技術者／プログラマのための ラムダ計算、論理、圏 第2回

檜山 正幸 (HIYAMA Masayuki)
2009年2月19日 (木曜)
18:00開始

今日の予定（おおよそ）

1. まえおき／まえせつ -- 10分
2. 大きなラムダと小さなラムダに慣れる 40分 (50分)
3. 計算と関数の関係を理解する 20分 (70分)
4. 休憩 -- 20分 (90分)
5. 計算の世界を探索 -- 40分 (130分)
6. 不可能性の証明 -- 40分 (170分)

状況により、この予定は（ときに大幅に）変更されるかも知れません

全体（3回）の目標

ラムダ計算、自然演繹による推論、デカルト閉圏の三位一体を知る。

今回は、大きなラムダ計算と小さなラムダ計算に慣れ、

計算の世界（the world of computation）を体感し、

計算の限界、ある主の判断・決定の不可能性を納得しましょう。

前回のテーマはこれだった

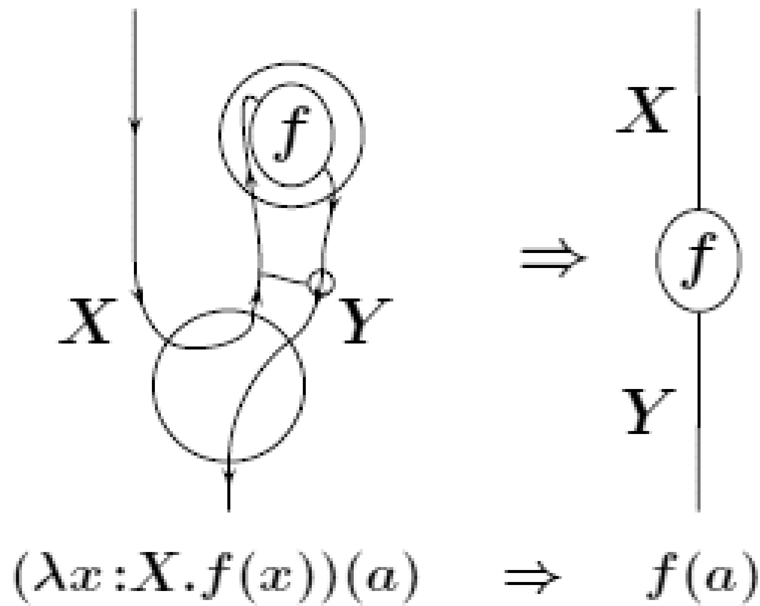
スノーグローブ

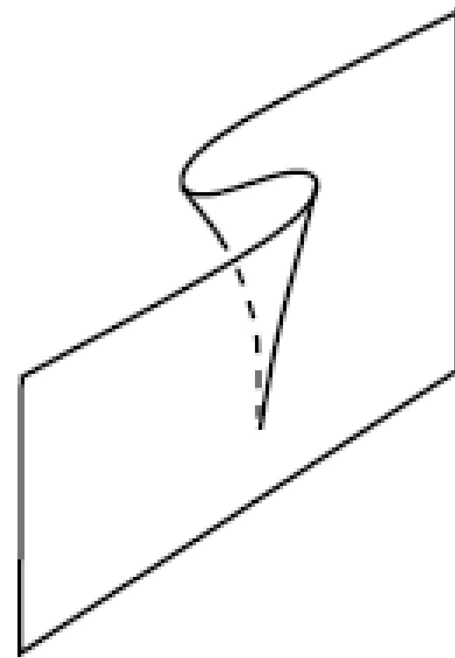
スノーグローブ現象と無縁ではいられない私達

認識と了解のタフネスが必要

これもテーマだった

バエズの絵





これは重要と言った

Eがとある関数コードの実行エンジンとして、

fを、Eのコード体系＝Eの機械語によってコード化した関数コードを$f^{\wedge E}$ と書く。

ただし、fの引数の1つは$f^{\wedge E}$のパラメータとして残るので、実引数を具体化した結果 $f^{\wedge E}(x)$ が関数コード。

- $f(x, y) = E(f^{\wedge E}(x), y)$

基本等式。通常は、Eを固定して単に $f^{\wedge E}$ と書く。

いよいよラムダだ、で前回終わったのだった

- λ -- 小文字のラムダ（「はいる」とは違う）
- Λ -- 大文字のラムダ（論理ANDとは別物）
- ^ -- ラムダに似ている（ハット、サーカムフレックス）
- < > -- 寝ころんだラムダ2つ＝ラムダ括弧

以下、前回と少し変更された部分があるので注意。

大きなラムダ式とその計算

インフォーマルなラムダ計算

- これは、我々が日常的に使う道具
- 紙やホワイトボードに書くもの
- 読み書きと計算は人間がする
- 経験と直感（直観？）が基盤
- 生きている私（あなた）が、世界を記述し、推論を行うために使う
- 大きなラムダ式は、関数を直接表現する、関数そのものの代理
- しばしば、大きなラムダ式と関数そのものが同一視される（区別しなくなる）

大きなラムダ式 (1)

例で示す。

1. <x, y| x + y>
2. <x| 3 × x + 2>
3. <a| 3 × a + 2>
4. <a, x, y| a × x + 2 × y + 1>
5. <| 5>

- <引数変数並び | 変数を含むかも知れない式> の形
- 本体（ボディ）の式に、引数並びに出てこない変数があるのはNG
- 原則的に、入れ子は許さない（伝統的数学の立場なら許すが）
- Roland Backhouse（http://www.cs.nott.ac.uk/~rcb/）さんあたりが使ってます

質問: <| 5> は 5 と同じか？

大きなラムダ式 (2)

- fが既に存在する（fと名付けられた）関数のとき、f = <x| f(x)> と書いてよい
  例： sqrt = <x| sqrt(x)>
- fが2引数なら f = <x, y| f(x, y)>、3引数以上も同様
- f = <x, y| x × x + y × y> のようにして、式で定義される関数に名前を与えてもよい

等号の意味

- <x| x × (x + 1)> = <x| x × x + x> は成立する
- そもそも関数fとgが等しいとは、許されるどんな具体的な値aに対しても f(a) = g(a)
- 箱（マシン）中身が見えていても、それは考慮しない
- fとgが等しければ、fのグラフとgのグラフは等しいし、逆も真

式は関数か、関数は式か

- 式は関数を定義する
- すべての関数が式で定義されるとは限らない
- だが、計算可能な関数は（広義の）式で定義できる
- 同じ式は同じ関数を定義する
- 違う式が同じ関数を定義することもある
- 式を見て関数の同一性（等しさ）を判断できるか？（後で問題にする）

大きなラムダ式の計算規則

- アルファ規則 <x| f(x)> = <y| f(y)>（ボディが式であってもよい）
- ベータ規則 <x| f(x)>(a) = f(a)
- イータ規則 <x| f(x)> = f

いずれもインフォーマルラムダ計算の規則。

経験と直感で納得。自明と言える。

小さなラムダ式とその計算

フォーマルなラムダ計算

- 関数のコード化に使う言語
- 関数コード実行エンジンのマシン語（プログラミング言語）
- 人間への指令ではなくて、マシンへの指令
- データとして扱う
- 数学的に厳密な定義に基づく
- 計算するのは（仮想的でも抽象的でも）実行エンジン
- 人間が計算するときもあるが、それは感情移入

小さなラムダ式の構文

- 定数リテラル： 1, 2, trueとか適当に
- 関数記号（関数ではない、むしろインストラクション）：A, M とか適当に
- 変数（ほんとは型付きだが、今はあまり型に注意してない）
- 適用記号・
- ラムダ記号λ
- ピリオド、カンマ、括弧

定数リテラルと関数記号をアトム記号、あるいは単にアトムとも言う。
どのくらいアトムを使うかは目的によりけり。

半分フォーマルな小さなラムダ式

- <x, y|2 × x + y + 3> 大きいラムダ式
- λx.λy.(A▪(A▪(M▪2▪x)▪y)▪3) 小さいラムダ式
- λx.λy.(2 × x + y + 3) 半分フォーマルな小さいラムダ式

セミフォーマルでは、小さなラムダ式の内部が人間可読(つうか読みやすく)に書いてある。大きなラムダ式との違いは:

1. 文字λを使う。
2. 入れ子を許す。
3. 自由変数を許す。
4. 常に1引数。

3種のラムダ式

種類	誰のため	目的
インフォーマルなラムダ式	人間	世界を記述
セミフォーマルなラムダ式	箱に入った人間や妖精	箱のなかで計算
フォーマルなラムダ式	記号計算をするマシン	マシンによる計算

ラムダ抽象＝ラムダオペレータ

- 例： Λ<x, y| 2 × x + y> = <x| λy.(2 × x + y)>
- 大きなラムダのボディ部の変化 2 × x + y → λy.(2 × x + y) をラムダ抽象と呼ぶことが多い
- だが、ラムダ抽象は大きなラムダ式に働く操作である！ 関数から、関数コードジェネレータを生み出す
- 伝統的数学の立場では、関数から、関数ジェネレータ＝高階関数 <x| <y| 2 × x + y>> を生み出す

ラムダ抽象の絵

描こう。

練習とか

適当にアドリブで

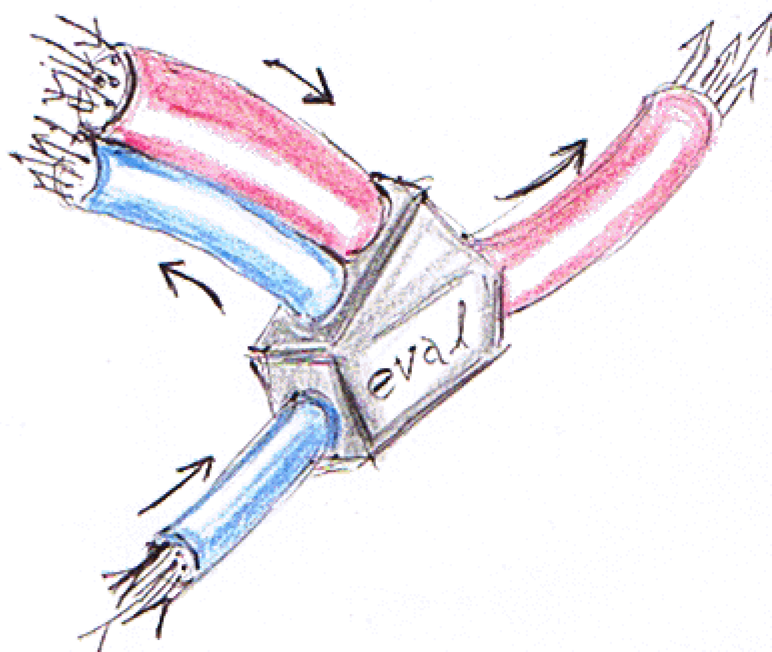
基本等式

- $\Lambda(f) = f\hat{}$
- $Exec(f\hat{}\,(a), b) = f(a, b)$

いろいろなバリエーションがある

ラムダ抽象の絵もっと

こりゃ向きが違うが、



スノーグローブとベータ変換

- 我々が大きなラムダ式の計算に行うベータ規則による計算
- 基本等式を使ったベータ変換（これも人間の視点）
- 関数コードの実行エンジンが行うベータ変換
- エンジンの変種、Exec, Apply, Evalでもベータ変換が微妙に違う

絵の描き方

口頭とホワイトボードで。

関数と計算機

関数だけで計算機の動作を記述・説明できるの？

はい、できますよ。

メモリー状態は1つの整数

どんな巨大なメモリーだって、
その状態はビット列

- すべてのビット列 ＝ 1以上の整数
- 都合で0も入れておこう（たいして意味はない、1ズラしてもいいし）
- N = {0, 1, 2, 3, ...}

ビット列	2進数	整数(10進)
''	1	1
'0'	10	2
'1'	11	3
'00'	100	4
'01'	101	5
'10'	110	6
'11'	111	7

プログラムは、走る前の状態から走り終わった後の状態への遷移を引き起こす。だから ...

関数と部分関数

- N = {0, 1, 2, 3, ...}
- f:N → N
- fは（通常の習慣では）全域関数
- いつでも全域がいいとは限らない
- f(x) := x/2
- f(x) := 1/x
- メモリ状態の表現
- 例外や無限走行
- 大きなkにより {0, 1, ..., k}でも充分

- f:N ⊃→ N
- fは部分関数
- 計算で値を求める関数は部分関数

コードとデータ

Eが計算機だとすると:

- E:N, N ⊃→ N
- E:(N^N, N) → N でいいか？
- やっぱり、E:(N^N, N) ⊃→ N
- もっと正確に言うと …

奇妙な不等式

- $a^a \leqq a$
- $a \times a \leqq a$

- $A^A \subseteq A$
- $A \times A \subseteq A$

- $(N^N) \subset N$（＝ではない）
- $N \times N \subseteq N$

もう一度記号の確認

- N -- 自然数の全体
- (N^N) -- 関数コードの全体
- f:X⊃→Y -- 部分関数
- <X⊃→Y> -- 部分関数の全体（ここだけの記法）
- f:X⊃→Y ⇔ f∈<X⊃→Y>

注意：部分関数の圏のなかで話をするなら、f:
X⊃→Y を f:X→Yと書いても差し支えない。

計算の世界を記述しよう

3種のラムダ式を道具として、
基本等式を念頭におきながら、
計算の世界を記述しよう。

計算の世界

- データ領域と関数（計算可能な部分関数）が構成する世界
- モノ（データ）と働き（関数）を徹底的に区別することにより、モノと働きを同一視できるメカニズムを解明する
- サブルーチンが関数なのではない！
- コンピュータの計算は関数だと思ってよい
- 手続きや副作用も関数として表現可能
- ブートストラップも関数
- プログラム断片も関数
- その他、計算行為や計算現象は何でも関数

もっと広く考えよう

- 計算の世界を包括的に眺める、そこで起きている計算現象を捉える
- 自然も生物も人間も機械も社会も計算している
- 機械の計算、力学（機械学）的、電気的（回路）、電子的
- 化学反応やDNAやらでも計算できる
- データ領域 -- 数値や記号的表現（文字、文字列、ツリーなど）、紙に書いたナニカ、音声、模様や絵なども
- 人間可読／可視／可聴な表現もデータ領域に入る
- データ領域には型が付く
- 型は運用上の約束事であり、「みなし方」のヒントまたは強制である（圏論では常に強制と考える）

計算の世界の前提あるいは仮説

- 自然数領域やその部分領域がある
- タプルが作れる
- 色々なエンジンがある
- そのなかには万能なエンジンがある
- エンジンに関しては基本等式が成立する
- 万能なエンジンに対して、エンジン自身を含めてすべての関数をコンパイルできる

最後の仮説は、スノーグローブ現象が起きていることを認めること。

人間による計算

- 人間可読な表現を人間が解釈計算することも「基本等式」
- 自然界の関数fを、人間可読表現f^H として記述することもイデアルコンパイル
- 解釈計算する人間をHとすると、HをエンジンMによってコード化すると H^M
- H^Mは人間可読な表現（H-関数コード）を、機械語コードH^M(s)に直す

その他いろんな例

- トランスレータ、エミュレータ
- スタンドアロンなコンパイラ
- クロスコンパイラ
- コンパイラプログラム
- リンカーみたいな
- 高級言語のネイティブコード化

不可能性の証明

それは無理だ！

できるわけない！！

なにが不可能なのか

人間や計算機が実行できるアルゴリズムにより、

1. プログラムが停止するかどうかを確実に判断すること
2. プログラムが全域的かどうかを確実に判断すること
3. 2つのプログラムが外延的に同値かどうかを確実に判断すること

ここで、「プログラム＝計算可能な関数」と思ってよい。
外延的に同値とは「部分関数として等しい」こと。

このプログラム（関数）は、
この引数で停止する

神様（超越者）の関数 GOOD(f, a) は、

- 関数fに引数aを渡すと停止して戻り値を出力するなら true
- 関数fに引数aを渡すと例外となるか無限走行して戻り値を出力しないなら false

を返す。GOOD: <N ⊃→N>, N → B

GOODが人間により記述され、コンパイルされ、ライブラリとして使えるようになったと仮定する。その“この世の関数”を good とする。

fの関数コードをビット列または整数とみなしたものをφとして、

・good(φ, a) = GOOD(f, a)

good: N, N ⊃→ B

関数sと値s(σ)

```
function forever() {
 while(true) {
  ; // do nothing
 }
 return 0;
}

function s(x) {
 if (good(x, x)) {
  forever();
 } else {
  return 0;
 }
}
```

sの関数コード（コンパイル済みイメージ）をビット列または整数とみなしたものをσとすると、値 s(σ) がどうなるか？

値s(σ)を考える

確認 : good(σ, σ) = GOOD(s, σ) は定義される。
　　　値は、trueかfalseのどちらか。

good(σ, σ) = true のとき :

1. GOOD(s, σ) = true
2. s(σ) は停止して値を持つ
3. if文のthen部が実行される
4. 無限走行する

good(σ, σ) = false のとき :

1. GOOD(s, σ) = false
2. s(σ) は例外か無限走行して値を持たない
3. if文のelse部が実行される
4. 値は0

んっ？

これはどうということ？

おまえの言うことが正しいなら、
おてんと様が西から昇るよ。

その仮定が正しいとするなら、
ありえない事が起きる。

おてんと様は西からは昇らない。ありえない事は起きない。
よって、おまえの言うこと／その仮定は正しくない。

もし全域性が判断できたら

- プログラム（関数コード）は順番に列挙できる。
- 全域なものだけを選び出せる。
- 全域なものを順に並べて $f_0, f_1, f_2, \ldots$ という列を作れる。
- $g(n) := f_n(n) + 1$ で定義されるgを作れる。
- gは計算可能な全域関数である

それはないよ。
よって、おまえの言うこと／その仮定は正しくない。

もし関数の等しさが判断できたら

確認: 部分関数fとgが等しいなら、その有効定義域も等しい。

与えられたfから、g(x) := (f(x) + 1)/(f(x) + 1) としてgを作ると:

1. f(x)が定義されているなら、g(x)も定義されて g(x) = 1
2. f(x)が定義されていないなら、g(x)も定義されない。

one := <x:N | 1 :N> として、g = one が判断できれば、

1. g = one ならば gは全域
2. g = one でないならば gは全域でない

あれ？

我々は知りえない

1. 与えられた2つの関数（計算可能な部分関数）が等しいかどうかを確実に知る実行可能な方法はない。
2. 与えられた関数（計算可能な部分関数）が全域的に定義されているかどうかを確実に知る実行可能な方法はない。
3. 与えられた関数（計算可能な部分関数）に具体的な値を渡して停止するか（戻り値が返るか）どうかを確実に知る実行可能な方法はない。

我々は超越的な関数の存在を夢想することも確信することもできる(?)が、超越的な関数を計算する方法を持たず、今後なにがあってもどうあがいても計算できない。

我々は何を知りえないかを知りうる

不思議だ