工学部専門科目
「プログラミング言語」SICP
第3章〜Modularity, Objects, State〜
その4

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

May 20, 2014

今日のメニュー

3.3 変更可能データを使ったモデリング

3.3.5 制約の伝播

3.3.5 制約の伝播

- ふつうのプログラムでは入力と出力が厳密に区別される
- n 変数の関係式

$$(\text{ひずみ}) \cdot (\text{断面積}) \cdot (\text{弾性係数}) = (\text{応力}) \cdot (\text{長さ})$$
$$(\text{電圧}) = (\text{電流}) \cdot (\text{抵抗})$$

では，$(n-1)$ 変数の値が決まると最後のひとつが決まる

$\Longrightarrow$ 入力・出力の立場が自在に決められるプログラム!?

制約ネットワーク

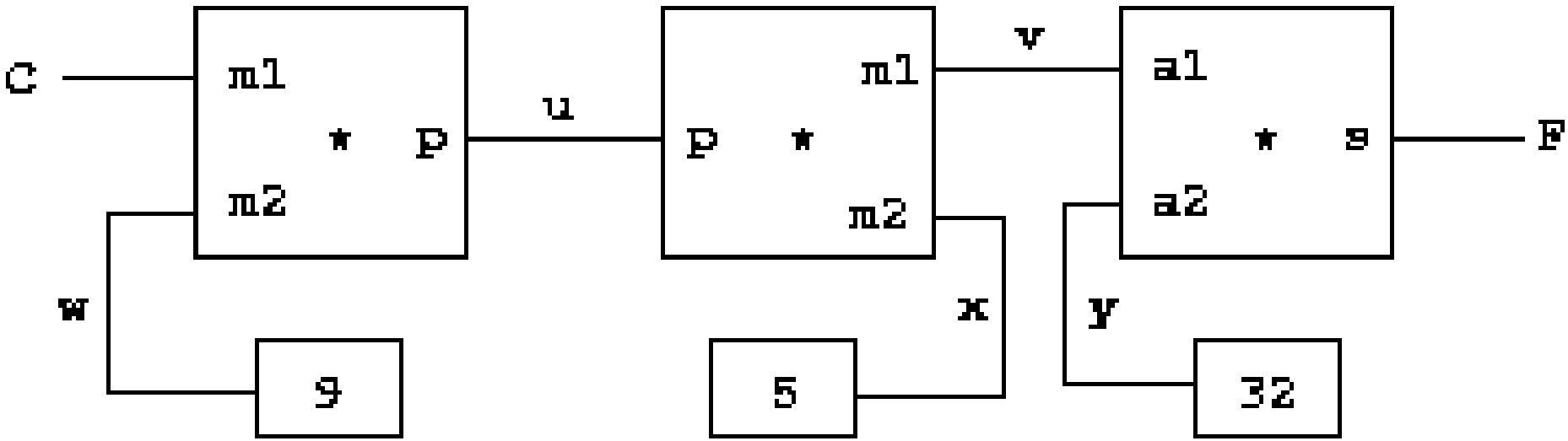
n 変数の関係式を「ネットワーク」で表現する

- 原始制約
  - $a = b + c$
  - $a = b \cdot c$
- コネクタ: ふたつの制約に現れる変数をつなぐ
  - 値を保持する
  - 「両端は同値である」という制約を課す

制約ネットワークの例

摂氏温度 (C) と華氏温度 (F) の換算式

$$9C = 5(F - 32)$$



- 線分がコネクタ
- 箱が原始制約

この節の内容

制約ネットワークを表現するための仕組みを実現する

- `(make-connector)`
  - **コネクタを作る**
- `(multiplier c1 c2 c3)`
  - **コネクタ** `c1, c2, c3` **間に制約** $c1 \cdot c2 = c3$ **を課す**
- `(adder c1 c2 c3)`
  - **コネクタ** `c1, c2, c3` **間に制約** $c1 + c2 = c3$ **を課す**
- `(constant n c)`
  - **コネクタ** `c` **の値が常に** `n` **になるように制約を課す**
- `(probe s c)`
  - **コネクタ** `c` **の値を観察し．** `c` **の値に変化があった時にメッセージ** `s` **とその値を表示する**

摂氏・華氏変換の表現

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v ...) (w ...) (x ...) (y ...))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

```
(define C (make-connector))
(define F (make-connector))

(celsius-fahrenheit-converter C F)

(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)
```

ネットワークを使う

```
(set-value! C 25 'user)
```

```
(set-value! F 212 'user)
```

```
(forget-value! C 'user)
```

```
(set-value! F 212 'user)
```

コネクタの操作

- (has-value? c)
  - コネクタ c に値がセットされたかを調べる
- (get-value c)
  - コネクタ c の値を得る
- (set-value! c n i)
  - コネクタ c の値を n にセットする．i はセットした人
- (forget-value! c r)
  - コネクタ c の値をリセットする．r はリセットする人 (セットした人と同じでなければならない)
- (connect c constr)
  - コネクタ c を制約 constr につなぐ

実装の方針

制約・コネクタをオブジェクトとして実現 (ただし制約オブジェクトの状態は変化しない)

- 制約オブジェクト (probe オブジェクトも含む)
  - 状態: つながっているコネクタ
  - 受け付けるメッセージ: `I-have-a-value`, `I-lost-my-value`
- コネクタオブジェクト
  - 状態変数
    - ★ `value`: 値
    - ★ `informant`: 最後に値をセットしたのが誰か (`#f` なら値はセットされていない)
    - ★ `constraints`: 接続されている制約オブジェクトのリスト
  - 受け付けるメッセージ: `has-value?`, `value`, `set-value!`, `forget`, `connect`

加算・乗算の制約オブジェクトの動作

- メッセージ `I-have-a-value` を受け取る $\Longrightarrow$
  - 接続されたコネクタ達から値を取り寄せる
  - 値が決まる場所があるなら，それを保持するためのコネクタに `set-value!` メッセージを送る
  - 値が決まらない場合はなにもしない
- メッセージ `I-lost-my-value` を受け取る $\Longrightarrow$
  - 接続されたコネクタ全てに一旦値をリセットするよう `forget` メッセージを送信
  - `I-have-a-value` を受けた時の動作を行い値の再計算

定数制約オブジェクトの動作

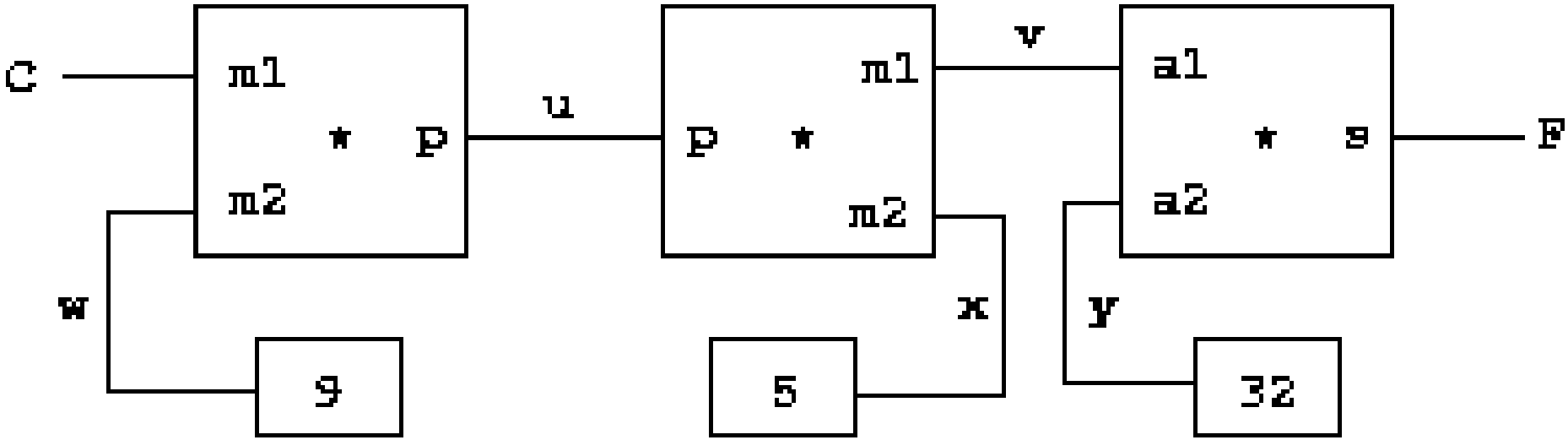
- オブジェクト作成時に与えられたコネクタに値をセットする
- が，後は何もしない

コネクタオブジェクトの動作

- メッセージ `set-value!` を受け取る $\Longrightarrow$
  - 既に値を保持していて，それとは違う値をセットしようとしているならエラー
  - 値と誰がセットしたかを記憶
  - 接続された制約オブジェクト全て (セットしたオブジェクト以外) に `I-have-a-value` を送信する
- メッセージ `forget` を受け取る $\Longrightarrow$
  - 前に値をセットした人とメッセージ送信者が同じかチェック (違うなら何もしない)
  - `informant` をクリア
  - 接続された制約オブジェクト全て (リセットしたオブジェクト以外) に `I-lost-my-value` を送信する

- メッセージ `connect` を受け取る $\Longrightarrow$
  - 制約オブジェクトを登録
  - 自分が値を持っている状態なら，その値を新しい制約オブジェクトに知らせる (`I-have-a-value`)

例で説明



probe オブジェクトの動作

- メッセージ `I-have-a-value` を受け取る $\Longrightarrow$
  - 接続されたコネクタの値を表示
- メッセージ `I-lost-my-value` を受け取る $\Longrightarrow$
  - 値が無くなった旨メッセージを表示

adder の定義 (1/3)

```
(define (adder a1 a2 sum) ;; 引数はコネクタ
  (define (process-new-value) ...)
  (define (process-forget-value) ...)
  (define (me request) ;; リクエストメッセージで場合わけ
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error ...))))
  ;; 初期化（接続）のための処理
  (connect a1 me) ;; 黄色地の名はメッセージ送信手続き
  (connect a2 me)
  (connect sum me)
  me)
```

adder の定義 (2/3)

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2)
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
                       (- (get-value sum) (get-value a1
                       me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1
                       (- (get-value sum) (get-value a2
                       me))))
  ...)
```

adder の定義 (3/3)

```
(define (adder a1 a2 sum)
  (define (process-new-value) ...)
  (define (process-forget-value)
    (forget-value! sum me)
    (forget-value! a1 me)
    (forget-value! a2 me)
    (process-new-value))
  ...)
```

multiplier の定義(1/3)

```
(define (multiplier m1 m2 product)
  (define (process-new-value) ...)
  (define (process-forget-value) ...)
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error ...)
  ( connect  m1 me)
  ( connect  m2 me)
  ( connect  product me)
  me)
```

multiplier の定義 (2/3)

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1)
                    (= (get-value m1) 0))
               (and (has-value? m2)
                    (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                       (* (get-value m1) (get-value m2))
                       me))
          ((and (has-value? product) (has-value? m1))
           ...)
          ((and (has-value? product) (has-value? m2))
           ...)))
```

multiplier の定義(3/3)

```
(define (multiplier m1 m2 product)
  (define (process-new-value) ...)
  (define (process-forget-value)
    (forget-value! product me)
    (forget-value! m1 me)
    (forget-value! m2 me)
    (process-new-value))
  ...)
```

constant の定義

```
(define (constant value connector)
  (define (me request)
    (error ...))
  (connect connector me)
  (set-value! connector value me)
  me)
```

probe の定義

```
(define (probe name connector)
  (define (print-probe value)
    (newline)
    (display "Probe: ")
    (display name)
    (display " = ")
    (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value)
    (print-probe "?"))
  (define (me request) ...) ;; ディスパッチャ
  (connect connector me)
  me)
```

メッセージ送信手続き

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))
(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

make-connector の定義(1/4)

```
(define (make-connector)
  (let ((value false)       (informant false)
        (constraints '()))
    (define (set-my-value newval setter) ...)
    (define (forget-my-value retractor) ...)
    (define (connect new-constraint) ...)
    (define (me request)
      (cond ((eq? request 'has-value?)
             (if informant true false))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error ...))))
    me))
```

make-connector の定義(2/4)

```
(define (make-connector)
  (let ((value false) ...)
    (define (set-my-value newval setter)
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter
                              inform-about-value
                              constraints))
            ((not (= value newval))
             (error ...))
            (else 'ignored)))
    ...))
```

make-connector の定義(3/4)

```
(define (make-connector)
  (let ((value false) ...)
    ...
    (define (forget-my-value retractor)
      (if (eq? retractor informant)
          (begin (set! informant false)
                 (for-each-except retractor
                                  inform-about-no-value
                                  constraints))
          'ignored))
    ...))
```

make-connector の定義(4/4)

```
(define (make-connector)
  (let ((value false) ...)
    ...
    (define (connect new-constraint)
      (if (not (memq new-constraint constraints))
          (set! constraints
                (cons new-constraint constraints)))
      (if (has-value? me)
          (inform-about-value new-constraint))
      'done)
    ...))
```

補助手続き

```
(define (for-each-except
          exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception)
           (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items)))))
  (loop list))
```

- `exception` を除く `list` の要素それぞれに `procedure` を適用する

メッセージ送信手続き

```
(define (has-value? connector)
  (connector 'has-value?))
(define (get-value connector)
  (connector 'value))
(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))
(define (forget-value! connector retractor)
  ((connector 'forget) retractor))
(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

宿題： 5/27(火) 午前10:30 締切

- Ex. 3.33
- **レポートには**
  - **考え方の説明**
  - **プログラムリストと考え方の対応**
  - **実行例**

  を示すこと
- **レポート** (pdf) **とプログラムファイルを提出システムを通じて提出**
- **友達に教えてもらったら、その人の名前を明記**
- web **は出典を明記** (**「同じ」回答は減点**)

来週の予定範囲 3.5.1〜3.5.2

予習ポイント:

- **ストリームとはどんなデータ構造か？ リストとの違いは何か？**
- **ストリームを使った素数列の計算の仕組み**
- **なぜ無限のデータ列が表現できるのか？**