ソフトウェア構成特論 第1回

大学院理工学研究科 電気電子情報工学専攻　篠埜 功

2014年4月10日

1　はじめに

ソフトウェアの基礎的な事項、特に、プログラムの形式的扱いにおいて有用な概念である集合の帰納的定義 (inductive definition) および帰納法 (induction)、近年有用性が認識されつつあるプログラミング言語の型システム (type system)、およびプログラムの意味論 (semantics) について解説を行う。帰納法については整礎帰納法 (well-founded induction) について解説し、数学的帰納法 (mathematical induction) や構造帰納法 (structural induction) などはその例であることを示す。型システムについては、関数型言語およびオブジェクト指向言語の核となる部分を対象として具体的な型システムを提示して解説する。意味論は命令型言語、関数型言語、およびオブジェクト指向言語の核となる部分を対象とし、主に操作的意味論 (operational semantics) について解説する。

この講義の達成目標は以下の3つとする。

1. 集合を帰納的に定義し、帰納法によりその集合に関する簡単な性質を証明できるようになる。
2. 単純な型システムについて、短いプログラムが型について整合性を持つことを示せるようになる。
3. 単純な言語の操作的意味が与えられたとき、短いプログラムの意味を示せるようになる。

このようなことを勉強あるいは研究するのは、現実社会のソフトウェア開発の問題点を解決していくことを目的としている。ただ、現時点では現実とはかなりの距離がある（と私は感じている）。ソフトウェアの根源的な部分に興味がある人にはこの講義の受講を勧めたいが、そうでない場合にはこの講義から得られるものが大学院修了後に直接役に立つとは限らないということを認識した上で受講するかどうかを決めて欲しい。この講義では、ML系言語やJavaなどの実際の言語との対応を示しながら上記の事柄を解説していきたい。この講義では、情報工学科2年次のプログラミング言語論程度の内容を前提とし、それよりも基礎寄りの内容を解説する。

以下の5冊を参考書とする。

- Glynn Winskel, *The Formal Semantics of Programming Languages — An Introduction*, The MIT Press, 1993.
- Benjamin C. Pierce, *Types and Programming Langauges*, The MIT Press, 2002.

- John C. Mitchell, *Foundations for Programming Languages*, The MIT Press, 1996.
- John C. Mitchell, *Concepts in Programming Languages*, Cambridge University Press, 2002.
- Ravi Sethi, *Programming Languages — Concepts & Constructs 2nd edition*, Addison Wesley, 1996.

講義資料は毎回講義用 web page

`http://www.sic.shibaura-it.ac.jp/~sasano/lecture/lecture.html`

に置く。内容は変更を行う可能性がある。変更した場合は講義用 web page に明示する。

2 集合の帰納的定義および帰納法

帰納的に定義された集合の要素が何らかの性質を満たすことを示す際に帰納法が用いられる。プログラミングの世界、特に関数型言語の世界でよく用いられるのが構造帰納法である。構造帰納法や高校等で勉強する数学的帰納法は整礎帰納法の例である。

2.1 数学的帰納法 (mathematical induction)

中学、高校等で習う数学的帰納法は、自然数について成り立つ性質を証明する際に用いる。自然数の集合の要素（つまり自然数）について成り立つ性質を証明する際に用いられるのが数学的帰納法である。

例 1 任意の 1 以上の自然数 n について、1 から n までの和は $n(n+1)/2$ と等しい。

一般に、任意の 1 以上の自然数 n について $P(n)$ が成立することを示したい場合、以下の 2 つを示せばよい。

- $P(1)$ が成立する
- 1 以上の任意の自然数 k について、$P(k) \Rightarrow P(k+1)$ が成立する

これが数学的帰納法である。上記の例の場合、示したい性質は、

$$P(n) : 1 \text{ から } n \text{ までの和は } n(n+1)/2 \text{ と等しい}$$

である。

自然数の集合を帰納的に定義することにより、数学的帰納法が正しいことを示すことができる。

2.2 構造帰納法 (structural induction)

自然数と同様、プログラムは再帰的な構造を持っている。また、関数型言語において再帰的なデータ型がよく用いられる。プログラムの集合や再帰的なデータ型の要素の集合は帰納的に定義される。このような集合の要素について成り立つ性質を証明する際に用いられるのが構造帰納法である。

2.3 整礎帰納法 (well-founded induction)

上記でのべた数学的帰納法や構造帰納法は整礎帰納法の特別な場合である。整礎帰納法は整礎な順序が定義されている集合の要素について成り立つ性質を証明する際に用いる。整礎帰納法を理解すれば必要に応じて様々な帰納法を自分で作り上げて使うことができる。

3 型システム

この講義では静的型付き言語 (statically typed langauge) について説明する。動的型付き言語 (dynamically typed language) については対象外とする。

3.1 型

ディジタルコンピュータにおいては、情報は数値（有限種類の記号の並び、大抵の場合、ビット列）で表される。同じ数値であっても、別の情報を表しうるので、あるビット列がどういう意味を持つかを定めるのが型である。C 言語では、char, int, double 等の型がある。このようなものを基本型といい、基本型を組み合わせて得られるのが複合型である。C 言語では、配列型、構造体型、ポインタ型、共用体型などが複合型を構築する手段として提供されている。ML 系言語では、直積、レコード、ヴァリアント、参照、再帰的データ型などが複合型を構築する手段として提供されている。

3.2 型システム

型システム (type system) とは、与えられたプログラムが型に関して整合性を持っているかどうかを検査するための体系である。

3.3 動的型付き言語、静的型付き言語の利点、欠点

- 静的型付き言語の例: C, C++, Java, ML, Haskell など
- 動的型付き言語の例: Lisp, Emacs Lisp, Scheme, Ruby など

静的型付き言語では、コンパイル時に型の整合性の検査をするため、実行時に型エラーが起こらないことを保証することができる。これはソフトウェアの正しさを部分的に保証している。型に関するプログラムの正当性の検査は、実用上使える、非常に有用な手段と考えられる。

例えば、C 言語で、

```
int main (void)
{
  int a[3];
  a = 1;
  return 0;
}
```

などはコンパイル時に `a = 1` の部分で型の不整合が検出されエラーになる。型について不整合のあるプログラムを書くことは非常に多く、それがすべて取り除かれることにより非常に多くのバグを除くことができるということである。

これに対し、動的型付き言語では実行時に型の整合性の検査を行うので、型が合わなかった場合はエラーで実行が止まったり例外が発生したりしてしまう。例えば、大規模なシステムにおいてほとんど実行されない部分に型の不整合があった場合に、ソフトウェアの出荷前の検査で見つからず、出荷後にエラーが見つるということになりえるが、静的型付き言語ではそのようなエラーを事前に発見することができる。また、動的型付き言語で行う実行時の型検査を静的型付き言語では省略できるので効率が良い。ただし、静的型付き言語では、動的型付き言語に比べプログラミングの柔軟性が損なわれるという欠点がある。

また、静的型付き言語では、コンパイル時に型が定まることにより、型情報による最適化を行えるという利点がある。つまり、どちらが良いとは一概には言えず、プログラマの好みやどのようなプログラムを作るか等による。プログラムの正当性が重要視される場合には静的型付き言語を用いるのが良いと考えられる。

参考 任意に与えられたプログラムがある性質（入出力関係）を満たしているかどうかを判定するプログラムは一般には存在しない。つまり、例えばプログラミングの課題で、自動採点プログラムを作成するのは一般に不可能である。それに対し、ある特定のプログラムがある性質を満たしていることを示すことは、定理証明器などを使えばできる場合がある。ただし現時点では定理証明器を使う方法が実用的なソフトウェアで用いられている例を（私は）知らない。

4 操作的意味論

プログラムがどのような挙動をするか、例えば、関数の返り値、画面、キーボード、ネットワーク等の外部機器とのやりとり等について、明確に（形式的に）論じるものが意味論である。意味論には大きく以下の 3 つがある。

- 表示的意味論 (denotational semantics)
- 公理的意味論 (axiomatic semantics)
- 操作的意味論 (operational semantics)

表示的意味論とは、プログラムの集合から、プログラムが表すもの（意味）の集合への写像である。ある程度複雑な言語（再帰的なデータ型や再帰関数の定義をする言語）の意味の集合は完備半順序集合 (CPO, complete partial order) とする。1970 年代に Dana Scott がラムダ計算に対する表示的意味論を構築するために CPO を用いる方法を考案した。

公理的意味論とは、プログラムが満たす性質を導くための論理体系である。命令型言語の核となる部分に対して Charles Antony Richard Hoare（略して Tony Hoare）が考案したのが Hoare logic（ホーア論理）である。ホーア論理についてはプログラミング言語論の講義で解説した。

操作的意味論とは、プログラムの動作を記述するものである。操作的意味論は大きく以下の 3 つに分かれる。

- small step semantics

- big step semantics (natural semantics、自然意味論)
- SECD machine(SECD 機械) 等、抽象的な機械の動作を記述したもの

プログラミング言語論の講義で極めて小さな C 言語のサブセットについて意味を定義したが、それが big step semantics である。

表示的意味論についてはシステム理工学専攻の「Computational Models」、公理的意味論については、電気電子情報工学専攻の「プログラミング言語特論」で扱われている。

現在広く使われている言語の中では Standard ML と Scheme が操作的意味論で定義されている。その他の言語は自然言語（英語など）で意味が定義されている。例えば、C 言語は ISO 規格で意味が英語で記述されている。Ruby など、広く使われている言語でも自然言語での意味の記述すら存在しなかった言語もある。（Ruby は自然言語での意味の記述が最近行われ、2012 年に ISO 規格になっている。）

参考 SECD machine は Peter J. Landin が 1960 年代に考案した抽象機械であり、ラムダ式を SECD machine の機械語に翻訳 (compile) することによりラムダ式の意味を記述しようとしたものである。S は stack, E は environment, C は code, D は dump を表す。