 多相型と型推論の種々のデータ構造への拡張
  ラベル付きレコードやラベル付きバリアントなどプログラミングでよく使用されているデータ構造の中には，従来の多相型と型推論の理論が適用できないものがある．多相型と型推論の理論をこれらデータ構造へ拡張できれば，より柔軟な型システムが実現できると期待される．

- オブジェクト指向プログラミングとの融合
  オブジェクト指向プログラミング言語と呼ばれる一連のプログラミング言語が近年注目を集めている．これら言語はある特定のプログラミングスタイルを共通にもっているが，十分に洗練された計算モデルをもっているとはいえず，現状では，上記のいずれかの既存のプログラミング言語に望ましいと思われる機能を追加したものと見なすのが妥当と思われる．時間が許せば，関数型言語の枠組みの中でオブジェクト指向プログラミングの機能を表現する研究も紹介する．

講座では時間が許す限り，種々のトピックを紹介する．