プログラム意味論班 活動報告書

立命館コンピュータクラブ
2019 年度前期プロジェクト活動

2019 年 8 月 8 日

松本　幸大[*1]
小泉　孝弥[*2]
山口　流星[*2]
西見　元希[*1]
田中　聖也[*3]
深田　紘希[*3]
堀田　隆成[*3]
八木田　裕伍[*3]

[*1] 情報理工学部 情報理工学科 セキュリティ・ネットワークコース 2 回生
[*2] 理工学部 数理科学科 3 回生
[*3] 情報理工学部 情報理工学科 1 回生

1　はじめに

文責: 松本　幸大

このプロジェクトの目的はプログラム意味論，特に操作的意味論のゼミ形式での学習を通じて，普段触れているプログラミング言語のもつ「意味」に関する深い見識を身につけることである．活動はゼミ担当者を設定したうえで週 2 回行い，1 日目に来られない班員は 2 日目に参加することとしていた．ゼミは「プログラミング言語の基礎概念 (五十嵐淳 著，サイエンス社)」の内容に沿って行った．

2　第 1 週　自然数の加算・乗算・比較

文責: 松本　幸大

初回は，主に導出システムと BNF，抽象構文木について学んだ．導出システムは，議論の対象に対する様々な「判断 (judgement)」を「推論規則 (inference rule)」に従って導くための記述体型である．

3　第 2 週　メタ定理と帰納法による証明

文責: 山口　流星

3.1　導出システム

前回の活動では，導出システムとは何かについて学んだ．導出システムは，ある判断を相当する推論規則に従って導出するものであり，次の 4 つの導出システムを学んだ．

- ペアノ自然数の加算，乗算の導出システム Nat
- ２つのペアノ自然数の比較をする導出システム CompareNat1,2,3
- 算術式の評価をする導出システム EvalNatExp,
- 算術式の簡約を行う導出システム ReduceNatExp

導出システムは，推論規則から判断が導かれることから，推論規則を変更すると，それによって導出できる判断が変わる．これによって，内容は正しいが，導出ができない導出システムや，内容が誤りであることを導出できる導出システムなどを形成できる．

3.2　判断の一般的性質

3.2.1　メタ定理

メタ定理とは，具体的な導出システム上で，数学的証明がなされた判断の一般的性質のことである．つまり，ある導出システムにおいては，常に成り立つ判断の性質があるということである．ここで表す数学的証明は，

帰納法を指す．以下にメタ定理の例を示すが，これは俗に加算の加法性と呼ばれるものであることが分かる．

導出システム Nat 上で，任意のペアノ自然数 n_1, n_2, n_3 に対して，次の判断は同値である．

- n_1 plus n_2 is n_3 (Nat)
- n_2 plus n_1 is n_3 (Nat)

3.2.2 メタ理論

メタ理論とは，導出システムの「出来」を考察するものである．すなわち，判断の内容が正しいかどうかが考察の本質である．

3.2.3 完全性と健全性

メタ定理では，判断が導出できるかどうかが本質であるため，その結果によって内容が正しいことは全く保証しない．その一方，メタ理論では，導出される結果から内容が正しいかどうかを考察する．よって，メタ定理とメタ理論には明確な違いあり，導出と意味内容に直接的な関係が無い．一般に，ある導出システムに対して，次が定義されている．

- 完全性
  内容的に正しい判断ならば，全て導出できる
- 健全性
  導出できるならば，その判断の内容は全て正しい

新しく生成した導出システムが次を満たすがどうかは重要なテーマである．しかし，我々は判断の内容についてはより数学的に踏み込まなければならず難度が高いため，この先は判断の内容に依らず証明できるメタ定理について考える．

3.2.4 メタ定理の例

次の節で帰納法によって証明するメタ定理をいくつか紹介しておく．
なお，これ以降では，「判断 J が導出できる」という表現を，単に，「J である」と表記することにする．

定理 1 (加法の単位元)
導出システム Nat 上で，任意のペアノ自然数 n に対して，次の判断が導出できる．

(1) Z plus n is n
(2) n plus Z is n

定理 2 (加法の一意性)
導出システム上 Nat で，任意のペアノ自然数 n_1, n_2, n_3, n_4 に対して，
n_1 plus n_2 is n_3 かつ n_1 plus n_2 is n_4 ならば，$n_3 \equiv n_4$ である．

定理 3 (加法の演算について閉じていること (全域性))
導出システム Nat 上で，任意のペアノ自然数 n_1, n_2 に対して，

あるペアノ自然数 n_3 が存在して，n_1 plus n_2 is n_3 である．

3.3 帰納法

これからメタ定理を数学的に証明するための技法となる，帰納法について見ていく．

3.3.1 帰納法

定義 1 (帰納法)

X を集合[*1]とし，$P(x)$ を x に関する命題とする．
帰納法とは「任意の X の元 x に対して，$P(x)$ である」という命題を証明するために使われる原理[*2]のことである．

帰納法の原理は，命題の対象となる集合 X が抽象的である．集合 X が自然数集合であれば迷うことは無いであろう．しかし，見慣れない集合でも，性質の良い集合[*1]あれば命題となり得る．この場合，命題を注意深く読むことが重要である．

3.3.2 数学的帰納法

定義 2 (数学的帰納法)

X を自然数全体集合[*3]とし，$P(n)$ を自然数 n に関する命題とする．
このとき「任意の自然数 n に対して，$P(n)$ である」は，次の「(a) かつ (b)」と同値である．

(a) $P(0)$ である．
(b) 任意の自然数 k に対して，$P(k)$ ならば，$P(k+1)$ である．[*4]

数学的帰納法は，自然数に関する帰納法の原理の 1 つである．今後の拡張のため，数学的帰納法で行っていることを整理すると，次のことが言える．

- (a) では，一番小さい自然数 0 について[*5]，命題 $P(0)$ が成立している．
- (b) では，それより大きい自然数 i について，命題 $P(i)$ ならば $P(i+1)$ が成立する．

三段論法を用いることで，すべての自然数 n に対して，$P(n)$ が言える．

3.3.3 累積帰納法

定義 3 (累積帰納法)

X を自然数全体集合[*6]とし，$P(n)$ を自然数 n に関する命題とする．
このとき，「任意の自然数 n に対して，$P(n)$ である」は，次の「(a) かつ (b)」と同値である．

[*1] 実は性質の良い集合であり，後に整礎順序集合として再定義する．
[*2] 原理は，ここでは証明を与えることなく事実を認める程度のものである．つまり，上記の命題を示すための同値条件は成立すると認めるものとする．
[*1] で
[*3] ここでは，次への拡張のため，0 を含む
[*4] $P(k)$ を，帰納法の仮定と呼ぶ．
[*5] 集合の中で一番小さい値，すなわち下限．
[*6] 数学的帰納法と同様に，0 を含む

(a) $P(0)$ である．
(b) 任意の自然数 k に対して，$P(0)$ かつ $P(1)$ かつ…かつ $P(k-1)$ ならば，$P(k)$ である．

累積帰納法は，数学的帰納法の 1 つである．よって，数学的帰納法と累積帰納法は，論理的に同値である．
数学的帰納法と異なる点は，同値条件の (b) であり，累積帰納法を用いることで証明しやすくなる場合がある．
また，(a) は，(b) における $k = 0$ を指す[*7]ので，明記しない場合がある．

3.3.4 帰納法による定義

帰納法を用いて一意性を確かめることによって関数定義することを，帰納法による (関数) 定義という．例として，定理**??**・定理**??**から，次が言える．

- 2 つのペアノ自然数を加法で演算したとき，演算結果は存在するならばただ 1 つである．
- 2 つのペアノ自然数の演算結果は必ずペアノ自然数として存在する．

これにより，任意の 2 つのペアノ自然数 n_1, n_2 に対して，その加法は n_1 plus n_2 は，ただ一つのペアノ自然数 n_3 に決まることが分かる．
以上より，ペアノ自然数における加算操作は，一意性を満たし，数学的な意味での関数[*8]の定義を満たす．つまり，次のように，ペアノ自然数上の関数 $plus(n_1, n_2)$ を定義できる．
$plus(n_1, n_2) = n_3 := n_1$ plus n_2 is n_3

3.3.5 構造帰納法

定義 4 (構造帰納法)
構造帰納法とは，*BNF* [*9]で定義される対象に関する帰納法の原理の総称である．

この段階では，BNF によって定義した集合はペアノ自然数と算術式の 2 つである．これらに関しての構造帰納法を考えていくことで，メタ定理の数学的証明が可能となる．

3.3.6 ペアノ自然数に関する構造帰納法

ペアノ自然数と自然数は 1 対 1 対応するため，数学的帰納法をペアノ自然数で置き換えるだけで得ることが出来る．

定義 5 (ペアノ自然数に関する構造帰納法)
X を **Nat** とし[*10]，$P(n)$ をペアノ自然数 n に関する述語とする．
このとき「任意のペアノ自然数 n に対して，$P(n)$ である」は，次の「(a) かつ (b)」と同値である．

(a) $P(Z)$ である．
(b) 任意のペアノ自然数 n に対して，$P(n)$ ならば，$P(S(n))$ である．

定理**??**を構造帰納法によって証明しよう．

[*7] $k = 0$ のとき，帰納法の仮定はないので，前提なしに $P(0)$ が成立する．
[*8] 厳密な数学の関数の定義は省略する．
[*9] バッカスーナウア記法．プログラミング言語の構文定義をする際の標準的な記法である．
[*10] BNF によって定義されたペアノ自然数の集合

定理 4 (加法の単位元 (再掲))
導出システム Nat 上で，任意のペアノ自然数 n に対して，次の判断が導出できる．

(1) Z plus n is n
(2) n plus Z is n

述語 P は，ここでは判断の事であり，「任意のペアノ自然数 n について，判断 $P(n)$ が導出できる」ことを示せ．ということになる．そして，導出するということは，推論規則を用いて示すということであるから，今回は導出システム Nat 上の推論規則を用いる．

では，構造帰納法を用いて証明していこう．

まず，(1) については，導出システム Nat の推論規則 P-ZERO より，任意の自然数に対して，次のような導出木が作れる．

$$\frac{}{\text{Z plus } n \text{ is } n}\ \text{P-ZERO}$$

よって，成り立つ．

次に (2) について．これは

$$P(n)：\text{判断 } n \text{ plus Z is } n \text{ が導出できる}$$

ということを示せばよい．

実際，推論規則 P-ZERO と，推論規則 P-SUCC を n 回繰り返すことで導出できることは直ちに分かる．よって，ペアノ自然数に関する帰納法を用いて証明する．

$P(n)$ は，次の (a) かつ (b) と同値である．

(a) P(Z)：判断 Z plus Z is Z が導出できる
(b) 任意のペアノ自然数 k に対して，
「判断 k plus Z is k が導出できる」 ならば 「判断 $S(k)$ plus Z is $S(k)$ が導出できる」

つまり，(a)，(b) を示せば，任意のペアノ自然数 n について，$P(n)$ が言える．

(a) P-ZERO を用いると，次のような導出木が作れる．

$$\frac{}{\text{Z plus Z is Z}}\ \text{P-ZERO}$$

よって，成り立つ．

(b) 任意のペアノ自然数 k について，判断 k plus Z is k が導出できると仮定する．この導出を D とすると，導出木は次のように書ける．

$$\text{D} = \frac{\vdots}{k \text{ plus Z is } k}$$

D に推論規則 P-SUCC を用いて，次のような導出木が作れたので，(b) も成り立つ．

$$\frac{D}{S(k) \text{ plus Z is } S(k)}\ \text{P-SUCC}$$

以上より，(a),(b) より，任意のペアノ自然数 n について，判断 $P(n)$ が導出できたので，証明終わり．

ここまで，ペアノ自然数に関する構造帰納法を定義し，証明の様子をみたが，幾つか問題点がある．1 つは，メタ定理によっては，構造帰納法による証明では示すことが出来ないものが存在するということであるが，これは次項で解決する．

もう 1 つは，構造帰納法の定義をこのような拡張によって定義すると，ペアノ自然数以外の BNF 定義の対象に構造帰納法を定義できないということである．そこで，BNF 定義とペアノ自然数に関する構造帰納法の対応に注目する．まず，BNF によって定義されるペアノ自然数集合は次のように記述されていた．

定義 6 (ペアノ自然数集合 Nat)
ペアノ自然数の集合 **Nat** を次の構文で定義する．

$$n \in \mathbf{Nat} ::= Z | S(n)$$

この時，集合 **Nat** の構成規則は次のとおりである．

- $Z \in \mathbf{Nat}$
- 任意の n について，$n \in \mathbf{Nat}$ ならば，$S(n) \in \mathbf{Nat}$

これを見ると，BNF によって定義される集合の元の構成規則とペアノ自然数に関する構造帰納法が対応付いていることが分かる．よって，構造帰納法は，対象となる BNF 集合の元の構成規則から機械的に得ることが出来る．

3.3.7 算術式に関する構造帰納法

前項の結果より，算術式の BNF 定義 **Exp** を，BNF 集合の元の構成規則と読み替えて，算術式に関する構造帰納法を定義しよう．

定義 7 (算術式集合 Exp)
算術式の集合 **Exp** を次の構文で定義する．

$$e \in \mathbf{Exp} ::= n | e + e | e * e$$

この時，集合 **Exp** の構成規則は次のとおりである．

- 任意の $n \in \mathbf{Nat}$ について，$n \in \mathbf{Exp}$
- 任意の e_1, e_2について，$e_1, e_2 \in \mathbf{Exp}$ ならば，$e_1 + e_2 \in \mathbf{Exp}$
- 任意の e_1, e_2について，$e_1, e_2 \in \mathbf{Exp}$ ならば，$e_1 * e_2 \in \mathbf{Exp}$

これより，算術式に関する構造帰納法を定義しよう．

定義 8 (算術式に関する構造帰納法)
X を **Exp** とし，$P(e)$ を算術式 e に関する述語とする．
このとき「任意の算術式 e に対して，$P(e)$ である」は，次の「(a) かつ (b) かつ (c)」と同値である．

(a) 任意のペアノ自然数 n について，$P(n)$ である．
(b) 任意の算術式 e_1, e_2について，$P(e_1)$ かつ $P(e_2)$ ならば，$P(e_1 + e_2)$ である．
(c) 任意の算術式 e_1, e_2について，$P(e_1)$ かつ $P(e_2)$ ならば，$P(e_1 * e_2)$ である．

3.3.8 導出に関する帰納法

算術式も木構造を持っているが，導出の構造も本質的には木構造を持っている．そして，導出の構成規則は，その導出システムの推論規則である．よって，前項にて算術式に関する構造帰納法を定義したが，これより本質性を用いれば導出に関する帰納法を定義することが出来る．

例として，導出システム CompateNat1 における判断 n_1 is less than n_2 の導出に関する帰納法を定義してみよう．まず，導出システム CompareNat1 の推論規則を，D が判断 J の導出であることを $D \in J$ と，算術式の形式で表すと，次のように書ける．

(1) 任意の n に対して，

$$\frac{}{n \text{ is less than } S(n)}\ \text{L-SUCC} \in n \text{ is less than } S(n)$$

(2) 任意の $n_1, n_2, n_3, D_1 \in n_1$ is less than n_2 及び $D_2 \in n_2$ is less than n_3 に対して，

$$\frac{D_1 \qquad D_2}{n_1 \text{ is less than } n_3}\ \text{L-TRANS} \in n_1 \text{ is less than } n_3$$

これを用いることで，CompareNat1 上で，判断 n_1 is less than n_2 の導出に関する帰納法を定義できる．

定義 9 (CompareNat1 における判断 n_1 is less than n_2 の導出に関する帰納法 (ver.1))
二つのペアノ自然数と導出に関する述語 P に対し，「任意のペアノ自然数 n_1, n_2, 判断 n_1 *is less than* n_2 の導出 D に対して，$P(n_1, n_2, D)$」と，以下の (a) かつ (b) は同値である．

(a) 任意のペアノ自然数 n に対して，

$$P\Big(n, S(n), \frac{}{n \text{ is less than } S(n)}\ \text{L-SUCC}\ \Big)$$

(b) 任意のペアノ自然数 n_1, n_2, n_3, 判断 n_1 is less than n_2 の導出 D_1 に，判断 n_2 is less than n_3 の導出 D_2 に対して，$P(n_1, n_2, D_1)$ かつ $P(n_2, n_3, D_2)$ ならば，

$$P\Big(n_1, n_3, \frac{D_1 \qquad D_2}{n_1 \text{ is less than } n_3}\ \text{L-TRANS}\ \Big)$$

である．

しかし，通常のメタ定理の文言では，述語 P の中では導出に明示的に言及しないことが多い．そのため，次のように導出に関する部分を省略した形で単純化することが多い．

定義 10 (CompareNat1 における判断 n_1 is less than n_2 の導出に関する帰納法 (ver.2))
二つのペアノ自然数に関する述語 P に対し，「任意のペアノ自然数 n_1, n_2 に対して，n_1 *is less than* n_2 ならば，$P(n_1, n_2)$」と，以下の *(a)* かつ *(b)* は同値である．

(a) 任意のペアノ自然数 n に対して，$P(n, S(n))$ である．
(b) 任意のペアノ自然数 n_1, n_2, n_3 に対して，$P(n_1, n_2)$ かつ $P(n_2, n_3)$ ならば，$P(n_1, n_3)$ である．

同様にして，CompareNat2, CompareNat3 でも，判断 n_1 is less than n_2 の導出に関する帰納法が定義できる．

定義 11 (CompareNat2 における判断 n_1 is less than n_2 の導出に関する帰納法 (ver.2))
二つのペアノ自然数に関する述語 P に対し，「任意のペアノ自然数 n_1, n_2 に対して，n_1 *is less than* n_2 ならば，$P(n_1, n_2)$」と，以下の *(a)* かつ *(b)* は同値である．

(a) 任意のペアノ自然数 n に対して，$P(Z, S(n))$ である．
(b) 任意のペアノ自然数 n_1, n_2 に対して，$P(n_1, n_2)$ ならば，$P(S(n_1), S(n_2))$ である．

定義 12 (CompareNat3 における判断 n_1 is less than n_2 の導出に関する帰納法 (ver.2))
二つのペアノ自然数に関する述語 P に対し，「任意のペアノ自然数 n_1, n_2 に対して，n_1 *is less than* n_2 ならば，$P(n_1, n_2)$」と，以下の *(a)* かつ *(b)* は同値である．

(a) 任意のペアノ自然数 n に対して，$P(n, S(n))$ である．
(b) 任意のペアノ自然数 n_1, n_2 に対して，$P(n_1, n_2)$ ならば，$P(n_1, S(n_2))$ である．

3.3.9 整礎帰納法

これまで定義してきた帰納法には，実は共通点が存在する．それは，次の点である．

- 集合の中で一番小さい要素に対しては，述語 (命題)P をそのまま返す
- 大きい要素に対しては，その部分で小さい要素に対して P が成り立つと仮定して，
  その大きい要素に対しても P が成り立つことを示す．

よって，この論法が妥当であるためには，一番小さい要素が必ず存在すること．つまり，無限に小さい要素を見つけることが出来ない性質を持っていることが重要である．そのような性質を持つ集合は次のように厳密に定義される．

定義 13 (整礎順序集合)
整礎順序集合 $(X, \prec)$ とは，集合 X と X 上の二項関係 $\prec \subseteq X \times X$ の組で，$\prec$ が次の整礎性の条件を満たすことである．
整礎性の条件：X の任意の空でない部分集合 S が $\prec$ に関する極小元 [*11]を持つこと．

これより，集合 X を整礎順序集合として帰納法の原理を考えると，整礎帰納法の原理を得られる．

定義 14 (整礎帰納法の原理)
整礎順序集合 $(X, \prec)$ と X の元 x に関する述語 $P(x)$ が与えられたとき，次は同値である

- 「任意の $x \in X$ に対して，$P(x)$」
- 「任意の $x \in X$ に対して，「任意の $y \prec x$ に対して $P(y)$」ならば，$P(x)$」

つまり，整礎帰納法の原理は，帰納法原理の抽象化であり，今回登場した全ての帰納法は，整礎帰納法の具体例であったことが分かる．すなわち，それらの集合は全て整礎順序集合であったということである．
また，同じ集合でも，二項関係の定義の仕方によって異なる帰納法の原理を得ることが出来る．その代表例として，数学的帰納法を累積帰納法が挙げられる．それぞれ集合は自然数の集合として定義されるが，順序 $\prec$ は次のような違いがあったのである．

[*11] $\prec$ に関する極小元とは，任意の $s \in S$ について，$s \not\prec m$ を満たすような m のこと

- 数学的帰納法：$n \prec m := m = n + 1$
- 累積帰納法：$n \prec m := n < m$

3.4 総評

メタ定理とメタ理論については，導出と意味内容の関係性をよく押さえておくことが重要であった．比較的平易な内容であったため，ここまでの理解は良かったと感じた．次に，帰納法については，一番身近な数学的帰納法から拡張してメタ定理を証明する帰納法を習得したが，やはり普段の感覚と異なる帰納法に少々悩ましく感じている印象だった．一部数学的要素が強い項もあったが，そちらは感覚をつかむ程度に内容を扱い，次項を読めるように工夫した．とはいえ，全体として抽象的な理論も多く，理論としての難易度は高かったように感じた．

第 3 週　整数・真偽値式の評価

小泉　孝弥

1　やった内容

1.1　導出システム EvalML1

Definition 1.1. (ML1 の構文)
ML1 における， 式の集合 **Exp** と式の評価値である値の集合 **Value** を以下の BNF で定義する．

1. $i \in \mathbf{int}$
2. $b \in \mathbf{bool}$
3. $v \in \mathbf{Value} ::= i|b$
4. $e \in \mathbf{Exp} ::= i|b|e\ op\ e|if\ e\ then\ e\ else\ e$
5. $op \in \mathbf{Prim} ::= + \mid - \mid * \mid <$

ここで， $\mathbf{int} = \{n \in \mathbb{N} | 0 \leq n \leq 9\}$ で $\mathbf{bool} = \{$ true， false $\}$ である．

Example 1.2. (ML1 の例)
ML1 の具体例としては以下のようなものがある．

1. $1 + 2 * 3$
2. if $3 <$ 4then true else 5

ここで， 具体例の 1 に注目すると $+$， $*$ という 2 つの演算を用いている． しかし， 今のままでは計算結果の一意性が崩れるため計算をすることができない． そこで，それぞれの演算子の強さというものを定義する．

Definition 1.3. (結合演算子の強さ)
op_1， $op_2 \in \mathbf{Prim}$ とする． op_1 が op_2 より結合が強いとは

$$\forall e_1, e_2, e_3 \in \mathbf{Exp}, e_1\ op_1\ e_2\ op_2\ e_3 = (e_1\ op_1\ e_2)\ op_2\ e_3,$$

$$\forall e_1, e_2, e_3 \in \mathbf{Exp}, e_1\ op_2\ e_2\ op_1\ e_3 = e_1\ op_2\ (e_2\ op_1\ e_3).$$

が成り立つことである． またこのことを $op_1 \to op_2$ と表す．

Definition 1.4. (結合の強さが等しい)
op_1， $op_2 \in \mathbf{Prim}$ とする． op_1 と op_2 の結合が等しいことを

$$op_1 \to op_2 \text{かつ } op_2 \to op_1$$

と定義し， $op_1 = op_2$ と表す．

Axiom 1.5. (**Prim** の結合の強さ)
Prim の元について計算順序を以下のように定める．

$$* \rightarrow + = - \rightarrow <$$

Attention 1.6. (条件文の範囲)
条件文を使うときはその範囲を指定してあげる必要がある． 例えば，

$$\text{if } e_1 \text{ then } e_2 \text{ else } e_3 + e_4.$$

という文があった場合，

$$(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) + e_4.$$

という文と，

$$\text{if } e_1 \text{ then } e_2 \text{ else } (e_3 + e_4).$$

文の 2 通りの結果が得られてしまい， 一意に定まらないため範囲を指定してあげる必要がある．

Definition 1.7. (EvalML1 の判断形式)
導出システム **EvalML1** における判断の形式は以下の 5 つである．

1. $e \Downarrow v$
2. i_1 plus i_2 is i_3
3. i_2 minus i_2 is i_3
4. i_2 times i_2 is i_3
5. i_1 less than i_2 is b

また， これには推論規則というものが存在する．

$$\frac{}{i \Downarrow i}\ E-INT$$

$$\frac{}{b \Downarrow b}\ E-BOOL$$

$$\frac{e_1 \Downarrow true \quad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}\ E-IFT$$

$$\frac{e_1 \Downarrow false \quad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}\ E-IFF$$

$$\frac{e_1 \Downarrow i_1 \quad e_2 \Downarrow i_2 \quad i_1 \text{ plus } i_2 \text{ is } i_3}{e_1 + e_2 \Downarrow i_3}\ E-PLUS$$

E-PLUS と同様に E-MUNUS, E-TIMES, E-LT というものも定義される．

Axiom 1.8. (ML1 式における帰納法の原理)
ML1 式の成立を述べた命題 P に対し，「任意の $e \in \mathbf{Exp}$ について $P(e)$ が成立する.」と以下の 4 つが成立することは同値である.

1. 任意の $i \in \mathbf{int}$ について $P(i)$ である.
2. 任意の $b \in \mathbf{bool}$ について $P(b)$ である.
3. 任意の $e_1, e_2 \in \mathbf{Exp}$ と $op \in \mathbf{Prim}$ について， $P(e_1)$ かつ $P(e_2)$ ならば $P(e_1\ op\ e_2)$ である.
4. 任意の $e_1, e_2, e_3 \in \mathbf{Exp}$ について， $P(e_1)$ かつ $P(e_2)$ かつ $P(e_3)$ ならば $P(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)$ である.

Theorem 1.9. (評価の一意性)
任意の $e \in \mathbf{Exp}$, $v_1, v_2 \in \mathbf{Value}$ について， $e \Downarrow v_1$ かつ $e \Downarrow v_2$ ならば $v_1 \equiv v_2$ である.

1.2 導出システム EvalML1Err

前説では判断システム **EvalML1** というものを定義してきた． ここで， 以下のような式を考える.

$$1 + \text{true}$$

ここで $1 + \text{true} \in \mathbf{Exp}$ であるが， これは評価ができない． すなわち,

$$\nexists v \in \mathbf{Value} \text{ s.t.} (1 + \text{true}) \Downarrow v.$$

このようなものに対しても評価を与えるため **Error** というものを付け加えた **EvalML1Err** というものを考える.

Definition 1.10. (導出システム EvalML1Err)
導出システム **EvalML1Err** は以下の判断形式を持つ.

1. $e \Downarrow v$
2. $e \Downarrow \mathbf{Error}$
3. i_1 plus i_2 is i_3
4. i_2 minus i_2 is i_3
5. i_2 times i_2 is i_3
6. i_1 less than i_2 is b

また， これにも $\mathbb{E}\tilde{\supset}\prec\mathbb{ML}\mathbb{F}$ のものに加えて， 新しく追加した **Error** についても推論規則を定める．しかしながら量が多いためここではその一部を紹介することにとどめる (詳しくは参考文献を参照).

$$\frac{e_1 \Downarrow b}{e_1 + e_2 \Downarrow \mathbf{Error}}\ E - PLUSBOOLL$$

$$\frac{e_1 \Downarrow \mathbf{Error}}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow \mathbf{Error}}\ E - IFERROR$$

$$\frac{e_1 \Downarrow \mathbf{Error}}{e_1 + e_2 \Downarrow \mathbf{Error}}\ E - PLUSERRPRL$$

Theorem 1.11. (評価の全域性)
導出システム **EvalML1Err** において， 任意の $e \in \mathbf{Exp}$ において， 以下の 1. か 2. のどちらか一歩が成立する.

1. $\exists v \in \mathbf{Value}$ s. t.$e \Downarrow v$
2. $e \Downarrow \mathbf{Error}$

Theorem 1.12. (評価の一意性)
導出システム **EvalML1Err** において，任意の $e \in \mathbf{Exp}$, $v_1, v_2 \in \mathbf{Value} \cup \{\mathbf{Error}\}$ について，$e \Downarrow v_1$ かつ $e \Downarrow v_2$ ならば $v_1 \equiv v_2$ である.

これらの定理より， $\Downarrow$ が **Exp** から **Value** $\cup \{$**Error**$\}$ への写像であることが言えた.

2 感想

今回の意味論の発表では， 評価という概念が写像であることの証明をした． プログラミングが普段， 私が勉強している数学の言葉で考えることによって， プログラミングへの理解が少し深まった気がした．また，他の学部の人にも数学の概念を説明する機会も何度かあったので， 自分の数学の概念への理解も深めることができる良い機会となった.

1　定義，変数束縛と環境

定義とは，簡単のため「物事に名前をつける」こととし，そのつけた名前を**変数名**というと定義する．

例えば，数学では，$E = \{x | x$ は任意の偶数 $\}$ のように，と E という変数を任意の偶数の集合であると定義できる．

プログラムの分野においても，多くのプログラミング言語では変数をその名前とともに宣言でき，宣言した変数を参照でき，変数には参照できる時間的・空間的な**有効範囲**が定められている．

ここでは，変数名とその変数が参照する値との組である**変数束縛**とその集まりである**環境**という概念を導入し，式の評価を定義する．

1.1　let による定義

OCaml では以下のような `let` 式を用いて変数の定義を行う．

`let` ＜変数＞ `=` ＜式$_1$＞ `in` ＜式$_2$＞

.

これを英語だと思って直訳すると，「＜式$_2$＞において＜変数＞を＜式$_1$＞と等しいとせよ」ということで，「式の値をどのように計算するか」という観点から読み直すと「`let` 式全体の値を求めるには，＜変数＞を＜式$_1$＞（の値）として＜式$_2$＞の値を求めよ」ということである．

1.2　変数宣言の有効範囲

OCaml の `let` 式の場合，＜変数＞の有効範囲は＜式$_2$＞である．ただし，ある変数の有効範囲内に**同じ名前**の変数が宣言されている場合には，内側の変数の宣言が優先される．

1.3　let 式の評価と環境

定義のある式の評価を行う方法として，一番単純な方法のひとつが，**代入**とも呼ばれる「変数参照を定義内容で置換する」という方法である．変数の参照先である式をいつ計算するかという問題は，プログラミング言語の特徴を決める重要な要素であり，**評価戦略**とも呼ばれている．

環境は，変数を含んだ式を評価する上での前提のようなものであり，「環境 $\mathcal{E}$ のもとで式 e の値は v である」のように使う．環境は，`let` 式が書かれるたびに＜式$_1$＞の評価結果を Push され，＜式$_2$＞が評価されると Pop されるスタックとも考えられる．

1.4　導出システム EvalML2

EvalMath2 は，ML1 に `let` 式を加えた ML2 の式の評価を表現したシステムである．まず，式や環境（メタ変数 $\mathcal{E}$）などの定義構文は以下のように与えられる．

$$
\begin{aligned}
i &\in \text{int} \\
b &\in \text{bool} \\
x, y &\in \text{Var} \\
v &\in \text{Value} ::= i | b \\
\mathcal{E} &\in Env ::= \bullet | \mathcal{E}, x = v \\
e &\in Exp ::= i | b | x | e\ op\ e | \texttt{ if } e \texttt{ then } e \texttt{ else } e \\
&\quad | \texttt{ let } x \texttt{ = } e \texttt{ in } e \\
op &\in Prim ::= + | - | * | <
\end{aligned}
$$

ここでは空の環境を $\bullet$ で表している．式の自由変数の集合 FV(e) を以下のように式の構造に関して帰納的に定義する．

$$
\begin{aligned}
FV(i) &= \emptyset \\
FV(b) &= \emptyset \\
FV(x) &= \{x\} \\
FV(e_1\ op\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\texttt{if} e_1 \texttt{ then } e_2 \texttt{ else } e_3) &= FV(e_1) \cup FV(e_2) \cup FV(e_3) \\
FV(\texttt{ let } x = e_1 \texttt{ in } e_2) &= FV(e_1) \cup (FV(e_2) \setminus \{x\})
\end{aligned}
$$

これは let の場合の $EV(e_2)\backslash\{x\}$ が，x の有効範囲が e_2 であることを示している．また，環境中で束縛されている変数の集合 $dom(\mathcal{E})$ を，以下のように環境の構造に関して帰納的に定義する．

$$
\begin{aligned}
dom(\bullet) &= \emptyset \\
dom(\mathcal{E}, x = v) &= dom(\mathcal{E}) \cup \{x\}
\end{aligned}
$$

ML2 式が意味を持つ（評価の結果が値になるかエラーになる）ためには，環境が自由変数に対する値を与える（$EV(e) \subseteq dom(\mathcal{E})$）ことが十分条件になる．

評価の判断は，すでに議論したように，式と値だけではなく環境に言及する必要があり「環境 $\mathcal{E}$ のもとで式 e の値は v である」という意味を以下のような式で書くと定義する．

$$
\mathcal{E} \vdash e \Downarrow v
$$

EvalML1 に変数と環境を追加した EvalML2 は以下のようになる．ただし，ほとんどの推論規則は推論するときに環境を考慮するという点での違いしかないため，ここで紹介するものは新たに追加された推論規則のみに留める．

$$
\frac{}{\mathcal{E}, x = v \vdash x \Downarrow v} \quad \text{(E-VAR1)}
$$

$$
\frac{(y \neq x) \quad \mathcal{E} \vdash x \Downarrow v_2}{\mathcal{E}, y = v_1 \vdash x \Downarrow v_2} \quad \text{(E-VAR2)}
$$

$$
\frac{\mathcal{E} \vdash e_1 \Downarrow v_1 \quad \mathcal{E}, x = v_1 \vdash e_2 \Downarrow v}{\mathcal{E}, \vdash \texttt{let} x = e_1 \texttt{in} e_2 \Downarrow v} \quad \text{(E-LET)}
$$

2　感想

日頃何気なく使っているプログラミング言語だが，そのコードの「意味」を深く考える「意味論」に触れることができた．また，ML 系の言語に初めて触れ，既知のプログラミング言語と対比して特徴を掴むこともできた．大変意義のある活動になったと考える．

5章　関数と再帰

1. はじめに

　第5章では関数と再帰的定義の機能のを ML2 に追加した ML3 を導入しその式の意味を導入システム EvalML3 として与えた．ここで言う関数とはプログラミング言語の概念の一つである．
ここで $\bigcirc * \bigcirc = \bigcirc^2$ といった関数があるとする。このとき ○ に数字を入れることを関数適用または、関数呼び出しといい、その結果として $\bigcirc^2$ といった返り値を得られる．このとき○に入れる数字を引数と呼び、○自体をパラメータまたは、仮引数と呼ぶ.

　今回のプロジェクトの学習のために用いた OCaml では関数を引数に取る関数や、関数を返り値とする関数といった「関数に関する関数」を扱える．このような関数を高階関数と呼ぶ.

2. fun 式と関数定義

　OCaml では、関数を

$$\texttt{fun} \ <\text{変数}> \rightarrow <\text{式}>$$

という構文で書くことができる．式を（関数の）本体（body）と呼ぶ.
以下は二乗関数 square を定義して 1 から 5 までの二乗和を計算する式である．

$$\texttt{let} \quad \texttt{square} = \texttt{fun} \quad x \rightarrow x * x \quad \texttt{in}$$

$$\texttt{square} \quad 1 + \texttt{square} \quad 2 + \texttt{square} \quad 3 + \quad \texttt{square} \quad 4 + \texttt{square} \quad 5$$

この式の一行目は関数 *square* の定義である．ここで、関数の定義は「関数を作る構文（fun)」と「式を定義する構文（let)」という独立した構文の組み合わせで実現されている．
また、この式は以下のようにも記述することができる．

$$\texttt{let} \quad \texttt{square} \quad x = x * x \quad \texttt{in}$$

$$\texttt{square} \quad 1 + \texttt{square} \quad 2 + \texttt{square} \quad 3 + \texttt{square} \quad 4 + \texttt{square} \quad 5$$

3. 高階関数

　上と同様に考えると 3 乗和を計算する式は以下のようになる．

$$\texttt{let} \quad \texttt{cube} = \texttt{fun} \quad x \rightarrow x * x * x \quad \texttt{in}$$

$$\texttt{cube} \quad 1 + \texttt{cube} \quad 2 + \texttt{cube} \quad 3 + \texttt{cube} \quad 4 + \texttt{cube} \quad 5$$

この式は、任意の整数上の関数 f について、f 1 から f 5 までの和を求める式を書くことができる．つまり f をパラメータとして考えれば以下の関数を考えることができる．

$$\texttt{let f} = \texttt{f 1} + \texttt{f 2} + \texttt{f 3} + \texttt{f 4} + \texttt{f 5}$$

を考えることができる．ここで 1 から 5 までの三乗和から二乗和の差を求める式は

$$\texttt{let sum} = \texttt{fun f} = \texttt{f 1+f 2+f 3+f 4+f 5 in}$$

$$\texttt{let square} = funx \rightarrow x * x \ \texttt{in}$$

$$\texttt{let cube} = \texttt{fun}\ x \rightarrow x * x * x \quad \texttt{in}$$

$$\texttt{sum cube} - \texttt{sum square}$$

このことからわかるように、関数を別の関数の引数として渡せるということは、関数適用式において呼び出される関数は環境に依存し、（一般には）式を評価してみるまでわからないということでもある．

上の sum は関数を引数とする関数の例だが、関数を返り値とする関数も有用である．特に複数の引数を取る関数を記述するためにこれは使うことができる. ここで二整数の大きい方を返す関数を定義する. この二引数関数を一引数関数で記述すると以下のようになる

$$\texttt{let max} = \texttt{fun x} \rightarrow \texttt{fun y} \rightarrow \texttt{if}\ x{<}y \quad \texttt{then}\ y\ \texttt{else}\ x\ \texttt{in}...$$

if は x と y の大小比較をして大きい方を値とする式である. この式で fun がどのような振る舞いをするかというと、外側の fun x → …は適用されると、fun y →…という関数を返す関数である. この max は以下のように使うことができる

$$\texttt{let max} = ...\ \texttt{in}$$

$$\texttt{let f} = \texttt{max 5 in}$$

$$\texttt{f 4}$$

f は max の x に 5 を与えたものであり、y を引数として、5 と比べて大きいものを返す関数だと考えられる．

この方法は二引数関数だけでなく一般の多変数関数にも拡張可能であり、このような”複数の引数を取る関数を、関数を返す関数で実現する”ことをカリー化、カリー化された関数のことをカリー化関数という． カリー化関数は場合によっては、max 5 のような部分的に引数の与えられた関数を使うことが可能である．このような関数を何度も使いたい場合、

$$\texttt{let max} = ...\ \texttt{in}$$

$$\texttt{let max5} = \texttt{max}\ 5\ \texttt{in}$$

$$...\quad \texttt{max5}\ x... \ \texttt{max5}\quad y...$$

などと何度も比較に使うことができる．このように、本来必要な個数より少ない個数の引数を与えることを部分適用と呼び、これによりより特化した関数を作ることができるようになる．

4. 静的有効範囲と関数閉包

$$\texttt{let}\ a = 3\ \texttt{in}$$

$$\texttt{let f} = \texttt{fun}\ \ \rightarrow y * a\ \texttt{in}$$

$$\texttt{let}\ a = 5\ \texttt{in}$$

$$\texttt{f}\ 4$$

上式は二行目で定義されている式により引数 y に a を掛ける関数であり、それをもう一つの a の有効範囲内である四行目で 4 に適用している．a は 1 行目と 3 行目で定義されているが、ここで 4 に掛けら

れる数は一体何なのだろう．これを実際に OCaml で試すと $4*3$ で 12 となる．また `f 4` を `f 5` にすると $5*3$ が計算され 15 となる．つまり、関数の呼び出し時に有効な環境に束縛されている a の値にかかわらず `f` は 3 倍を計算する関数である．

これは 4.2 節で説明された静的有効範囲のためであり、`fun` 式の a が 1 行目で定義された a の有効範囲であるからであり、どこで呼び出されても関数での a の値は 3 である．

このことから静的有効範囲を正しく実現するためには本体のパラメータ以外の変数の `fun` 式を評価した時点での値が必要になることがわかる．このような、関数の式に関する情報とパラメータ以外の変数の値を組みにしたものを関数閉包と呼ぶ．

5. 導出システム EvalML3(その 1)

これまでの関数の意味を導出システムにしたものが EvalML3 である.ML3 の式は、ML2 の式に `fun` 式、関数適用式を加えたものとして定義される．

p84 参照

ここで Value の定義に追加された（ε）[`fun x → e`] が, 関数閉包を示す値である. パラメータ以外の変数の値を保持するために, 環境を使っている.（このため ML3 では値と環境の定義が互いに依存している.）`fun` 式は結合の優先度が一番弱くできるだけ右に伸びる. 関数適用式は結合が最も強く左結合する.

判断の形式は ML2 と同じく,EvalML3 で追加される式は以下の 2 つである．

p84 参照

6. 再帰的定義

OCaml では再帰的定義のために `let rec` 式という構文がある．記述法は以下

`let rec` ＜変数＞=＜式＞ `in` ＜式＞

let 式とは違い＜変数＞の有効範囲は前後の式両方である．

ここでフィボナッチ数列の 5 番目を求めるための式を考えると以下のようになる

$$\texttt{let rec fib} = \texttt{fun } n \rightarrow$$

$$\texttt{if n<3 then 1 else fib } (n-1) + \texttt{fib}(n-2) \texttt{ in}$$

$$\texttt{fib } 5$$

再帰は使い方次第で、値どころかエラーなどの実行結果も得られない式をつくることができる. 例えば、

$$\texttt{let rec f} = \texttt{fun}x \rightarrow \texttt{f } x + \texttt{f } x \texttt{ in}$$

$$\texttt{f } 3$$

この式によると、`f 3` の値は `f 3`+`f 3` の値でありまた、その値は（`f 3`+`f 3`)+(`f 3`+`f 3`) である. つまりこのままこの式の値は発散する.

このように実行結果の無い式がある場合、式の計算をどの順番で行うかによって、結果が変わる場合が出てくる. 例えば、

$$\texttt{let rec f} = \texttt{fun } x \rightarrow \texttt{f } x + \texttt{f } x \texttt{ in}$$

$$\texttt{let zero} = \texttt{fun } y \to 0 \texttt{ in}$$

$$\texttt{zero(f 3)}$$

関数 $zero$ は引数として何が与えられても 0 を返す関数でありこれに、計算を行うと発散する式が引数として与えられている. この式の値は 0 としても良い気がするが、OCaml では関数の本体を計算する前に実引数の計算を済ませその値を関数に渡すことになっているので、この式は計算結果を持たない. 関数適用に関してこのような方式（評価戦略）を値呼びと呼ぶ. これに対し「実引数の計算をする前にひとまず本体の計算を始め、実引数の計算が必要になった時点でする」ような方式を、遅延評価と呼ぶ.

7. 導出システム EvalML3(その 2) `let rec` 式などを含めた BNF 定義は以下のようになる.
   p87 参照
   ここでは再帰関数（の値）を
   $$(\,\varepsilon\,)[\texttt{rec } x = \texttt{fun } y \to e]$$
   という形の特別な関数閉包として考える. この関数閉包は `let rec` 式を評価したときに作られ、y が関数のパラメータ,x が本体 e 中で自分自身を指すための名前である. この特別な関数閉包は, 実引数が与えられると、パラメータを実引数に束縛させるだけでなく自分自身を示す名前 x を関数閉包自身に束縛するものである. この 2 つ目の束縛によって再帰的関数が実現される.
   これを表したのが以下の 2 つの規則である。
   p87 参照

8. 感想
   今回の意味論の学習を通して, 普段意識しないようなプログラミング言語の, 特に関数の考え方を勉強することができた. 関数のカリー化や, 関数閉包などは所見時大変理解に苦しんだが, 自身のプログラミング言語に関しての理解が深まることにつながったと考える.

第6週目

文責: 西見元希

操作的意味論を実装で理解するために，OCaml の基本的な文法を学習した.

学習した文法

基本的な概念

端末で ocaml コマンドを実行すると repl と呼ばれる対話型実行環境が開く． 入力として受け付けることができるのは OCaml の式として定められたものだけであり， 入力の末尾にはここまでが式であることを伝えるために `;;` をつける必要がある.

算術演算について

数値を評価するとその数値自身を返す.

```
# 5;;
- : int = 5
```

評価結果に現れている int は評価結果の型を示している． 型が同じ数に限り四則演算ができる．これは演算子の定義がそうなっているからである.

```
# 2 + 3;;
- : int = 5

# 2.0 +. 3.0;;
- : float = 5.

# 2 +. 3.0;; (* 2はint, 3.0はfloat型 *)
  Error: This expression has type float but an expression was expected of type int
```

名前の束縛

他言語でいう変数定義にあたる．値に名前をつけることができる.

構文は `let name = expression` である． また，同じ構文で関数も定義できる．再帰的な関数を定義する場合は `let` の後に `rec` キーワードをつける必要がある.

```
let a = 5;;
let f x = x + x;;
let rec fact n =
  if n = 1
    then 0
    else n * fact (n - 1);;
```

局所変数

式の中で使いたいちょっとした変数を定義するのにletを用いる． 構文は `let name = expression in body` であり，ここで定義した `name` は `body` 式の中でのみアクセスできる.

```
let fourth x =
  let
    sqr = x * x
  in
    sqr * sqr;;
```

リスト

`[1; 2; 3; 4; 5]` などのようなデータの列をリストと呼ぶ．各データはそれぞれ同じ型である必要がある．

```
# [1;2;3;4;5];;
- : int list = [1; 2; 3; 4; 5]
# ["a"; "b"; "c"];;
- : string list = ["a"; "b"; "c"]
(* 異なる型は同じリストに入らない *)
# [1; "a"];;
  Error: This expression has type string but an expression was expected of type int
```

リストの先頭に値を追加するときは::演算子を使う．(右の項はリストでなければならない)

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]
```

このことから，`[1;2;3;4;5]` というリストは `1::2::3::4::5::[]` という式と同じ意味を持つことが分かる．

パターンマッチ

if のように条件分岐しながらも判定した式の構成要素に名前をつけることができる機能．

```
(* パターンマッチを用いた階乗を求める関数の例 *)
let rec fact x =
  match x with
    0 -> 1
  | n -> n * fact (n-1);;
```

ヴァリアント

構造を持つ型であり，代数的データ型などとも呼ばれる．使用例は次の通り．

```
type figure =
    Point (* Pointという値 *)
  | Circle of int  (* Circle 3 は半径3の円 *)
  | Rectangle of int * int (* Rectangle (3,4) は3*4の長方形 *)
  | Square of int;; (* Square 5 は一辺が5の正方形 *)
```

総評

感想

関数型言語 OCaml の文法を学習することで，OCaml の最大の特徴であるパターンマッチの高級さに驚かされた．この知見を活かしてこれからの開発に役立てていきたい．

第7週

文責: 西見元希

OCaml で抽象構分木を評価することのできる簡単なインタプリタを実装した．ここではその過程を説明する．

前提知識

操作的意味論とは，プログラムの意味を段階的に実行して見せることでプログラムの意味を説明しようというものである．英語で言えばwalkという単語の意味を実際に歩いて見せることによって説明することに似ている．ここでは，OCamlで算術式の抽象構文木を簡約して計算結果を求めるという抽象機械を実装した． 抽象機械とは，コンピュータの理論モデルのようなものであり一般的には入力・入力に加える作用・出力の3つの要素からなる．

実装

基礎的な算術式を評価する抽象機械の定義

まず，抽象構分木の定義から行った．ここでは整数・加算式・乗算式を算術式と呼ぶことにした．

```
type expression =
    Num of int
  | Add of expression * expression
  | Multiply of expression * expression
```

これが算術式を示すexpression型を定義であり，値の例は次のようになる・

- `Num 3` :算術式 `3` という意味
- `Add (Num 2, Num 3)` :算術式 `2+3` という意味
- `Multiply (Add (Num 1, Num 2), Num 4)` :算術式 `(1+2)*4` という意味

また，抽象構文機を直感的に表示する関数も定義しておく． 例えば `Add (Num 3, Num 4)` に適用すると `"<<(3 + 4)>>"` という文字列を返す．

```
let show exp =
  let rec showexp = function
      Num a -> string_of_int a
    | Add (x,y) -> "(" ^ showexp x ^ " + " ^ showexp y ^ ")"
    | Multiply (x,y) -> "(" ^ showexp x ^ " * " ^ showexp y ^ ")"
  in "<< " ^ showexp exp ^ " >>"
```

次に算術式を1段階だけ簡約する関数reduceを書く．この関数は次のように動作する．

- 整数を受け取ったらそれ以上簡約できないのでそのまま返す．(操作的には簡約しているとも言えてしまうので，簡約できるかどうかを判定する関数を別途書く必要がある)
- 加算式を受け取ったら，可能であれば左辺を簡約する．左辺が簡約不可能であれば右辺を簡約する．両方とも簡約不可能であればその式はAdd (Num x, Num y)の形になっているのでNum (x+y)を返す．
- 乗算式も加算とほぼ同様のことを行う．

以上の動作はOCamlのパターンマッチのおかげで直感的に記述することができる．

```
let is_reducible = function
    Num a -> false
  | _ -> true
  (* let is_reducible exp = match exp with *)

let rec reduce = function
    Num a -> Num a
  | Add (Num x, Num y) -> Num (x + y)
```

```
  | Add (x,y) ->
      if is_reducible x then Add (reduce x, y) else Add (x, reduce y)
  | Multiply (Num x, Num y) -> Num (x * y)
  | Multiply (x,y) ->
      if is_reducible x then Multiply (reduce x, y) else Multiply (x, reduce y)
```

これで式の簡約ができるようになった.

実行例:

```
# let e = Add (Add (Add (Num 6, Num 2), Num 3), Add (Num 1, Num 4));;
val e : expression =
  Add (Add (Add (Num 6, Num 2), Num 3), Add (Num 1, Num 4))
# reduce e;;
- : expression = Add (Add (Num 8, Num 3), Add (Num 1, Num 4))
# let e2 = Add (Add (Add (Num 6, Num 2), Num 3), Multiply (Num 1, Num 4));;
val e2 : expression =
  Add (Add (Add (Num 6, Num 2), Num 3), Multiply (Num 1, Num 4))
# reduce e2;;
- : expression = Add (Add (Num 8, Num 3), Multiply (Num 1, Num 4))
# reduce (reduce e2);;
- : expression = Add (Num 11, Multiply (Num 1, Num 4))
# reduce (reduce (reduce e2));;
- : expression = Add (Num 11, Num 4)
# reduce (reduce (reduce (reduce e2)));;
- : expression = Num 15
```

最後に，式が最終的な値になるまでis_reducibleとreduceを繰り返す関数runを定義する． これで抽象機械が完成する.

```
let rec run exp =
  print_endline @@ show exp;
  if is_reducible exp
    then run (reduce exp)
    else exp;;
```

定義の拡張

`expression` や `show` などを拡張し，bool 値や比較演算を評価できるようにしていく． 追加する項目は次の通りである.

- expression型におけるBooleanとLessthanの追加
- show関数のBooleanとLessthanへの対応
- is_reducibleのBooleanへの対応
- reduce関数におけるBooleanとLessthanの簡約規則の追加

実際の定義は次のようになる.

```
type expression =
    Num of int
  | Add of expression * expression
  | Multiply of expression * expression
  | Boolean of bool
  | Lessthan of expression * expression;;

let show exp =
  let rec showexp = function
      Num a -> string_of_int a
    | Add (x, y) -> "(" ^ showexp x ^ " + " ^ showexp y ^ ")"
    | Multiply (x, y) -> "(" ^ showexp x ^ " * " ^ showexp y ^ ")"
    | Boolean p -> string_of_bool p
    | Lessthan (x,y) -> "(" ^ showexp x ^ "<" ^ showexp y ^ ")"
  in "<< " ^ showexp exp ^ " >>";;

let is_reducible = function
    Num a | Boolean a -> false
  | _ -> true;;

let rec reduce = function
```

```
    Num a -> Num a
  | Boolean p -> Boolean p
  | Add (Num x, Num y) -> Num (x + y)
  | Add (x, y) ->
      if is_reducible x then Add (reduce x, y)
        else Add (x, reduce y)
  | Multiply (Num x, Num y) -> Num (x * y)
  | Multiply (x, y) ->
      if is_reducible x then Multiply (reduce x, y)
        else Multiply (x, reduce y)
  | Lessthan (Num x, Num y) -> Boolean (x < y)
  | Lessthan (x,y) ->
      if is_reducible x then Lessthan (reduce x,y)
        else Lessthan (x,reduce y);;
```

これで新しい式が実行できるようになった.

```
# let exp = Lessthan(Add(Num 3, Num 4), Multiply(Num 3, Num 2));;
val exp : expression = Lessthan (Add (Num 3, Num 4), Multiply (Num 3, Num 2))
# run exp;;
<< ((3 + 4)<(3 * 2)) >>
<< (7<(3 * 2)) >>
<< (7<6) >>
<< false >>
- : expression = Boolean false
```

練習課題

活動時間内における演習としてifや&&, ||の機能を実装した. 短縮のためifの実装のみの解答例を掲載する.

```
type expression =
  (* 省略 *)
  | If of expression * expression * expression

let show exp =
  let rec showexp = function
      (* 省略 *)
    | If (p,t,f) ->
        "if " ^ showexp p ^ " then " ^ showexp t ^ " else " ^ showexp f
  in
    "<< " ^ showexp exp ^ " >>"

let reduce = function
  (* 省略 *)
  | If (Boolean p, t, f) -> if p then t else f
  | If (p, t, f) -> If(reduce p, t, f)
```

実行時エラーの実装

計算の失敗を表現するためにErrorを定義する. Errorはexpression型の値の一つとし, 定義されていない演算やErrorを含んだ演算を行った場合にErrorを返すようにする. また, is_reducible関数内では便宜上Errorも簡約不可能とみなす.

```
type expression =
  (* 省略 *)
  | Error

let show exp =
  let rec showexp = function
    (* 省略 *)
    | Error -> "error"
  in "<< " ^ showexp exp ^ " >>"

let rec reduce = function
    Add (a, b) ->
      begin match (a, b) with
          (Num x, Num y) -> Num (x + y)
        | (x, y) when is_reducible x -> Add (reduce x, y)
        | (x, y) when is_reducible y -> Add (x, reduce y)
```

```
      | _ -> Error
    end
| Multiply (a, b) ->
    begin match (a, b) with
        (Num x, Num y) -> Num (x * y)
      | (x, y) when is_reducible x -> Multiply (reduce x, y)
      | (x, y) when is_reducible y -> Multiply (x, reduce y)
      | _ -> Error
    end
| Lessthan (a, b) ->
    begin match (a, b) with
        (Num x, Num y) -> Boolean (x < y)
      | (x, y) when is_reducible x -> Lessthan (reduce x, y)
      | (x, y) when is_reducible y -> Lessthan (x, reduce y)
      | _ -> Error
    end
| If (cond, t, f) ->
    begin match (cond, t, f) with
        (Boolean p, t, f) -> if p then t else f
      | (p, t, f) when is_reducible p -> If (reduce p, t, f)
      | _ -> Error
    end
|  _ as o -> o
```

総評

感想

今回の実装を中心とした意味論の演習によって，これまでの数学的な理論だけではなく実際の言語処理系の実装において どのようにプログラム意味論が用いられるかをなんとなくつかめてもらうことができたと考えている． 一方で反省点としては プログラミング言語 OCaml を初めて触った人や一回生にとっては 今回のヴァリアントとパターンマッチを多用した実装は 少しハードルの高いものだったのではないかという点が挙げられる．

第8週

文責: 西見元希

7週目に作った抽象構文木を評価することのできるインタプリタを拡張し，変数と代入の機能を実装した.

実装の方針と前提知識

その式が参照できる変数を，「変数名の文字列と値のペア」のリストを作って保持することで変数の参照を実装する． もし簡約途中で変数を発見した場合，そのリストを参照することで値に置換できる． 変数の定義として局所変数の定義が可能なlet式を合わせて実装する． この「変数名の文字列と値のペア」のリストを環境(environment)と呼ぶ.

実装

変数と let 式の定義

まず，expression 型の定義を拡張した．合わせて，直感的に記述するため string に name という alias をつけた． Var は変数を表す expression 型の値であり，変数名の文字列を保持する．Let は let 式を表す expression 型の値である.

```
type name = string
type expression =
    (* 省略 *)
  | Var of name
  | Let of name * expression * expression
```

show 関数の拡張

次に，show 関数を Var と Let に対応させた． 式が環境を持ち回るので環境を表示する機能も追加した.

```
let show (env, exp) =
  let rec show_exp exp = match exp with
    (* 省略 *)
    | Var name -> name
    | Let (name, value, body) ->
      "let " ^ name ^ " = " ^ show_exp value ^ " in " ^ show_exp body
  in
  let show_var pair =
    "(" ^ fst pair ^ ":" ^ show_exp (snd pair) ^ " )"
  in
    print_endline @@ "env = [" ^ String.concat " " (List.map show_var_pair env) ^ "]";
    print_endline @@ "exp = " ^ show_exp exp
```

reduce 関数の拡張

次に，reduce を修正した．簡約の際に環境を参照できるよう， 引数を環境 env と式 exp のペアとし，返り値も同じく env と exp のペアを返すようにした． reduce の定義の概要は次の通り.

- これまでの式の再帰部分では受け取った環境をそのまま渡してやればよい.
- Var name の式は name の値を assoc 関数を用いて環境から検索する.
- Let (name, value, body) の式は次のように条件分岐する
  - value が Error ならば (env, Error) を返す
  - value が簡約可能ならば (env, Let(name, value を簡約したもの, body)) を返す
  - value が簡約不可能ならば ((name, value)::env, body) を返す (つまり環境に value を追加)

最終的な reduce 関数は次のようになった.

```
let rec reduce (env, exp) = match exp with
  | Add (a, b) ->
      begin match (a, b) with
          (Num x, Num y) -> (env, Num (x + y))
        | (x, y) when is_reducible x ->
            (env, Add (snd (reduce (env, x)), y))
        | (x, y) when is_reducible y ->
            (env, Add (x, snd(reduce (env, y))))
        | _ -> (env, Error)
      end
  | Multiply (a, b) ->
      begin match (a, b) with
          (Num x, Num y) -> (env, Num (x * y))
        | (x, y) when is_reducible x ->
            (env, Multiply (snd(reduce (env, x)), y))
        | (x, y) when is_reducible y ->
            (env, Multiply (x, snd(reduce (env, y))))
        | _ -> (env, Error)
      end
  | Lessthan (a, b) ->
      begin match (a, b) with
          (Num x, Num y) -> (env, Boolean (x < y))
        | (x, y) when is_reducible x ->
            (env, Lessthan (snd(reduce (env, x)), y))
        | (x, y) when is_reducible y ->
            (env, Lessthan (x, snd(reduce (env, y))))
        | _ -> (env, Error)
      end
  | If (x, y, z) ->
      begin match (x,y,z) with
          (Boolean p, t, f) -> if p then (env, t) else (env, f)
        | (p,t,f) when is_reducible p ->
            (env, If (snd(reduce (env, p)),t,f))
        | _ -> (env, Error)
      end
  | Var name -> (env, List.assoc name env)
  | Let (name, value, body) ->
      begin match value with
          Error -> (env, Error)
        | v when is_reducible v ->
            (env, Let (name, snd (reduce (env, value)), body))
        | _ -> ((name, value)::env, body)
      end
  | _ as v -> (env, v)
```

run 関数の修正

最後にrun関数を環境を持ち回るように修正した.

```
let run exp =
  let rec _run (env, exp) =
    show (env, exp);
    if is_reducible exp
      then _run (reduce (env, exp))
      else (env, exp)
  in
  _run ([], exp)
```

最終的な実行例

```
# let e =
  Let ("x", Add(Num 3, Num 3),
    Let ("y", Multiply (Var "x", Num 3), Add (Var "y", Var "x")));;

# run e;;
env = []
exp = let x = (3 + 3) in let y = (x * 3) in (y + x)
env = []
exp = let x = 6 in let y = (x * 3) in (y + x)
```

```
env = [(x:6)]
exp = let y = (x * 3) in (y + x)
env = [(x:6)]
exp = let y = (6 * 3) in (y + x)
env = [(x:6)]
exp = let y = 18 in (y + x)
env = [(y:18) (x:6)]
exp = (y + x)
env = [(y:18) (x:6)]
exp = (18 + x)
env = [(y:18) (x:6)]
exp = (18 + 6)
env = [(y:18) (x:6)]
exp = 24
- : (string * expression) list * expression =
([("y", Num 18); ("x", Num 6)], Num 24)
```

総評

感想

実装による操作的意味論の実践を行うことで，理論の学習において特に理解に難があったと思われる． エラーを含めた推論と変数と環境関連の推論規則の理解が進んだのではないかと感じられた． 初めて触る慣れない言語で大規模なコードを書くのはかなり難易度は高かったと思われるが， 今後異なる言語でこの分野を学習したり処理系を自作したりするときに OCaml が如何に簡潔に書けていたかを実感するものと思われる． 担当者としてもコードと活動資料の両方を用意するのは大変であったが，実りのある活動になったと感じている．