ソフトウェア論理
Logic in Computer Software

亀山幸義

筑波大学 コンピュータサイエンス専攻

第 1 週

変わるもの vs 変わらないもの

含蓄の深い CM (by イチロー)

- 「変わらなきゃ」
- 「変わらなきゃ、も、変わらなきゃ」
- 「変わらなきゃ、も、変わらなきゃ、も、変わらなきゃ、···」

コンピュータ科学

- Mathematics: 静的、普遍の原理 (「定理」)、変わらないもの。
- Computer Science: 動的、面白い、現代的、進歩が速い etc.
- プログラム: 極めて動的、計算が進む、止まってられない。
- コンピュータの世界に「変わらないもの」なんてあるのか？

変わらないもの

- プログラムの検証 (by 水谷先生): ループ不変条件。
- プログラムの意味を考える: 変わらないものを探す旅。

概要

- 後半の授業では、関数プログラミングに基づいたプログラム言語論を学ぶ。
- このため、PCFと呼ばれる計算体系を設定する。
- PCFの構文と型付け規則を与える。
- PCFでのプログラミングをやってみる。

PCF: Programming Language for Computable Functions [Gordon Plotkin]

Term (項)

$e ::= 0 \mid 1 \mid \ldots$	`int` の定数
$\mid$ `true` $\mid$ `false`	`bool` の定数
$\mid x$	変数
$\mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2$	いろいろな式の構成
$\mid$ `if` e_1 `then` e_2 `else` e_3	条件式
$\mid \lambda x.\ e \mid e_1\ e_2$	ラムダ式とその適用
$\mid \lambda(\overline{x}).\ e \mid e_1\ (\overline{e_2})$	ラムダ式とその適用 (多引数)
$\mid$ `fix` $f(\overline{x}).\ e$	再帰呼び出し

f, x は変数, e は式。
$\overline{x}$ は 1 個以上の変数の列。$\overline{e}$ は 1 個以上の式の列。

項の例

$1 + x$

$\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ 2$

$\texttt{if}\ x = 0\ \texttt{then}\ y + 1\ \texttt{else}\ f(2 + z)$

$f\ 0$

$f(g(h\ 0))$

$\lambda x.\ x$

$\lambda x.\ x + 1$

$(\lambda x.\ x + 1)\ 3$

$(\lambda x.\ x + 1)\ ((\lambda x.\ x + 1)\ 3)$

$(\lambda f.\ f(f\ 3))(\lambda x.\ x + 1)$

$\lambda x.\ \ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f(x - 1)$

$\texttt{fix}\ f(x).\ \ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ f(x - 1)$

ラムダ式 (Lambda Expression)

$\lambda x.\ e$

- $f(x) = e$ となる関数 f のこと。
- 関数の入力が x、出力が e.
- 関数を使うときは $f(103)$ と書いてもよいが、$f\ 103$ でもよい。
- PCF では、関数をデータとして扱うことができる。
- 例: $\lambda f.\ f(x) + 1$.

再帰呼出し (Recursive Call)

$\texttt{fix}\ f(x).\ e$

- $f(x) = e$ となる関数 f のこと。???
- 単純なラムダ式との違い: e の中に f が現れてもよい。
- 再帰呼び出し。
- ある種の関数型言語 (OCaml) では、`let rec f x = e` というように rec と書く。
- fix: fixed point, fixpoint, 不動点。

プログラミング例 (1 から x までの和)

$$\texttt{fun1} \stackrel{def}{=} \texttt{fix}\ f(x).\ \ \texttt{if}\ x = 0\ \texttt{then}\ 0\ \texttt{else}\ (x + (f\ (x - 1)))$$

$$\begin{aligned}
\texttt{fun1}(3) &\rightsquigarrow \texttt{if}\ 3 = 0\ \texttt{then}\ 0\ \texttt{else}\ (3 + (\texttt{fun1}\ (3 - 1))) \\
&\rightsquigarrow \texttt{if false then}\ 0\ \texttt{else}\ (3 + (\texttt{fun1}\ (3 - 1))) \\
&\rightsquigarrow 3 + (\texttt{fun1}\ (3 - 1)) \\
&\rightsquigarrow 3 + (\texttt{fun1}\ (2)) \\
&\rightsquigarrow 3 + (\texttt{if}\ 2 = 0\ \texttt{then}\ 0\ \texttt{else}\ 2 + (\texttt{fun1}\ (2 - 1))) \\
&\rightsquigarrow^* 3 + (2 + (1 + 0)) \\
&\rightsquigarrow^* 6
\end{aligned}$$

プログラミング例 (階乗: 1 から x までの積)

$$\mathtt{fun2} \stackrel{def}{=} \mathtt{fix}\ f(x).\ \mathtt{if}\ x = 0\ \mathtt{then}\ 1\ \mathtt{else}\ x * (f\ (x-1))$$

$$\begin{aligned}
\mathtt{fun2}(3) &\rightsquigarrow \mathtt{if}\ 3 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * (\mathtt{fun2}\ (3-1)) \\
&\rightsquigarrow \mathtt{if}\ \mathtt{false}\ \mathtt{then}\ 1\ \mathtt{else}\ 3 * (\mathtt{fun2}\ (3-1)) \\
&\rightsquigarrow 3 * (\mathtt{fun2}\ (3-1)) \\
&\rightsquigarrow 3 * (\mathtt{fun2}\ (2)) \\
&\rightsquigarrow 3 * (\mathtt{if}\ 2 = 0\ \mathtt{then}\ 1\ \mathtt{else}\ 2 * (\mathtt{fun2}\ (2-1))) \\
&\rightsquigarrow^* 3 * (2 * (1 * 1)) \\
&\rightsquigarrow^* 6
\end{aligned}$$

プログラミング例 (2 引数のかけ算)

$$\mathtt{fun3} \stackrel{def}{=} \mathtt{fix}\ f(x,y).\ \mathtt{if}\ x = 0\ \mathtt{then}\ 0\ \mathtt{else}\ y + f\ (x-1, y)$$

$$\begin{aligned}
\mathtt{fun3}(3,2) &\rightsquigarrow \mathtt{if}\ 3 = 0\ \mathtt{then}\ 0\ \mathtt{else}\ 2 + \mathtt{fun3}\ (3-1, 2) \\
&\rightsquigarrow 2 + \mathtt{fun3}\ (3-1, 2) \\
&\rightsquigarrow 2 + \mathtt{fun3}\ (2, 2) \\
&\rightsquigarrow^* 2 + (2 + \mathtt{fun3}\ (1, 2)) \\
&\rightsquigarrow^* 2 + (2 + (2 + \mathtt{fun3}\ (0, 2))) \\
&\rightsquigarrow^* 2 + (2 + (2 + 0)) \\
&\rightsquigarrow^* 6
\end{aligned}$$

プログラミング例 (止まらない計算)

$$\mathtt{fun4} \stackrel{def}{=} \mathtt{fix}\ f(x).\ f(x)$$

$$\begin{aligned}
\mathtt{fun4}\ (0) &\rightsquigarrow \mathtt{fun4}(0) \\
&\rightsquigarrow \cdots
\end{aligned}$$

構文と意味 (Syntax vs Semantics)

構文と意味: 永遠の対立?

- なぜ「意味」を考えるのだろうか？
- なぜ「意味」を考えずにいられるのか？

プログラム言語の構文と意味

- 構文論: どういう文字列がプログラムになっているかを決める。(cf. 日本語の構文)
- 意味論: プログラムを走らせると、その答えは何かを決める。

プログラム言語に意味を与える3つの主要な方法

- Operational Semantics (操作的意味論)
  Specify how we compute the value of e.
- Axiomatic Semantics (公理的意味論)
  Specify how we can reason about (the properties of) e.
- Denotational Semantics (表示的意味論、あるいは、外延的意味論)
  Specify what is the corresponding function (or some mathematical object) to e.

この授業前半で、水谷先生が教えた Hoare 論理は、公理的意味論の例。(ソフトウェア検証では、それが一般的)
プログラム言語論では、操作的意味論、表示的意味論を採用することが多い。

形式意味論 vs 非形式的意味論

- どんなプログラム言語の解説書でも、与えらえたプログラムを実行するとどんな結果になるか、一応は書いてある。
- しかし、ほとんどの解説書では、「どんなプログラムでも、計算結果がわかる」ほど詳細には書いていない。
- プログラムの解析、検証、(保守、再利用 etc.) をするためには、これではいけない。
- 形式意味論: 形式と意味の合体? ではなくて、形式的に (厳密に) 意味を与えたもの。
- 今週は、PCFの意味を非形式的に考えるところまで。形式意味論は来週。

計算の例の続き

$$\texttt{double} = \lambda f.\ \lambda x.\ f(f\ x)$$
$$\texttt{inc} = \lambda y.\ y + 1$$

とするとき、(double inc) 1 を計算する。

$$\begin{aligned}(\texttt{double inc})\ 10 &\rightsquigarrow (\lambda x.\ (\texttt{inc}(\texttt{inc}\ x)))\ 10\\ &\rightsquigarrow \texttt{inc}(\texttt{inc}\ 10)\\ &\rightsquigarrow \texttt{inc}(10+1)\\ &\rightsquigarrow (10+1)+1\\ &\rightsquigarrow^* 12\end{aligned}$$

計算の例の続き

$$\texttt{gt} \stackrel{def}{=} \texttt{fix}\ f(x,y).\ \texttt{if}\ x = 0\ \texttt{then false else}$$
$$\texttt{if}\ y = 0\ \texttt{then true else}\ f\ (x-1, y-1)$$

$$\begin{aligned}&\texttt{gt}\ (2,\ 3)\\ &\rightsquigarrow^* \texttt{gt}\ (1,\ 2)\\ &\rightsquigarrow^* \texttt{gt}\ (0,\ 1)\\ &\rightsquigarrow^* \texttt{false}\end{aligned}$$

$$\begin{aligned}&\texttt{gt}\ (3,\ 2)\\ &\rightsquigarrow^* \texttt{gt}\ (2,\ 1)\\ &\rightsquigarrow^* \texttt{gt}\ (1,\ 0)\\ &\rightsquigarrow^* \texttt{true}\end{aligned}$$

fix の話

再帰呼びだし (recursive call) を使って、

$$f(x) = N$$

という関数 f を定義したい。N は f と x を含む (かもしれない) 項とする。この関数 f を、PCF では、

$$\texttt{fix}\ f(x).\ N$$

と書く。

fix の意味

$$f(x) = \cdots f(e_1) \cdots f(e_2) \cdots$$

- その厳密な意味は?
- 展開: 左辺が現れるたびに、右辺に書き換える。
- 無限に展開した「極限」を表すのではないか？

$$\texttt{fact} \stackrel{def}{=} \texttt{fix}\ f(x).\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1))$$

$$\begin{aligned}
&\texttt{fact}(3) \rightsquigarrow \texttt{if}\ 3 = 0\ \texttt{then}\ 1\ \texttt{else}\ 3 * (\texttt{fact}\ (3-1)) \\
&\texttt{fact}(2) \rightsquigarrow \texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * (\texttt{fact}\ (2-1)) \\
&\quad \cdots
\end{aligned}$$

直感的には OK だが、「無限に展開したもの」のままでは、そのプログラムの性質を (有限時間内で) 解析したり、検証したりできない。

fix の意味

$$\texttt{fact} \stackrel{def}{=} \texttt{fix}\ f(x).\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1))$$

まず、

$$F \stackrel{def}{=} \lambda f.\ \lambda x.\ \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f\ (x-1))$$

を考えよう。

- F は、「関数 f と数 x をもらうと、1 回展開して計算した結果を返す」関数である。(F 自身は、再帰呼びだしを使っていない。)
- n 回展開したものを f_n と書くと、$f_{n+1}(x) = (F(f_n))(x)$ となる。(ただし、$f_0(x) = \text{undefined}$ とする。)
- $\texttt{fact}$ は言わば f_∞ なので、$\texttt{fact}(x) = (F(\texttt{fact}))(x)$ となる。
- この場合、$\texttt{fact}$ を、上記の方程式の不動点と言う。

fix の意味

関数に関する方程式:

$$f(x) = (F(f))(x)$$

がどんな x に対しても成立するようにしたい。

- 与えられるもの F: 「関数を一回、展開した関数に変換する」関数。
- 見つけるもの f: 「関数を一回、展開した関数に変換する」関数。
- x は任意の数 (をあらわす式)。

F が前述のものであるとき、その解 (の 1 つ) として、$\texttt{fact}$ は与えられる。

fix の意味

ラムダ計算における定理 (再帰定理, Recursion Theorem):
F を任意のラムダ式とするとき、どんなラムダ式 x に対しても、

$$f(x) = (F(f))(x)$$

を満たすラムダ式 f が存在する。

- 現実的な意味: どんな再帰関数の定義を書いても、(少なくともラムダ式としては) 存在する。
- 目茶苦茶な再帰関数定義を書いても、まともな関数が定まるという意味?
- 目茶苦茶な再帰関数定義を書いても、何らかの関数が定まるという意味?
- 目茶苦茶な再帰関数定義を書いても、何らかの部分関数が定まる。
  - $\texttt{fact}(-3)$ は値を持たない (値が未定義; undefined)。

今日のまとめ

プログラミング言語のコアとしてのラムダ計算

- 多くのプログラム言語のモデル
  - ラムダ計算に何らかの機能追加をしたものとしてとらえられる
  - C, Java, ...
- 最近のプログラム言語は、関数 (クロージャ) を直接持っている。
  - 関数型言語: Lisp, Scheme, ML (SML, OCaml, F#), Haskell...
  - 「関数型」以外: Perl, Ruby, JavaScript, ...
- 関数プログラミング
- プログラミング言語のコアとしてのラムダ計算
- 形式と意味
- 何のために意味を考えるのか？

Quiz

Q1. 以下の関数は何を計算するものか？
$\texttt{fun2} \stackrel{def}{=} \texttt{fix}\ f(x).\ \texttt{if}\ \texttt{gt}(x, 100)\ \texttt{then}\ x - 10\ \texttt{else}\ f\ (f\ (x + 11))$
Q2. 「C 言語は関数型プログラム言語でない」との言い方に違和感を持った人もいるだろう。C 言語では、関数そのものをデータとすることはできないが、「関数へのポインタ」をデータとすることはできる。(他の関数の引数や返り値 (return value) として関数へのポイントを取ることはできる。) では、C 言語は関数型プログラム言語なのだろうか？C 言語ではできない関数に対する操作はあるだろうか？
また、Java 言語はどうか？ 関数はクラス定義におけるメソッドみたいなものだから、クラス (のインスタンスとしてのオブジェクト) は、関数より複雑なデータ構造ではないだろうか。では、すべてのオブジェクト指向言語は関数型プログラム言語であると言えないだろうか？

(Partial) Answer

Q1. 以下の関数は何を計算するものか？
$\texttt{fun2} \stackrel{def}{=} \texttt{fix}\ f(x).\ \texttt{if}\ \texttt{gt}(x, 100)\ \texttt{then}\ x - 10\ \texttt{else}\ f\ (f\ (x + 11))$
A1. $x > 100$ のとき $\texttt{fun2}(x) = x - 10$ で、$x \leq 101$ のとき $\texttt{fun2}(x) = 91$ となる関数。(John McCarthy の 91 関数)

補足: 少し計算してみると、以下の事実がわかる。
・$90 \leq x \leq 100$ なら $\texttt{fun2}(x) = \texttt{fun2}(\texttt{fun2}(x + 11)) = \texttt{fun2}(x + 1)$。よって $\texttt{fun2}(x) = \texttt{fun2}(101) = 91$
・$79 \leq x \leq 89$ なら $\texttt{fun2}(x) = \texttt{fun2}(x + 1)$。よって $\texttt{fun2}(x) = \texttt{fun2}(90) = 91$
以下同様。(正式には $101 - x$ に関する数学的帰納法で証明)

(Partial) Answer

Q2. C 言語ではできなくて、関数型プログラム言語でできることは？
A2 の例 1. 「ない」、なぜなら、C 言語で、関数型プログラム言語の処理系 (コンパイラあるいはインタープリタ) を記述することができるので、その処理系を通して、どんな関数型プログラム言語のプログラムも実行できるはず。
A2 の例 2. 処理系を作る等の大がかりなことを許さなければ、「ある」、具体的には「実行中に関数を作ること」は、ファイルに文字列を書きこんでコンパイルしなおしたり等をしない限りできない。(C 言語の通常のプログラムでできるのは「最初から定義している関数」へのポインタを取ること。実行中に関数を生成することは基本的にはできない。)

(Partial) Answer

Q3. Java 言語のオブジェクトは、関数型言語における関数を表現できるか？
A3.「できる」、なお、一般的にいって、オブジェクト指向言語のオブジェクトの方が (関数型言語の) 関数より複雑なデータなので、「オブジェクト指向言語は関数型言語でもある」と言ったところで、あまり意味がない。