実践離散数学と計算の理論
第3回　計算可能性

大阪大学大学院　工学研究科　電気電子情報通信工学専攻
王　イントウ

前回の質問

Q：高階論理と高階関数についてもっと説明してほしい

- 命題計算(真偽値の計算)は零階論理
- 一階述語計算(階数=1)は一階論理:
  ある述語の引数が項であるとき、その述語の階数は1
  - 項（term）とは、変数または定数、あるいは項である引数に関数を適用したもの
  - 例：x, a, f(x, g(b)), $S(x) \wedge T(x)$
- 高階述語計算(階数≥2)は高階論理: 二階論理、三階論理…
  ある述語の引数のうち最も高い階数がnであるとき、その述語の階数はn+1
  ・例：$S(x) \wedge T(S)$　　Sは階数1、そしてTは階数２

  $p(f(x)) \wedge q(f)$　　pは階数1、**fは階数1**、そしてqは階数2

  f(x)は項であるが、fは集合(つまり集合または述語)である。

Q：高階論理と高階関数についてもっと説明してほしい

- 高階関数：関数の引数に他の関数を取る or 関数を返り値として返す(e.g. 再帰関数)
- 高階関数では、階数が型で決まる。(型システムでラムダ計算の部分に入る)
- ほとんどの関数は1階関数で、ほとんどの高階関数は2階関数になる。
- http://jats-ug.metasepi.org/doc/ATS2/ATS2TUTORIAL/c988.html
  ご参考ください。

Q：現量子の分配や、同値などについて形式証明の中で記述する際は、どのようにに記載すればよろしいのでしょうか？ 命題倫理のほうでは、英語の略語のように記述していましたが、そのようなものが存在するのでしょうか？？

- 限量子の分配などの同値関係は論理公理に属する
- 論理公理を先にラベルをつけて証明の途中に引用する

A1: $\exists x(p(x) \vee q(x)) \equiv \exists x\ p(x) \vee \exists x\ q(x)$

高階述語論理のイメージ

- 高階な述語は述語自体を引数に取る述語。P(x, y)、Q(P)のQ。
- または、述語自体を限量する。P(x, y)、∀PとしたときのP。
- 述語自体について言及するときに用いる：eg. 再帰的関数
- P(x, y)を、「xはyの親である」としたとき、これは親子関係だが、親子関係とは一体何なのか？どういう関係なのかなど、関係について記述
- 例えば、Q(P)を、「Qを親子関係はXXXである」と記述すると、Qは人間関係を引数にとって、分類する（YES/NOを返す）ような述語になる

計算可能性
（Computability）

停まらない計算（1/2）
ゴールドバッハ予想（Goldbach's Conjecture）

すべての2より大きな偶数は2つの素数の和で表すことが出来る

```
goldbach?() {
    for each (n > 2 and n is even) {
        for each (1 <= x < n and x is prime) {
            for each (1 <= y < n and y is prime) {
                if (n == x + y) {
                    goto found;
                }
            }
        }
        return false;
found:
    }
    return true;
}
```

停まらない計算（2/2）
フェルマーの最終定理（Fermat's Last Theorem）

$n \geq 3$となる自然数 n について

$$x^n + y^n = z^n$$

となる自然数の組み合わせ x, y, z は存在しない

```
fermat?() {
    for each (n > 2, x > 0, y > 0, z > 0) {
        if (x^n + y^n = z^n) {
            return false;
        }
    }
    return true;
}
```

停止問題（1/2）（Halting Problem）

仮定：ある計算が停止するか判定可能なhalt?という関数を定義可能

```
halt?(fun, x) {
    if (fun(x)は停止する) {
        return true;
    } else {
        return false;
    }
}
```

仮定が成立しない証明：

halt?(m0, x)=trueの時に、
無限ループとなるような関数m0を構成

```
m0(x) {
    if (halt?(m0, x) = true) {
        loop; /* 無限ループ */
    } else {
        return true;
    }
}
```

停止問題（2/2）（Halting Problem）

halt?(m0, x)がtrueと仮定

　つまり、halt?はm0を停止すると判定

　しかし、m0は停止しないので矛盾

halt?(m0, x)がfalseと仮定

　つまり、halt?はm0を停止しないと判定

　しかし、m0は停止するので矛盾

よって、仮定が矛盾

```
halt?(fun, x) {
    if (fun(x)は停止する) {
        return true;
    } else {
        return false;
    }
}

m0(x) {
    if (halt?(m0, x) = true) {
        loop; /* 無限ループ */
    } else {
        return true;
    }
}
```

原始帰納的関数（Primitive Recursive Function）（1/2）

0. 絶対止まることがわかってるような関数

1. 定数0は原始帰納的関数（引数0の関数）

2. $S(x) = x + 1$ と定義される関数 S は原始帰納的関数
   後継者関数（successor function）とも呼ぶ

3. $f(x_1, \ldots, x_n) = x_i$ と定義される関数 f は原始帰納的関数
   射影（projection）とも呼ぶ

4. f と g_i が原始帰納的関数のとき、以下の関数 h は原始帰納的関数
   h を f と g_i の合成（composition）とも呼ぶ

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

原始帰納的関数（Primitive Recursive Function）（2/2）

5. f と g が原始帰納的関数のとき、以下の関数 h は原始帰納的関数
   このように定義する方法を原始帰納法（Primitive Recursion）と呼ぶ

$$h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$
$$h(S(n), x_1, \ldots, x_n) = g(n, h(n, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

加算（Addition）

$$id(x) = x$$

$$2nd(x_1, x_2, x_3) = x_2$$

$$S \circ 2nd(x_1, x_2, x_3) = S(2nd(x_1, x_2, x_3)) = S(x_2)$$

$$plus(0,y) = id(y) = y$$

$$plus(S(x), y) = S \circ 2nd(x, plus(x, y), y) = S(plus(x, y))$$

加算の例

$$
\begin{aligned}
plus(3,2) &= S(plus(2,2)) \\
&= S(S(plus(1,2))) \\
&= S(S(S(plus(0,2)))) \\
&= S(S(S(2))) \\
&= S(S(3)) \\
&= S(4) \\
&= 5
\end{aligned}
$$

加算の定義

$plus(0,y) = y$

$plus(S(x), y) = S(plus(x, y))$

減算（Subtraction）

$$pred(0) = 0$$
$$pred(S(x)) = x$$

$$monus(x,0) = x$$
$$monus(x, S(y)) = pred(monus(x, y))$$

$$monus(x, y) = \begin{cases} x - y & if\ x \geq y \\ 0 & if\ x < y \end{cases}$$

以降 $x -^m y := monus(x, y)$ と定義。

減算の例（1/2）

$$
\begin{aligned}
monus(3,2) &= pred(monus(3,1)) \\
&= pred(pred(monus(3,0))) \\
&= pred(pred(3)) \\
&= pred(2) \\
&= 1
\end{aligned}
$$

減算の定義

$pred(0) = 0$

$pred(S(x)) = x$

$monus(x, 0) = x$

$monus(x, S(y)) = pred(monus(x, y))$

減算の例（2/2）

$$
\begin{aligned}
monus(1,3) &= pred(monus(1,2)) \\
&= pred(pred(monus(1,1))) \\
&= pred(pred(pred(monus(1,0)))) \\
&= pred(pred(pred(1))) \\
&= pred(pred(0)) \\
&= pred(0) \\
&= 0
\end{aligned}
$$

減算の定義

$pred(0) = 0$

$pred(S(x)) = x$

$monus(x,0) = x$

$monus(x, S(y)) = pred(monus(x, y))$

乗算（Multiplication）

$$mul(0,y) = 0$$

$$mul(S(x), y) = plus(mul(x, y), y)$$

乗算の例

$$
\begin{aligned}
mul(3,2) &= plus(mul(2,2),2) \\
&= plus(plus(mul(1,2),2),2) \\
&= plus(plus(plus(mul(0,2),2),2),2) \\
&= plus(plus(plus(0,2),2),2) \\
&= plus(plus(2,2),2) \\
&= plus(4,2) \\
&= 6
\end{aligned}
$$

乗算の定義

$mul(0,y) = 0$

$mul(S(x), y) = plus(mul(x, y), y)$

if式（If Expression）

$$true = 0$$

$$false = S(0) = 1$$ （非０の自然数なら何でも良いが、代表値として1を利用）

$$if(0, y, z) = y$$

$$if(x, y, z) = z$$

同値比較

$$isZero(0) = true$$
$$isZero(x) = false$$

$$equal(x, y) = isZero(plus(x -^m y, y -^m x))$$

以降 $x = y \overset{\mathrm{def}}{=} equal(x, y)$ と定義。

同値比較の例（1/2）

$$
\begin{aligned}
equal(3,3) &= isZero(plus(3 -^m 3, 3 -^m 3)) \\
&= isZero(plus(0,0)) \\
&= isZero(0) \\
&= 0
\end{aligned}
$$

同値比較の定義

$isZero(0) = true$

$isZero(x) = false$

$equal(x, y) = isZero(plus(x -^m y, y -^m x))$

同値比較の例（2/2）

$$\begin{aligned} equal(4,2) &= isZero(plus(4 -^m 2, 2 -^m 4)) \\ &= isZero(plus(2,0)) \\ &= isZero(2) \\ &= 1 \end{aligned}$$

同値比較の定義

$$isZero(0) = true$$

$$isZero(x) = false$$

$$equal(x,y) = isZero(plus(x -^m y, y -^m x))$$

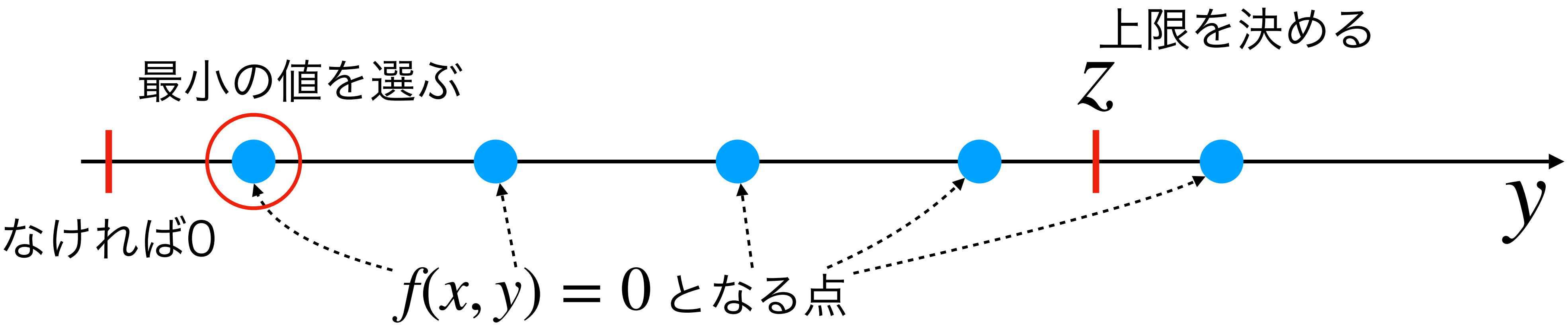
有界最小化（Bounded Minimization）

$\mu y\,.\,(A)$ は条件 A を満たす最小の y を選ぶ。なければ0。とする。

$f(x, y)$ という原始帰納的関数があったとき、

$$g(x, z) = \mu y\,.\,(y < z \wedge f(x, y) = 0)$$

という関数 $g(x, z)$ を得ることを有界最小化という。



有界最小化関数の原始帰納的定義

$$h(x, z, 0) = 0$$

$$\begin{aligned} h(x, z, S(w)) = \ &\textbf{if } f(x, z -^m S(w)) = 0 \textbf{ then } z -^m S(w) \\ &\textbf{else } h(x, z, w) \end{aligned}$$

$$g(x, z) = h(x, z, z)$$

除算（商の計算）

$$div(x, z) = \mu y\,.\,(y < S(x) \wedge x -^{m} (mul(y, z) + (z -^{m} 1)) = 0)$$

$$eg\,.\,div(x, 2) = div2(x) = \mu y\,.\,(y < x + 1 \wedge x -^{m} (2y + 1) = 0)$$

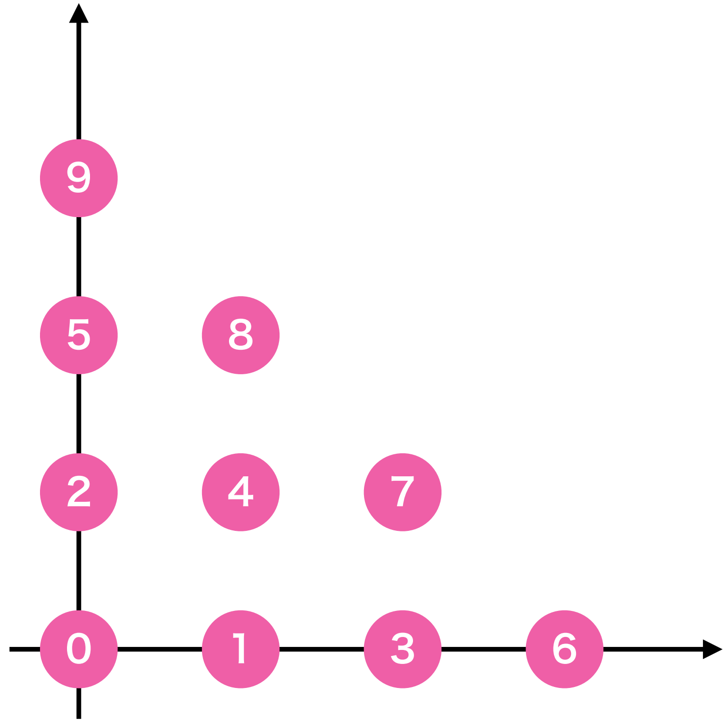
2値の符号化

2つの自然数を、1つの自然数に符号化可能。

符号化の方法はいろいろある。
符号化出来るという事実が重要。

符号化関数の例

$$p(x, y) = div((x + y + 1) \times (x + y), 2) + y$$



復号

$$p_1(p(x, y)) = x$$
$$p_2(p(x, y)) = y$$

という復号関数は以下のように定義可能。

$$p_1(z) = \mu x\,.\,(x < z + 1 \wedge p_1'(z, x) = z)$$
$$p_2(z) = \mu y\,.\,(y < z + 1 \wedge p_2'(z, y) = z)$$

$$p_1'(z, x) = p(x, \mu y\,.\,(y < z + 1 \wedge p(x, y) = z))$$
$$p_2'(z, y) = p(\mu x\,.\,(x < z + 1 \wedge p(x, y) = z), y)$$

有限列の符号化と復号

自然数の有限列 $x_0, x_1, \ldots, x_{n-1}$ は以下のように符号化可能。

$$p(x_0, p(x_1, \ldots, p(x_{n-1}, 0)\ldots))$$

符号 z から x_i を取り出す関数は以下のように定義可能。

$$a(z, i) = p_1(b(z, i))$$
$$b(z, 0) = z$$
$$b(z, S(i)) = p_2(b(z, i))$$

部分関数

集合 A から B への関数 f を

$$f : A \to B$$

とかき、A の部分集合 C から B への関数 g を以下のようにする。

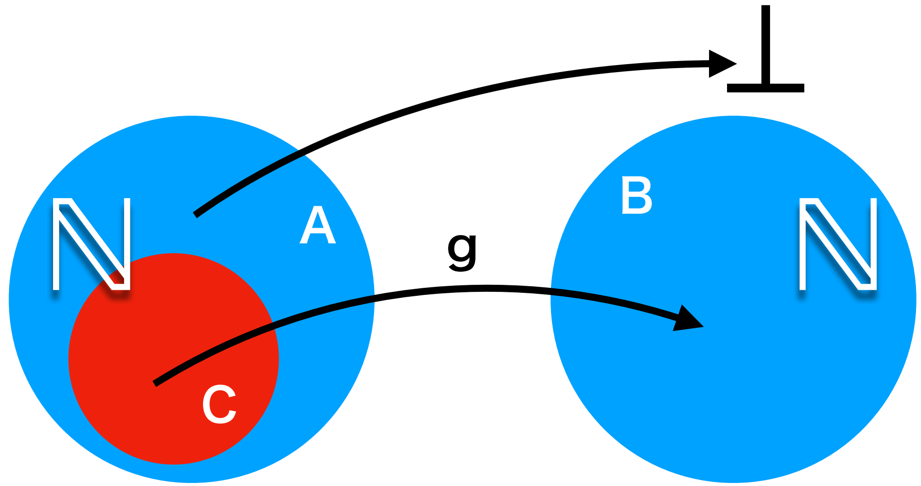
$$g : C \to B$$

このとき、g を次のように定義すると

$$g : A - C \to \perp$$

g は A から $B \cup \{ \perp \}$ への関数とみなすことができる。

このような g を f の部分関数と呼ぶ。



自然数から自然数への関数の部分関数

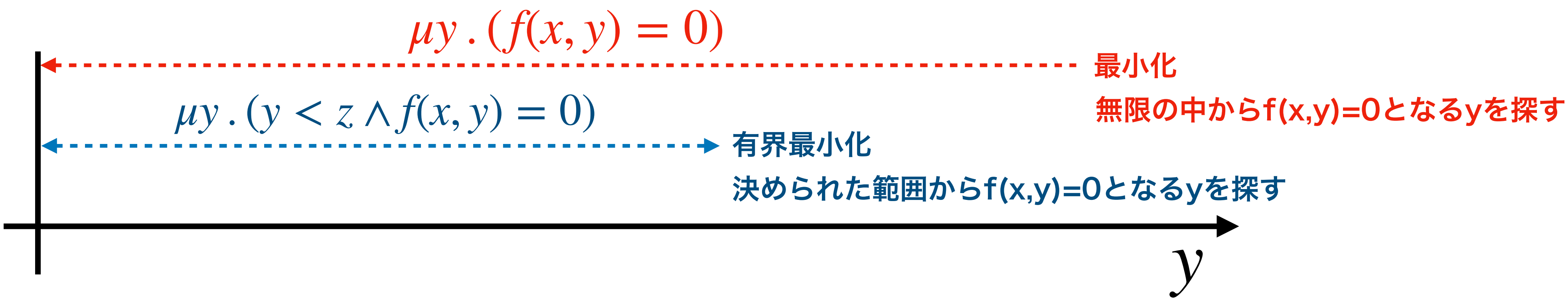
部分帰納的関数（Partially Recursive Function）

$f(x, y)$ と $g(x)$ という原始帰納的関数があったとき、

$$h(x) = g(\mu y \,.\, (f(x, y) = 0))$$

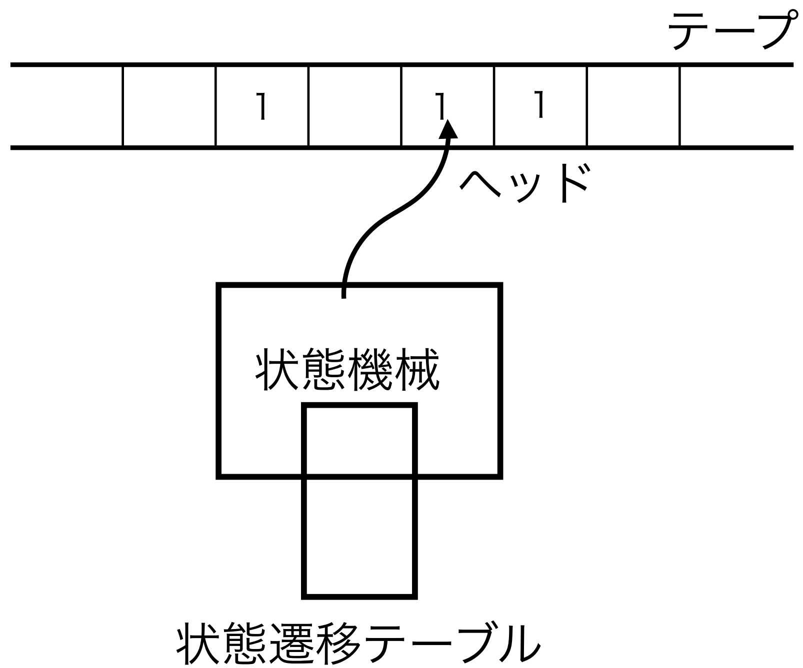
という $h(x)$ を部分帰納的関数という。

ただし、$f(x, y) = 0$ となる y がない場合は、$h(x) = \perp$ と定義する。



チューリングマシン

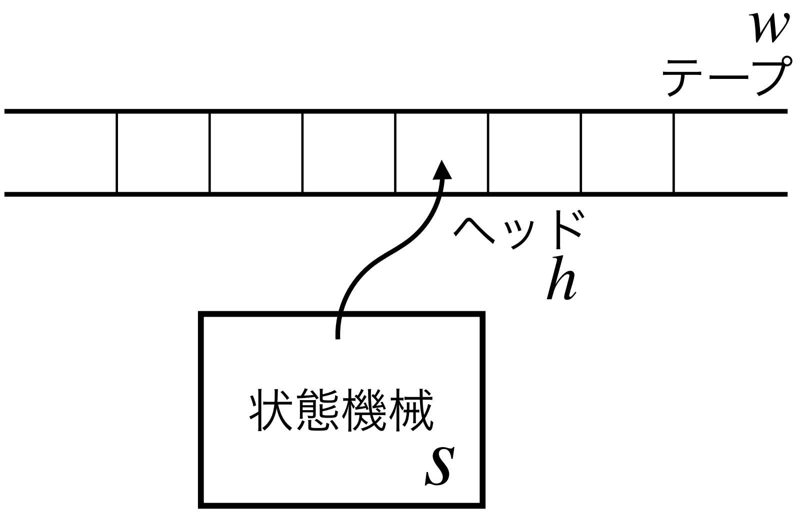
- テープ
- ヘッド
- 状態機械
- テーブル
  - 入力
    - 現在状態
    - ヘッドが指している文字
  - 出力
    - 次の状態
    - ヘッドが指す先に印字すべき文字
    - ヘッドの動かすべき方向
      - 右、左、動かさない

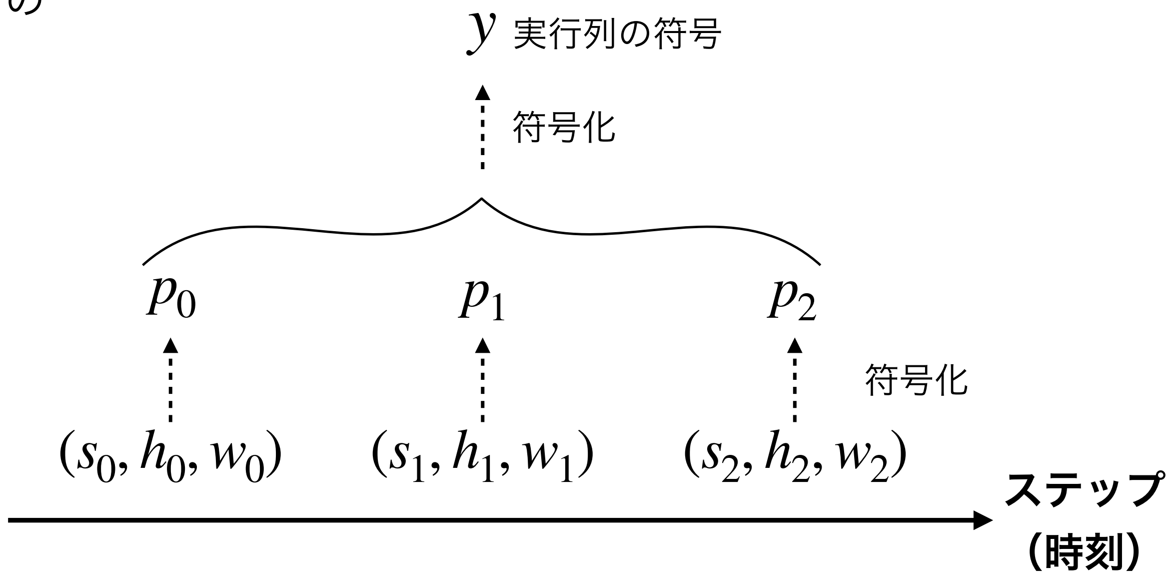


模造品：https://www.youtube.com/watch?v=ivPv_kaYuwk

チューリングマシンの実行列の符号化

$(s, h, w) \to p$ ：状態、ヘッド位置、
テープ文字を符号化したもの





チューリングマシンを実行する部分帰納的関数

チューリングマシンMに対して、以下のような原始帰納的関数を定義可能。

$$f_M(x, y) = \begin{cases} 0 & y\text{が入力 } x \text{ に対するチューリングマシン} M \text{ の実行列} \\ 1 & \text{それ以外} \end{cases}$$

$g(x)$ をチューリングマシンの実行列から結果を取り出す関数とすると、

$$h(x) = g(\mu y \,.\, (f_M(x, y) = 0))$$

という部分帰納的関数がチューリングマシンを実行する関数となる。

クリーネのT述語（Kleene's T-predicate）

$f_M(x, y)$ は、チューリングマシンのテーブル e に依存するため、
$T(e, x, y) = f_M(x, y)$
という原始帰納的関数 T を定義できる。

この T をクリーネのT述語といい、e をインデックスと呼ぶ。

チューリングマシンM が入力 x に対して停止することと、
$T(e, x, y) = 0$となる y が存在することは同値。

停止しない計算

ある x で $f_M(x, y) = 0$ となる y を求めてみる

(s_0, h_0, w_0) この実行列では終わっていないので、$f_M(x, y) = 1$ になる

(s_0, h_0, w_0) (s_1, h_1, w_1) この実行列でも終わっていないので、$f_M(x, y) = 1$ になる

(s_0, h_0, w_0) (s_1, h_1, w_1) (s_2, h_2, w_2) この実行列でも終わっていないので、$f_M(x, y) = 1$ になる

(s_0, h_0, w_0) (s_1, h_1, w_1) (s_2, h_2, w_2) ・・・

ステップ（時刻）

停止問題再訪（1/3）

p をチューリングマシンを実行する関数、e をインデックス
x を入力とするとチューリングマシンの停止判定関数は
以下のように定義可能と仮定。

$$h(p(e,x)) = \begin{cases} 0 & T(e,x,y) = 0 \text{ となる自然数 } y \text{ が存在} \\ 1 & \text{存在しない} \end{cases}$$

停止問題再訪（2/3）

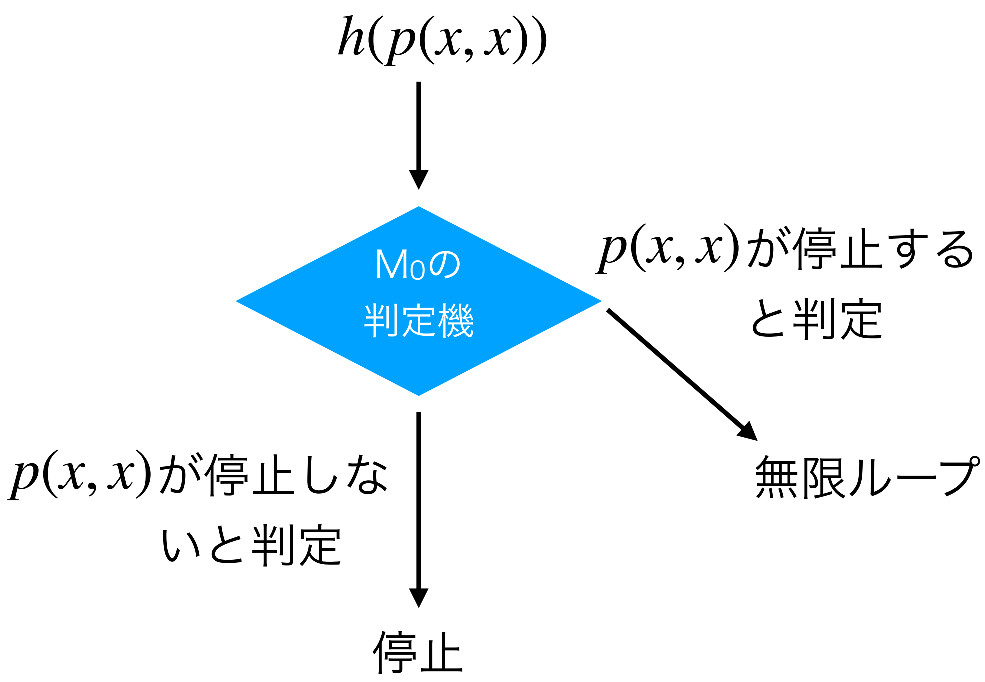
$$h(p(e,x)) = \begin{cases} 0 & T(e,x,y)=0\text{となる自然数 } y \text{ が存在} \\ 1 & \text{存在しない} \end{cases}$$

というように停止判定が可能であると仮定すると

$h(p(x,x)) = 0$　ならば停止しない

$h(p(x,x)) = 1$　ならば停止する

というチューリングマシン M_0 を構成可能



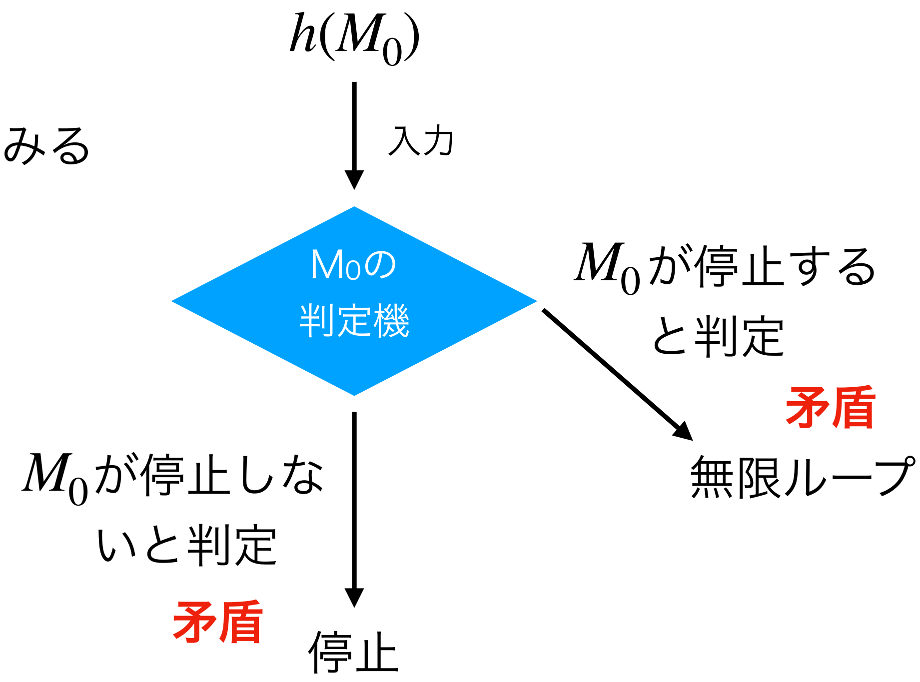
停止問題再訪（3/3）

$h(p(x,x)) = 0$　ならば停止しない
$h(p(x,x)) = 1$　ならば停止する

というチューリングマシン M_0 に自分自身を入力してみる
M_0 のインデックスを e_0 とすると

$h(p(e_0, e_0))$ が0でも1でも矛盾が生じる

よって、停止判定可能という仮定が誤り。



Coq

帰納型（Inductive Type）

- Coqでは型の定義を帰納的に行うことが可能
- 数学の帰納的な定義と一致

自然数の定義（ペアノ算術）

$$N = \begin{cases} 0 & \textbf{ゼロ} \\ S(N) & \textbf{N + 1} \end{cases}$$

$$\mathbb{N} = \{0, \underset{1}{S(0)}, \underset{2}{S(S(0))}, \underset{3}{S(S(S(0)))}, \cdots\}$$

Coqでの自然数の定義

```
Inductive nat: Set :=
| O : nat
| S : nat -> nat.
```

```
O, S O, S (S O), S (S (S O))), ...
0,  1,     2,          3,     ...
```

リスト

- 帰納型を用いるとリストが定義可能
- Aという型のリストは以下のように定義される（Aは型引数）
- nilは空リスト
- consはリストの先頭に追加する操作

```
Inductive list (A:Type) : Type :=
| nil : list A
| cons : A -> list A -> list A
```

PythonとCoqのリストの対応

Python	Coq	Coq（別表記）
`[ ]`	`nil`	`[ ]`
`[1, 2, 3]`	`cons 1 (cons 2 (cons 3 nil))`	`[1; 2; 3]`
`[3, 5].insert(0, 4)`	`cons 4 (cons 3 (cons 5 nil))`	`4::[3; 5]`

Coqでの関数適用

- 関数適用はC言語などと書き方が異なるため注意
- Coqでは「関数 スペース 引数」と書くと関数適用になる（括弧が必要ない）

C言語	Coq
func(arg)	func arg
func(arg1, arg2)	func arg1 arg2
funcA(funcB(arg))	funcA (funcB arg)

関数の型

- 関数の型は->で定義

- 例：
  nat -> nat は、natを受け取ってnatを返す関数型
  list (nat -> nat) は、natを受け取ってnatを返す関数のリスト型

パターンマッチ

- 帰納型をパターンによって分解可能

1を引く操作

```
match S O with
| O => 式
| S n => 式
end.
```

リストのパターンマッチ

```
match [1; 2; 3] with
| [] => 式
| h::t => 式
end.
```

非再帰関数定義

- 非再帰関数は、Definitionで定義可能

```
Definition proj (n:nat) (l:list nat) : nat :=
  nth n l O.
```

- projが関数名
- nがnat型の引数、lがlist nat型の引数
- 返り値の型がnat

再帰関数定義

- 再帰関数はFixpointで定義

```
Fixpoint app_gs (gs:list ((list nat) -> nat))
                (args:list nat) (l:list nat) : (list nat) :=
  match gs with
  | h::t => app_gs t args ((h args)::l)
  | [] => rev l
  end.
```

let式

- 変数束縛はlet式を利用

C言語	Python	Coq
`int a = 10;`	`a = 10`	`let a := 10 in`
`int a = 20;` `int b = 30;`	`a = 20` `b = 30`	`let a := 20 in` `let b := 30 in`

Coqによる
原始帰納的関数の実装

Coqでの実装方法

- 原始帰納的関数は、nat型のリストを引数に取り、nat型を返す関数とする

succ関数

2. $S(x) = x + 1$ と定義される関数 S は原始帰納的関数
   後継者関数（successor function）とも呼ぶ

```
Definition succ (n:list nat) : nat :=
  match n with
  | n'::_ => S n'
  | _ => 0
  end.
```

射影関数

3. $f(x_1, \ldots, x_n) = x_i$ と定義される関数 f は原始帰納的関数
   射影（projection）とも呼ぶ

```
Definition proj (n:nat) (l:list nat) : nat :=
  nth n l 0.
```

合成

4. f と g_i が原始帰納的関数のとき、以下の関数 h は原始帰納的関数
   h を f と g_i の合成（composition）とも呼ぶ

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

```
Fixpoint app_gs (gs:list ((list nat) -> nat)) (args:list nat) (l:list nat): (list nat) :=
  match gs with
  | h::t => app_gs t args ((h args)::l)
  | [] => rev l (* revはリストを逆順にする関数 *)
  end.

Definition compose
           (f:(list nat) -> nat)
           (gs:list ((list nat) -> nat))
           (args:list nat)
  : nat :=
  f (app_gs gs args []).
```

原始帰納法（1/2）

5. f と g が原始帰納的関数のとき、以下の関数 h は原始帰納的関数
   このように定義する方法を原始帰納法（Primitive Recursion）と呼ぶ

$$h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$

$$h(S(n), x_1, \ldots, x_n) = g(n, h(n, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

```
Fixpoint recurse'
           (f:(list nat) -> nat) (* f(x1, ..., xn) *)
           (g:(list nat) -> nat) (* g(n, h(n, x1, ..., xn), x1, ..., xn) *)
           (x0:nat)
           (args:list nat) (* x1 ... xn *)
  : nat :=
  match x0 with
  | O => f args
  | S n => g (n :: (recurse' f g n args) :: args)
  end.
```

原始帰納法（2/2）

```
Definition recurse
           (f:(list nat) -> nat) (* f(x1, ..., xn) *)
           (g:(list nat) -> nat) (* g(n, h(n, x1, ..., xn), x1, ..., xn) *)
           (args:list nat) (* x0 ... xn *)
  : nat :=
  match args with
  | h::t => recurse' f g h t
  | [] => 0
  end.
```

plus関数の例

$$plus(S(x), y) = S \circ 2nd(x, plus(x, y), y)$$

```
Definition plus (l:list nat): nat :=
  match l with
  | a::b::[] =>
    let f := proj 0 in
    let g := compose succ [proj 1] in
    recurse f g [a; b]
  | _ => 0 (* エラー。plus関数の引数は必ず2つである。 *)
  end.
```

Coqのインストールと設定

Coq IDEを利用する方法（推奨）

- CoqIDEが簡単に利用可能
  https://github.com/coq/platform/releases/tag/2022.04.0

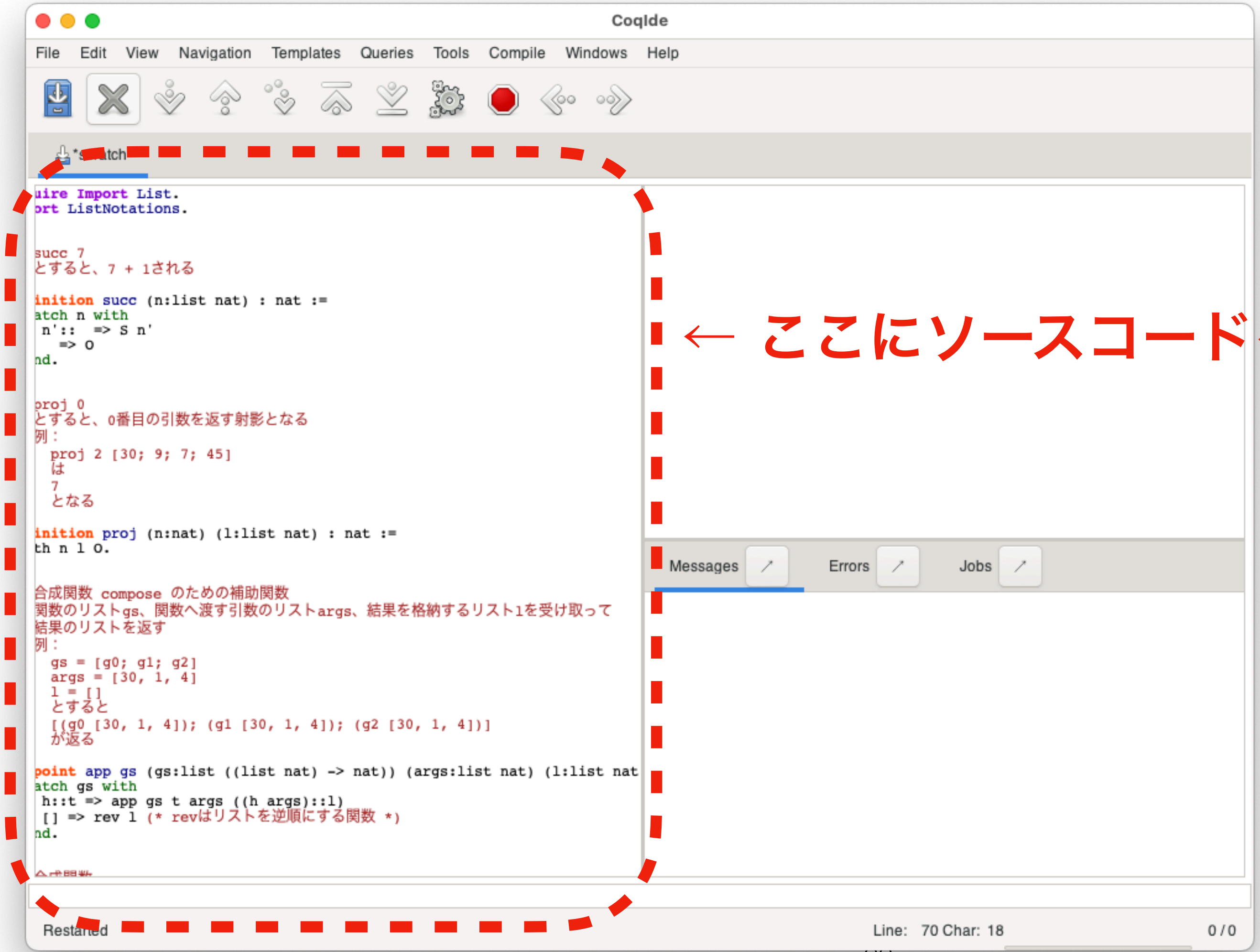
- Windowsは
  Coq-Platform-release-2022.04.0-version_8.15_2022.04-windows-x86_64_signed.exe
  インストール後、coqideを検索して起動

- Macは
  Coq-Platform-release-2022.04.0-version.8.15.2022.04-macos-signed.dmg
  アプリケーションフォルダにコピー後起動（セキュリティの設定で実行を許可する必要あり）

Dockerを利用する方法（プロ向け）

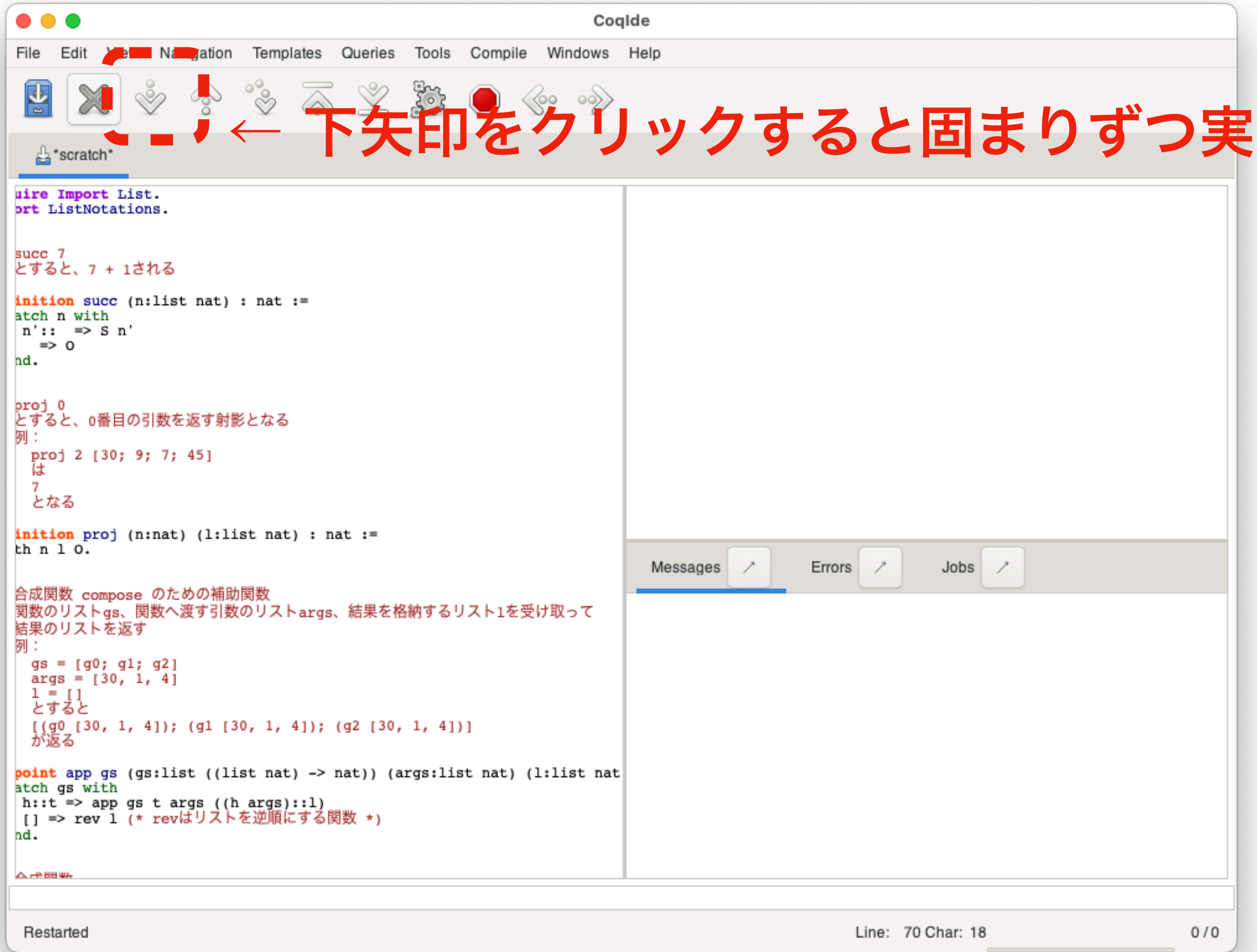
- CoqのDockerイメージを用いて、emacsで書いていく
  https://hub.docker.com/r/coqorg/coq/
- emacsにはproof-generalとcompany-coqをインストールすると良い
- 設定が複雑

CoqIDE



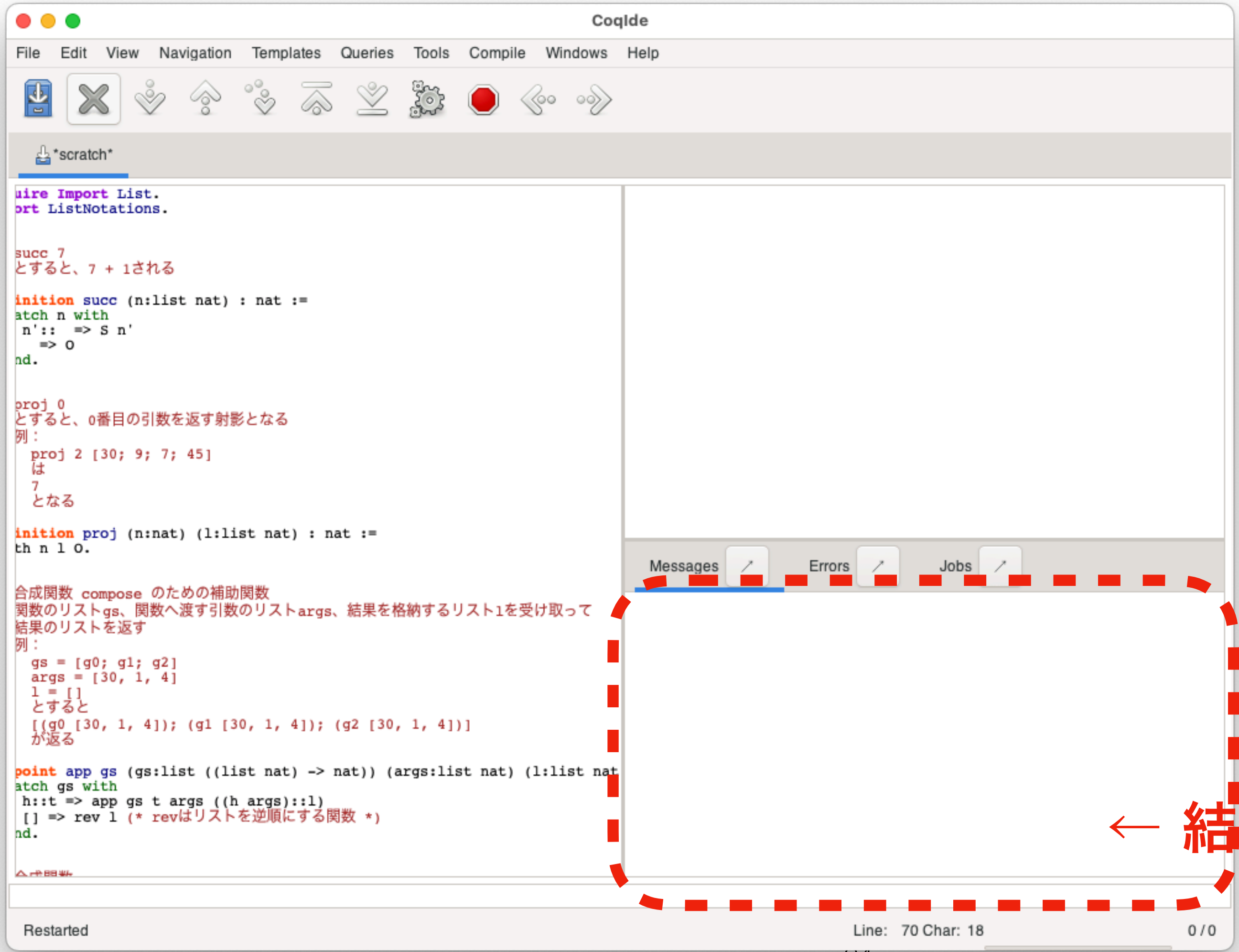
← ここにソースコードを書く

CoqIDE



← 下矢印をクリックすると固まりずつ実行

CoqIDE



← 結果がここに表示される

レポート

問題：チューリングマシン

チューリングマシンについて
自分なりにまとめて解説せよ

問題：停止問題

ある高校生向けの科学雑誌に「停止問題について」
と記事を書くことになったと仮定して、
停止問題について解説せよ。
解説記事はA4で1ページ以内とする。

問題：Coq

- Coqを用いて原始帰納的関数を2つ以上、自由に定義せよ
- 以下のURLのソースコードをもとにすること
  https://onl.sc/ReQLvhR
- 1つ以上は再帰を用いた関数を定義すること
- Compute命令で実行結果を表示して確認すること

レポート課題

- 本スライド中にある問題を解いてレポートとして提出せよ
- 締め切り：2022年6月20日 23:50 (JST)