簡約!? λカ娘

始まりの総集編

目次

第 0 章　関数型イカ娘とは! / この総集編について　@xhl_kogitsune　v

0.1　この総集編について v
0.2　関数型イカ娘とは v
0.3　関数型イカ娘アニメ紹介 vi

第 I 部　簡約! λ力娘 (2011 年夏)　1

第 1 章　β簡約！λ力娘　@xhl_kogitsune　3

1.1　λしなイカ ？ 3
1.2　SKI しなイカ? 5
1.3　有効活用しようじゃなイカ! 9

第 2 章　数をつくらなイカ？　@nushio　10

第 3 章　関数型イカガール　@tanakh　16

3.1　関数型イカガール第一話 16
3.2　関数型イカガール第二話 16
3.3　参考文献 24

第 4 章　加速しなイカ？　@nushio　26

4.1　夢のようなライブラリがあったじゃなイカ！ 26
4.2　GPU も、FPGA も、あるでゲソ！ 27
4.3　実アプリの性能と向き合わなイカ？ 28
4.4　そんなの、私が許さないでゲソ！ 29

第 5 章　OS を侵略しなイカ？　@master_q　32

5.1　HFuse でファイルシステムを侵略しなイカ？ 32
5.2　ヲレヲレファイルシステムを作ってみるでゲソ! 33
5.3　fuseGetFileStat はファイルサイズを有限時間で返答しなければイカん 34
5.4　ということで完成したでゲソ 35
5.5　この後は何を侵略するでゲソ？ 35

第 II 部　簡約!? λ力娘 (二期) (2011 年冬)　37

第 6 章　クラスタリングしなイカ？　@dekosuke　38

6.1　アルゴリズムとライブラリを決めなイカ？ 38
6.2　CSV をパーズしなイカ？ 40
6.3　実際にやってみるでゲソ 40
6.4　まとめでゲソ 41
6.5　参考文献でゲソ 41
6.6　kmeans-vector の license でゲソ 42

第 7 章　Copilot への希望と絶望の相転移　@master_q　**43**
7.1　Copilot はイカが？ . . . 43
7.2　Copilot DSL と C 言語コードの融合でゲソ . . . 46
7.3　Fibonacci.hs を例にして Copilot DSL の書き方を伝授するでゲソ! . . . 47
7.4　Wii リモコンで遊ぶでゲソ . . . 48
7.5　libc のないところでも操縦できるでゲッソ! . . . 51
7.6　けっきょく使えそうなんでゲソ？ . . . 55

第 8 章　加速しなイカ？！　@nushio　**57**
8.1　これまでのあらすじ . . . 57
8.2　進化しなイカ？ . . . 57
8.3　生存戦略しなイカ？ . . . 59
8.4　種をまかなイカ？ . . . 63
8.5　統計しなイカ？ . . . 64
参考文献 . . . 65

第 9 章　Java VM 上で tail call しなイカ？　@yryozo, @xhl_kogitsune　**66**
9.1　Java VM と関数型言語は相性が悪いでゲソ？ . . . 66
9.2　Java VM 上の関数型言語実装を紹介するでゲソ! . . . 70
9.3　Java VM 自体も侵略するでゲソ . . . 75
参考文献 . . . 83

第 10 章　関数型イカガール第 3 話　@tanakh　**86**

第 11 章　映画けいさん！　@tanakh　**91**

第 III 部　簡約!? 入力娘 (算) (2012 年夏)　**103**

第 12 章　運物語　@dif_engine　**105**
12.1　プロローグ . . . 105
12.2　ラーン・ユー・ア・ハスケル・フォー・グレート・グッド . . . 106
12.3　海の家「れもん」にて . . . 121
12.4　死闘 . . . 126
12.5　エピローグ . . . 131

第 13 章　RTS 海溝二万マイル　@master_q　**133**
13.1　RTS は Haskell コードを実行する VM でゲソ! . . . 133
13.2　[表層] Haskell の世界 . . . 134
13.3　潜水艦に乗り込もうじゃなイカ! . . . 134
13.4　[中深層] C 言語の世界 . . . 135
13.5　[漸深層] 幽霊船 . . . 140
13.6　Haskell の関数はどんなメモリ配置？ . . . 141
13.7　[深海層] cmm 言語の世界 . . . 143
13.8　浮上でゲッソ . . . 151
13.9　謝辞 . . . 152
13.10　参考文献 . . . 152

第 14 章　評価　@xhl_kogitsune　**153**
14.1　準備しなイカ? . . . 155
14.2　評価順序を変えなイカ? . . . 156

14.3 グラフ簡約しなイカ? . . . 161
14.4 Fully Lazy グラフ簡約 . . . 165
14.5 Optimal Reduction . . . 167
14.6 まとめと参考文献 . . . 173

第 15 章 オール・アバウト・ケーゾク・イン・スキーム @yryozo **175**
15.1 「継続」について勉強しなイカ? . . . 175
15.2 継続って誰が考えたんでゲソね? . . . 180
15.3 なんで Scheme と継続が関係あるのか不思議じゃなイカ? . . . 187
15.4 そろそろ CPS についてひとこと言っておくでゲソ . . . 193
15.5 「ゲンテーケーゾク」って聞いたことはないでゲソ? . . . 197
15.6 どうして限定継続が考え出されたのか気にならなイカ? . . . 199
15.7 他の種類の限定継続も紹介しておくでゲソ . . . 204
15.8 おわりに . . . 207
参考文献 . . . 207

第 16 章 愚者のコントロール @tanakh **209**
16.1 I . . . 209
16.2 II . . . 209
16.3 III . . . 210
16.4 IV . . . 211
16.5 V . . . 214
16.6 VI . . . 216
16.7 VII . . . 216
16.8 VIII . . . 217
16.9 IX . . . 217
16.10 X . . . 218
16.11 XI . . . 221
16.12 XII . . . 223
16.13 XIII . . . 223
16.14 XIV . . . 223
16.15 XV . . . 225
16.16 Epilogue . . . 225

第 IV 部 簡約! 入力娘 (4) (2012 年冬) 229

第 17 章 インターフェース @xhl_kogitsune **230**
17.1 Kihime Side . . . 230
17.2 Koyone Side . . . 236

第 18 章 Lighter than Light @master_q **243**
18.1 ダイエットの前に身体測定 . . . 243
18.2 ダイエット指標 . . . 244
18.3 開発環境 “フィットネスジム” . . . 245
18.4 作戦 I. integer-gmp パッケージをダイエット . . . 246
18.5 作戦 B. base パッケージをダイエット . . . 246
18.6 作戦 P. ghc-prim パッケージをダイエット . . . 250
18.7 作戦 R. GHC RTS をダイエット . . . 251
18.8 体が羽のよう! . . . 256

18.9 ダイエットは不健康？ . . . 259
18.10 シェイプ DOWN ガール . . . 260
18.11 参考資料 . . . 260

第 19 章 類は友を呼ぶ？ @darcshaskellmaster 262
19.1 参考文献 . . . 265

第 20 章 OCaml で printf じゃなイカ? @xhl_kogitsune 266
20.1 OCaml で printf しなイカ? . . . 266
20.2 どうやって format6 型になるでゲソ? . . . 267
20.3 フォーマット文字列をつながなイカ? . . . 268
20.4 参考文献 . . . 270

第 21 章 Haskell でも printf じゃなイカ?! @nushio 271
21.1 printf を再現するでゲソ！ . . . 272
21.2 型を追うでゲソ！ . . . 272
21.3 可変個引数関数の正体でゲソ！ . . . 277
21.4 まとめでゲソ！ . . . 278

第 22 章 けいさん！ highschool @tanakh 279

第 V 部 簡約!? 入力娘 Go! (2013 年夏) 299

第 23 章 めたせぴ☆ふぁうんでーしょん @master_q 300
23.1 はじまり . . . 300
23.2 レストランにて . . . 300
23.3 砂の中 . . . 302
23.4 最初のスケッチ: POSIX からの脱出 . . . 304
23.5 冷たい壁 . . . 310
23.6 二度目のスケッチ: 省メモリ化の追求 . . . 311
23.7 あこがれ . . . 320
23.8 三度目のスケッチ: コンテキストを操る . . . 322
23.9 突然の連絡 . . . 335
23.10 これからのこと . . . 336
23.11 参考文献 . . . 336

第 24 章 jhc コピペ @master_q 337
24.1 jhc たんへの愛の歌 . . . 337
24.2 参考文献 . . . 338

第 25 章 侵略者と転校生とアイドルとイカが再帰を学ぶそうですよ！ @nushio 339
25.1 侵略者 . . . 340
25.2 ネオオオサカ炎上 . . . 340
25.3 預言の書 . . . 347
25.4 強敵・災鬼獣 . . . 348

第 26 章 殺物語 @dif_engine 359
26.1 プロローグ . . . 359
26.2 準備：集合と関数 . . . 362
26.3 圏の導入 . . . 368
26.4 関手 . . . 374

26.5 自然変換 . . . 379
26.6 計算の抽象化と修飾された型 . . . 381
26.7 カンザスでないどこか . . . 390
26.8 もう一つの世界 . . . 394
26.9 死闘 . . . 398
26.10 エピローグ . . . 401

第 27 章 入力娘探索 ? @ark_golgo **404**
27.1 プロローグ . . . 404
27.2 囲碁のルールおさらい . . . 405
27.3 そもそもゲームにおける探索とは何か . . . 406
27.4 囲碁で探索が難しい理由 . . . 406
27.5 Df-pn . . . 407
27.6 λ探索 . . . 408
27.7 Df-pn λ探索 . . . 410
27.8 実験結果 . . . 411
27.9 エピローグ . . . 411
27.10 参考文献 . . . 411

第 28 章 ロマンティック・パージング @xhl_kogitsune **412**
28.1 ことのはじまり . . . 412
28.2 構文解析と文脈自由言語 . . . 413
28.3 Bottom-Up 構文解析 . . . 415
28.4 LR 構文解析 . . . 418

第 29 章 HaskEll Shaddai @fumieval **430**
29.1 Lens . . . 433
29.2 Traversal . . . 436
29.3 Iso と Prism . . . 437
29.4 Real World Lens . . . 439
29.5 リンク . . . 440

第 VI 部 簡約!? 入力娘 Rock! (2013 年冬) 441

第 30 章 入力娘探索 2? @ark_golgo **442**
30.1 プロローグ . . . 442
30.2 囲碁のルールおさらい . . . 442
30.3 AND-OR 探索 . . . 444
30.4 Min-Max 探索 . . . 446
30.5 α-β 探索 . . . 448
30.6 α-β 探索が囲碁ではうまく機能しなかった理由 . . . 448
30.7 原始モンテカルロ法 . . . 451
30.8 モンテカルロ木探索 . . . 452
30.9 モンテカルロ木探索を用いた囲碁ソフトの現状 . . . 453
30.10 エピローグ . . . 455
30.11 参考文献 . . . 455

第 31 章 僕のカノジョはスナッチャー @master_q **456**
31.1 川のほとり . . . 456
31.2 カノジョのこと . . . 456

31.3 Snatch-driven development . . . 457
31.4 Android NDK とそのサンプルアプリケーション . . . 458
31.5 スナッチ 1: 何もしない Haskell プログラム . . . 461
31.6 スナッチ 2: 最初のターゲット . . . 465
31.7 スナッチ 3: Haskell コードの近傍をさらにスナッチ . . . 471
31.8 こっそりと驚きを届けたい . . . 475
31.9 スナッチ 4: OpenGL ES をたたく . . . 475
31.10 スナッチ完了 . . . 480
31.11 プレゼントを作ろう . . . 481
31.12 お互いを理解することは小さく傷付け合うこと . . . 481
31.13 それでも明日はやって来る . . . 481
31.14 ソースコードライセンス . . . 482
31.15 参考文献 . . . 482

第0章

関数型イカ娘とは! / この総集編について

— @xhl_kogitsune

0.1 この総集編について

この本の構成について

この本は関数型とイカ娘のファンブックでゲソ。各著者が好きなことを書いた感じなので各章は独立して読めるでゲソ。以前の「λ カ娘」本がないと分からないこともないでゲソ。

本文収録内容

1 巻:「簡約！ λカ娘」 / 2 巻:「簡約！？ λカ娘 (二期)」/ 3 巻:「簡約！？ λカ娘 (算)」/ 4 巻:「簡約！ λカ娘 (4)」/ 5 巻:「簡約！？ λカ娘 Go!」/ 6 巻:「簡約！？ λカ娘 Rock!」

巻頭カラー口絵収録内容

- 総集編表紙別バージョン (七臣ナオミ 様)
- 1 巻「簡約！ λカ娘」表紙 (尾住 様)
- 2 巻「簡約！？ λカ娘 (二期)」表紙 (@dif_engine)
- 3 巻:「簡約！？ λカ娘 (算)」表紙 (@dif_engine)
- 4 巻:「簡約！ λカ娘 (4)」表紙 (七臣ナオミ 様)
- 5 巻:「簡約！？ λカ娘 Go!」裏表紙 (@dif_engine)
- 5 巻:「簡約！？ λカ娘 Go!」表紙 (@dif_engine)
- 6 巻:「簡約！？ λカ娘 Rock!」表紙 (@dif_engine)

0.2 関数型イカ娘とは

Q. 関数型イカ娘って何ですか?
A. いい質問ですね!
Q. 関数型イカ娘について教えてください
B. その願いは君にとって魂を差し出すに足るものかい？

関数型イカ娘とは、「イカ娘ちゃんは 2 本の手と 10 本の触手で人間どもの 6 倍の速度でコーディングが可能な超絶関数型プログラマー。型ありから型なしまでこよなく愛するが特に Scheme がお気に入り。」という妄想設定でゲソ。

2010 年 10 月のイカ娘アニメ第 5 話の次回予告で、イカ娘がパソコンを使っているシーン[*1]が映ったでゲソが、そこから

[*1] http://www.ika-musume.com/season_1/images/middle/story/vol05/img_02.jpg

スーパープログラマーイカ娘!!! お、おぢょうちゃん、Scheme とか、どうかなっ!?
http://twitter.com/xhl_kogitsune/status/28710865318, 2010/10/26 02:28

次回予告でイカ娘がパソコンに向かっているシーンから、イカ娘は超絶 Scheme プログラマーという電波まで受信した←今ここ。そんな彼女に私は C++ とかを布教すべきだろうか? でも彼女は Scheme で幸せそうだし... うーん。
http://twitter.com/xhl_kogitsune/status/28711063147, 2010/10/26 02:31

という電波を@xhl_kogitsune が受信したのが始まりでゲソ。

その後も、アニメを見ながら twitter 上で関数型イカ娘ネタが増えて行ったでゲソ。詳しくはイカの Togetter を見るといいでゲソ。

Togetter: 関数型イカ娘　http://togetter.com/li/63957
Twitter 上での関連発言まとめでゲソ。「見たけど分からなかったでゲソ…」という声も聞こえてくるでゲソ。

2010 年 12 月には Functional Ikamusume Advent Calendar jp 2010 というイベントがオンラインで開催されたでゲソ。

Functional Ikamusume Advent Calendar jp 2010　http://bit.ly/fSoVhy
持ち回りで 1 日 1 題関数型イカ娘ネタを書こうという会でゲソ。12 人も参加者が集まったでゲソ。

そして、ついに関数型イカ娘本を出す計画が (なぜか) 実現したのでゲソ! ←今ここ
さぁ、お前たち、この調子で手続き型を侵略するでゲソよ!

0.3　関数型イカ娘アニメ紹介

関数型イカ娘の活躍はアニメにも出てきたでゲソ。

学校に行かなイカ？ (第 5 話)　わたしが学校のパソコンを使って Scheme でゲームを作っている記念すべき回でゲソ。わたしはゲームで遊んでいたんじゃなくてデバッグをしていたでゲソ!

勉強しなイカ？ (第 6 話)　「数学とλ計算に突出した才能を示すイカ娘。しかし、海の家れもんの住人たちには相手にされず、イカ娘本人もまだ自分の真の力を理解していなかった……。」わたしには実は数学の才能もあったことが明らかになる回でゲソ。原作で数学が得意なら関数型が得意でもいいじゃなイカ!

研究しなイカ？ (第 7 話)　わたしが数学科や情報科学科を侵略して研究活動をする回でゲソ。そして、わたしは人類侵略のためについに λ プロセッサの開発に乗り出すのだった……でゲソ!

新能力じゃなイカ？ (第 8 話)　新能力に目覚めたでゲソ。手続き型の能力かと憶測を呼んだでゲソが、そんなことはなかったでゲソ。

ピンポンダッシュしなイカ？ (第 9 話)　ネットワークを侵略しなイカ! まずは ping からでゲソ!

make しなイカ？ (第 9 話)　わたしと一緒に OCamlMakefile でゲソ! 海の家れもんにやってきた客の make に興味津々のイカ娘。イカ娘「その make はどうするのでゲソ?」客「./configure; make; make install」と make のし方を教えてもらうイカ娘。make しないのかと尋ねるイカ娘に対して、渚は ant 派だから make はしないのだと答える。調子に乗って make していたイカ娘だが、最後は make clean できなくなって栄子に泣きつくのであった。

第I部

簡約! 入力娘 (2011年夏)

評価戦略

第1章

β簡約！λ力娘

— @xhl_kogitsune

お前たち! わたしと一緒にλしなイカ!

今日は、λ計算とSKI計算という、わたしが好きな二つの関数型計算モデルを紹介するでゲソ! どちらも、構文が3行で説明できるすごく単純な作りでゲソが、チューリング完全でゲソ。つまり、大雑把に言ってコンピュータで計算できることは全て計算できる素晴らしい能力を持っているでゲソ! 手続き型を侵略するでゲソ!

1.1 λしなイカ？

まずは楽しい楽しいλ計算とβ簡約について教えてやるでゲソ! λは怖くないでゲソよ～

1.1.1 λ式を書かなイカ?

お前たちも生き物であるからには、関数を書きたいという欲求に勝てないはずでゲソ。λ式はそんなお前たちにぴったりの関数の表現方法でゲソ!

え、なぜ関数はC言語とかでも書けるじゃなイカって? これだから人間は困るでゲソ。λ式で書く方が断然エレガントでゲソ。関数というからには式で書けるのが本当じゃなイカ。数式にはgotoなんて論外でゲソ。

最初に下準備として**変数**を考えるでゲソ。変数xはそのままλ式として成立するでゲソ。

$$x$$

最小のλ式じゃなイカ。ちなみに変数の具体的な名前はどうでもいいでゲソ。普通は小文字一文字を使うことが多いでゲソ。

さて、いよいよ**関数**を書こうじゃなイカ。xを受け取って$x+1$を返す関数を考えるでゲソ。普通の数式だとこんな感じでゲソ。

$$f(x) = x+1$$

これは、λ式ではこんな感じに書くでゲソ。

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

λは「らむだ」と読むでゲソ。入(はいる)でも人(ひと)でもイ(い)でもん(ん)でもないでゲソ。私の触手のように曲線が美しい文字じゃなイカ。λの直後に書いてあるxが引数で、その後の部分が関数が返す値を表す式でゲソ。これがλ式での関数の書き方でゲソ。一般的には、xを変数名、Aをλ式とすると

$$(\lambda x.A)$$

はxを受け取ってAを返す関数をあらわすλ式でゲソ。正確にはλ**抽象**と呼ぶでゲソ。私も覚えたての頃は一晩中砂浜にλ式で関数を書きまくったものでゲソ。

$x+1$というのは厳密なλ式ではないでゲソが、読みやすいし、私の好きなSchemeとかを使うとこういう式も書けるから、説明のためにこう書くことにするでゲソ。全部純粋なλ式で書く方法は後で説明するでゲソ。

関数が書けたら**関数適用**したくなるじゃなイカ。当然、λ式で関数適用も書けるでゲソ。さっき書いた関数を3に適用して$f(3)$を計算するにはイカのように書くでゲソ。

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

A,Bをλ式とすると

$$(AB)$$

と書くと、関数Aを引数Bに適用する関数適用$A(B)$を表すλ式になるでゲソ。これで関数適用し放題じゃなイカ!

えっ、これだと引数を1つしか取れないじゃなイカ、って? 心配ないでゲソ! 引数なんて1つあればなんだって出来るでゲソ! たとえば、$g(x,y) = x+y$みたいな2引数関数は、

$$(\lambda x.(\lambda y.x+y))$$

みたいにλ抽象2回で書けて、

$$((gx)y)$$

みたいに 1 引数の関数適用を 2 回繰り返せば適用できるのでゲソ。ここで、g は「x を受け取って、「y を受け取って、$g(x,y)$ を返す関数」を返す関数」でゲソ。

λ 式というか一般の関数型言語では、関数を値として受け取ったり返したりできるのでこういう芸当が出来るでゲソ! 関数を値として受け取ったり返したりする関数のことを**高階関数**と呼ぶでゲソ。高階しなイカ!

カッコを書くのが面倒になったら、適宜省略できるでゲソ。たとえば

$$(\lambda x.(\lambda y.(\lambda z.x+y+z)))$$

は

$$(\lambda x.\lambda y.\lambda z.x+y+z)$$

のように、

$$(((gx)y)z)$$

は

$$(gxyz)$$

のようにカッコを省略できるでゲソ。$g(fx)$ のような場合はカッコは省略できないでゲソ (省略すると $(gf)x$ という意味になってしまうでゲソ)。

λ 式

変数 x
λ 抽象 $(\lambda x.A)$
関数適用 (AB)

の三種類でゲソ。

1.1.2 β 簡約しなイカ?

λ 式は数式とかプログラムのようなものでゲソ。プログラムを書いたら次は実行しようじゃなイカ! さっきは $f(3)$ を $((\lambda x.(x+1))3)$ と書いたでゲソが、これだけでは 4 という値は出てこないでゲソ。β 簡約という、実行に当たる操作をしないと 4 は出てこないでゲソ!

β 簡約は、式の純粋なシンタックスだけで定義できる操作でゲソ。形式美をとんと楽しむでゲソ!

β 簡約は、

$$((\lambda x.A)B)$$

という形をした λ 式を、$A[B/x]$ に置き換える操作でゲソ。$A[B/x]$ というのは、「A の中にある x を B で置き換えたもの」に変換する操作でゲソ。さっきの例だと、$(x+1)$ の中の x を 3 に置き換えて、

$$((\lambda x.(x+1))3) \xrightarrow{\beta} (3+1)$$

と、$(3+1)$ という式に β 簡約されたでゲソ。β 簡約一回分を $\xrightarrow{\beta}$ という矢印で表現したでゲソ。β 簡約は楽しいでゲソ。λ 式を書くのと同じくらい楽しいでゲソ。

最後に $3+1$ を計算すると 4 になるでゲソが、次に書くように、$3+1$ から 4 を計算する過程もチャーチ数を使えば純粋な λ 計算と β 簡約で表現できるでゲソ。

β 簡約

$$((\lambda x.A)B) \xrightarrow{\beta} A[B/x]$$

1.1.3 α 変換しなイカ?

同じ変数名が複数回使われている場合は注意が必要でゲソ。たとえば、「x を受け取って x を返す関数 i(恒等写像)」は

$$(\lambda x.x)$$

と書けるでゲソ。「x を受け取って i を返す関数」は

$$(\lambda x.i)$$

と書けるでゲソ。これを合わせると、「x を受け取って恒等写像を返す関数」は

$$(\lambda x.(\lambda x.x))$$

となるでゲソ。これを 1 に適用して β 簡約するときに、何も考えずに全部の x を 1 に置換すると、

$$(\lambda x.(\lambda x.x))\ 1 \xrightarrow{\beta} (\lambda 1.1)$$

そもそも文法的におかしいじゃなイカ! λ 抽象の変数の部分は置換しない、ってしてもやっぱりダメでゲソ。

$$(\lambda x.(\lambda x.x))\ 1 \xrightarrow{\beta} (\lambda x.1)$$

恒等写像じゃないでゲソ! ここで問題なのは、外側の λ 抽象の x と、内側の λ 抽象の x の二種類の x があるのに一緒に扱ってしまっていることでゲソ。

この解決のためには、各 λ 抽象で使われる引数の変数名を変えればいいでゲソ。これを **α 変換**と呼ぶでゲソ。外側の λ 抽象の引数 x(1 番目の x) を x_1 に、内側の λ 抽象の引数 x(2 番目の x) を x_2 に変更するでゲソ。3 番目の x は x_1 か x_2 かでゲソが、候補のうち内側の方の λ 抽象を優先させて x_2 にするでゲソ。

$$(\lambda x_1.(\lambda x_2.x_2))\ 1 \xrightarrow{\beta} (\lambda x_2.x_2)$$

正しいじゃなイカ!

こんな、変数名がかぶるようなコード書かないんじゃなイカ、と思うかもしれないでゲソが、複数の

関数で同じ変数名を使って、それらを混ぜて β 簡約するとすぐかぶるようになるでゲソ。

α 変換

それぞれの λ 抽象の引数の変数名が被っている場合は、α 変換して違う変数名を割り振る必要があるでゲソ!

1.1.4 チャーチしなイカ?

λ 式の世界には λ と (と) と変数くらいしかないでゲソ。0 も、1 も、足し算 + もないでゲソ。そう、例えるのなら、ここは原子の世界、普段見ている計算は全て λ 式から構成されるのでゲソ。お前たちの体も実は λ で出来ているのかもしれないでゲソよ! λ を恐れよ! なのでゲソ。

まずは自然数[*1]を作るでゲソ。直接 0 は作れないでゲソが、「0 のようなもの」は作れるでゲソ。たとえばこんな感じでゲソ。

0 $(\lambda f.\lambda x.x)$
1 $(\lambda f.\lambda x.(fx))$
2 $(\lambda f.\lambda x.(f(fx)))$
3 $(\lambda f.\lambda x.(f(f(fx))))$

これを**チャーチ数**と呼ぶでゲソ。自然数 i を、「f と x を受け取って、f を x に i 回適用する関数」として表現するでゲソ。砂浜に棒を並べて数を数えるのと一緒でゲソ!

ん、変数名を使って

0 f_0
1 f_1
2 f_2
3 f_3

とかすればいいんじゃなイカだって? それだと変数名を変えたら一発でアウトじゃなイカ! それに、たとえば f_1 と f_2 から λ 式だけで f_3 を作り出すことが出来ないので使えないでゲソー。

その点、チャーチ数なら全てを λ 式でできるでゲソ! まずは足し算しなイカ! 正確には、足し算関数を書かなイカ! 足し算ということは、つまり「f と x を受け取って、f を x に i_1 回適用する関数 g_1」と「f と x を受け取って、f を x に i_2 回適用する関数 g_2」を受け取って「f と x を受け取って、f を x に i_1+i_2 回適用する関数」を返す関数 add を作ればいいでゲソ! 関数と関数を受け取って関数を返す関数とか、初めは頭がこんがらがるかもしれないでゲソが、高階関数の世界に慣れれば普通にできるようになるでゲソ!

順番にいくでゲソ。大枠は「g_1 と g_2 を受け取って何かを返す関数」なのでこうなるでゲソ。

$$\mathtt{add} = (\lambda g_1.\lambda g_2.\text{何か})$$

何かではないでゲソ☆「何か」は、「f と x を受け取って、何か 2 を返す関数」なので、

$$\mathtt{add} = (\lambda g_1.\lambda g_2.(\lambda f.\lambda x.\text{何か } 2))$$

こうなるでゲソ。「何か 2」は「f を x に i_1+i_2 回適用したもの」なので、g_1 と g_2 を使ってこう書けるでゲソ。

$$\mathtt{add} = (\lambda g_1.\lambda g_2.(\lambda f.\lambda x.(g_2 f(g_1 fx))))$$

できたじゃなイカ!

これで $3+1$ を計算してみるでゲソ!

$$\begin{aligned}
&(\mathtt{add}\ 3\ 1)\\
=\ &(\lambda g_1.\lambda g_2.(\lambda f.\lambda x.(g_2 f(g_1 fx))))\\
&\quad(\lambda f.\lambda x.(f(f(fx))))\\
&\quad(\lambda f.\lambda x.(fx))\\
\xrightarrow{\beta*}\ &(\lambda f.\lambda x.((\lambda f.\lambda x.(fx))f\\
&\quad((\lambda f.\lambda x.(f(f(fx))))fx)))\\
\xrightarrow{\beta}\ &(\lambda f.\lambda x.((\lambda f.\lambda x.(fx))f(f(f(fx)))))\\
\xrightarrow{\beta}\ &(\lambda f.\lambda x.(f(f(f(fx)))))\\
=\ &4
\end{aligned}$$

できたじゃなイカ!

チャーチ数

λ 式だけで表現できて演算もできる、自然数の表現方法でゲソ。

1.2 SKI しなイカ?

1.2.1 SKI じゃなイカ?

世界は *SKI* で全て表現できるでゲソ。正確に言えば、*SKI* **計算** *(SKI combinator calculus)* はチューリング完全でゲソ! SKI 計算というのは S,K,I という三種類のコンビネータ[*2]だけで計算を表現するものでゲソ。

$$\begin{aligned}
\mathtt{S}xyz &\to ((xz)(yz)) = xz(yz)\\
\mathtt{K}xy &\to x\\
\mathtt{I}x &\to x
\end{aligned}$$

ここで x,y,z は SKI コンビネータの列を表す変数で、SKI 計算本体には λ 計算の時にはあった変数は全く出てこないでゲソ。全ては S,K,I,(,) の列でゲソ。λ

[*1] プログラマは 0 から数えるでゲソ

[*2] エビみたいなものと思っておけばいいでゲソ

式と β 簡約の時みたいに、SKI 式で関数を書いて簡約を繰り返して計算をするでゲソ。カッコは λ 式の関数適用の時と同じように省略できるでゲソ。

賢明な諸君はもう気づいているんじゃなイカ。そう、I は S と K で表現できるから実は要らないでゲソ! 具体的には $\mathtt{I}x$ は $\mathtt{SKK}x$ に書き換えてもいいでゲソ。いいでゲソが、I を SKK と書くと私が SKK カ娘になってしまうので I はあえて残してあるでゲソ。ここでお前たちの練習のためにこれを実際に確かめてみようじゃなイカ!

$$\begin{aligned} \mathtt{I}z &\to z \\ \mathtt{SKK}z \to \mathtt{K}z(\mathtt{K}z) &\to z \end{aligned}$$

正しいじゃなイカ!

SKI は私の好きな λ 式でも表現できるでゲソ!

$$\begin{aligned} \mathtt{S} &= (\lambda x.(\lambda y.(\lambda z.xz(yz)))) \\ \mathtt{K} &= (\lambda x.(\lambda y.x)) \\ \mathtt{I} &= (\lambda x.x) \end{aligned}$$

λ 式版ででも $\mathtt{SKK} \xrightarrow{\beta*} \mathtt{I}$ を確かめてみるでゲソ!

$$\begin{aligned} \mathtt{SKK} &= (\lambda x.(\lambda y.(\lambda z.xz(yz))))\mathtt{KK} \\ &\xrightarrow{\beta} (\lambda y.(\lambda z.\mathtt{K}z(yz)))\mathtt{K} \\ &\xrightarrow{\beta} (\lambda z.\mathtt{K}z(\mathtt{K}z)) \\ &\xrightarrow{\beta} (\lambda z.z) \\ &= \mathtt{I} \end{aligned}$$

やっぱり正しいじゃなイカ!

SKI 計算

$$\begin{aligned} \mathtt{S}xyz \to ((xz)(yz)) &= xz(yz) \\ \mathtt{K}xy &\to x \\ \mathtt{I}x &\to x \end{aligned}$$

変数一つない、美しいフォルムじゃなイカ!

1.2.2 IKMSM じゃなイカ?

次に、私、イカ娘を示す SKI 式を作って見ようじゃなイカ。M はイカの式を表す文字でゲソ[*3]。

$$\mathtt{M} = (\lambda x.(xx))$$

または

$$\mathtt{M}x \to xx$$

これを使うと、私を示す SKI 式、IKMSM が出来るでゲソ! IKMSM は簡約するとイカのようになるでゲソ。

$$\begin{aligned} \mathtt{IKMSM} &\to \mathtt{KMSM} \\ &\to \mathtt{MM} \end{aligned}$$

MM は λ 式にすると $(\lambda x.xx)(\lambda x.xx)$ になるでゲソ。これは Ω とも書くことがあるでゲソ。Ω はタコじゃなイカと言う奴もいるでゲソが、Ω は私の触手の一形態を表す文字でゲソ。断じてタコなんかじゃないでゲソ!

ここでお前たちは私の恐ろしさを知ることになるでゲソ。イカの式を見るがいいでゲソ!

$$\begin{aligned} &\mathtt{IKMSM} \\ \to &\mathtt{MM} \\ \to &\mathtt{MM} \\ &\cdots \end{aligned}$$

つまり、私は永遠に変わらないまま簡約を続ける存在、つまり神でゲソ!

このことは別の方法でも証明できるでゲソ。

$$(\lambda m.\mathtt{IK}m\mathtt{S}m)$$

という、これまた私を示す λ 式を考えてみるでゲソ。これは β 簡約すると

$$\begin{aligned} (\lambda m.\mathtt{IK}m\mathtt{S}m) &\xrightarrow{\beta} (\lambda m.\mathtt{K}m\mathtt{S}m) \\ &\xrightarrow{\beta*} (\lambda m.mm) \end{aligned}$$

となるでゲソ。つまり、これを自分自身に適用した λ 式、$(\lambda m.\mathtt{IK}m\mathtt{S}m)(\lambda m.\mathtt{IK}m\mathtt{S}m)$ を β 簡約すると、

$$\begin{aligned} &(\lambda m.\mathtt{IK}m\mathtt{S}m)(\lambda m.\mathtt{IK}m\mathtt{S}m) \\ \xrightarrow{\beta*} &(\lambda m.mm)(\lambda m.mm) \\ \xrightarrow{\beta} &(\lambda m.mm)(\lambda m.mm) \\ &\cdots \end{aligned}$$

やっぱり Ω じゃなイカ!

1.2.3 SKI を書こうじゃなイカ!

これでお前たちは SKI を読めるようになったはずでゲソ。次は SKI を書こうじゃなイカ!

SKI コンビネータはチューリング完全なので、どんな λ 式でも SKI で書けるでゲソ。SKI コンビネータは λ 式にはあった変数名もないし、一見訳の分からない形をしているので、これでどうやって書けばいいのか戸惑うかもしれないでゲソ。でも心配は要らないでゲソ。半日も SKI で遊んでいればすらすら読み書き出来るようになれるでゲソ![*4]

[*3] Combinator Birds http://www.angelfire.com/tx4/cus/combinator/birds.html

[*4] イカ娘基準。でも λ 計算の下地があればイケるんじゃなイカ?

- S は関数適用のようなものでゲソ。関数適用できるのは S だけでゲソ。
- 引数を無視できるのは K だけでゲソ。これを使って関数適用や引数を調整するでゲソ。

まずは、関数合成をしてみようじゃなイカ! 1 引数関数 f,g を受け取って、その合成 (「x を受け取って $f(g(x))$ を返す関数」) を返す関数 c を作ろうじゃなイカ。

$$cfgx \to f(gx)$$

まずは、感覚を掴むために感覚と試行錯誤でやってみるでゲソ。

c は関数適用を含むから、S を含むんじゃなイカ? たとえばこんなカタチをしているんじゃなイカ?

$$c = \mathtt{S}$$

でもこれだと

$$cfgx = \mathtt{S}fgx \to (fx)(gx)$$

となってしまうので違うでゲソ。

次に

$$c = \mathtt{S}a$$

というカタチを考えてみるでゲソ (a は何かでゲソ。何であるかはこれから考えるでゲソ)。そうすると

$$cfgx = \mathtt{S}afgx \to ag(fg)x$$

ムムム、(fg) とか出てきてしまったでゲソ。これ以外に f はないでゲソ。(fg) から f を分離する手段はないので、詰みでゲソ。

じゃあ

$$c = \mathtt{S}ab$$

なんじゃなイカ?

$$cfgx = \mathtt{S}abfgx \to af(bf)gx$$

明らかな間違いはこの段階では見つからないでゲソ。これが $f(gx)$ になるようにしたいでゲソ。もう一回関数適用が必要な気がするので

$$af \to \mathtt{S}$$

のケースを考えるでゲソ。すると、

$$cfgx \to af(bf)gx \to \mathtt{S}(bf)gx \to bfx(gx)$$

やったでゲソ! これで、

$$af \to \quad \mathtt{S} \text{ つまり } \quad a = \quad \mathtt{KS}$$
$$bfx \to \quad f \text{ つまり } \quad b = \quad \mathtt{K}$$

であれば OK ということが分かったでゲソ。まとめると、

$$\mathtt{S(KS)K}fgx \to f(gx)$$

関数合成できたじゃなイカ! こんな風に、関数適用のための S の使い方を考えつつ、S と K で適当に辻褄を合わせれば結構できるでゲソ。

1.2.4 SKI に変換しなイカ?

さっきの SKI の書き方はある意味職人芸でゲソ。わたしは SKI を並べながら考えるのが大好きなのでついつい手で書いてしまうでゲソが、λ 式を SKI に規則的に変換する方法もあるでゲソ。[*5]

$$\begin{aligned}
T\{\lambda x.M\} &= A\{x, T\{M\}\} \\
T\{MN\} &= T\{M\}T\{N\} \\
T\{x\} &= x \\
A\{x,x\} &= \mathtt{I} \\
A\{x,y\} &= \mathtt{K}y \quad (\text{但し } y \text{ と } x \text{ は異なる}) \\
A\{x,PQ\} &= \mathtt{S}(A\{x,P\})(A\{x,Q\})
\end{aligned}$$

x,y は変数、M,N は λ 式、P,Q は変数交じりの SKI 式でゲソ。$T\{e\}$ が λ 式 e を SKI 式に変換する関数で、$A\{x,M\}$ は「変数 x を受け取って SKI 式 M を返す関数」の SKI 式を返す関数でゲソ。T が λ 式を構文に従って分解して A に投げて、A が実際に SKI を吐くでゲソ。

- 「x を受け取って x を返す」のはまさに I でゲソ。
- 「x を受け取って y を返す」のは $\mathtt{K}yx \to y$ なので $\mathtt{K}y$ でゲソ。
- 最後の場合も

$$\begin{aligned}
&\mathtt{S}\{A(x,P)\}\{A(x,Q)\}x \\
&\to (A\{x,P\}x)(A\{x,Q\}x) \\
&\to PQ
\end{aligned}$$

となるので大丈夫でゲソ。

途中で x とかの変数が混じることがあるでゲソが、最後には消えるでゲソ。

それでは早速、関数合成をする λ 式 $\lambda f.\lambda g.\lambda x.f(gx)$ を SKI に変換してみると、図 1.1 のようになるでゲソ!

結果は S(S(KS)(S(S(KS)(S(KK)(KS)))
(S(S(KS)(S(KK)(KK)))
(S(KK)I))))(S(S(KS)(S(S(KS)(S(KK)(KS)))
(S(S(KS)(S(KK)(KK)))(KI))))(S(KK)(KI))) …長いでゲソ! 一応これを c と置くと、$cfgx$ は $f(gx)$ になるでゲソ!

[*5] http://www.icfpcontest.org/ から引用したでゲソ

$$
\begin{aligned}
&T\{\lambda f.\lambda g.\lambda x.f(gx)\}\\
=&A\{f,A\{g,A\{x,T\{f(gx)\}\}\}\}\\
=&A\{f,A\{g,A\{x,f(gx)\}\}\}\\
=&A\{f,A\{g,\mathtt{S}(A\{x,f\})(A\{x,gx\})\}\}\\
=&A\{f,A\{g,\mathtt{S}(A\{x,f\})(\mathtt{S}(A\{x,g\})(A\{x,x\}))\}\}\\
=&A\{f,A\{g,\mathtt{S}(\mathtt{K}f)(\mathtt{S}(\mathtt{K}g)\mathtt{I})\}\}\\
=&A\{f,\mathtt{S}\big(\mathtt{S}(A\{g,\mathtt{S}\})(A\{g,\mathtt{K}f\})\big)\big(\mathtt{S}(\mathtt{S}(A\{g,\mathtt{S}\})(A\{g,\mathtt{K}g\}))(A\{g,\mathtt{I}\})\big)\}\\
=&A\{f,\mathtt{S}\big(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{K}f))\big)\big(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})\mathtt{I}))(\mathtt{KI})\big)\}\\
=&\mathtt{S}\big(\mathtt{S}(A\{f,\mathtt{S}\})(A\{f,\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{K}f))\})\big)\big(A\{f,\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})\mathtt{I}))(\mathtt{KI})\}\big)\\
=&\mathtt{S}\big(\mathtt{S}(\mathtt{KS})\big(\mathtt{S}(\mathtt{S}(A\{f,\mathtt{S}\})(A\{f,\mathtt{KS}\}))(A\{f,\mathtt{S}(\mathtt{KK})(\mathtt{K}f)\})\big)\big)\\
&\big(\mathtt{S}\big(\mathtt{S}(A\{f,\mathtt{S}\})(A\{f,\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})\mathtt{I})\})\big)\big(A\{f,\mathtt{KI}\}\big)\big)\\
=&\mathtt{S}\big(\mathtt{S}(\mathtt{KS})\big(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{KS})))(\mathtt{S}(\mathtt{S}(\mathtt{KS})(A\{f,\mathtt{KK}\}))(\mathtt{S}(\mathtt{KK})\mathtt{I}))\big)\big)\\
&\big(\mathtt{S}\big(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{S}(A\{f,\mathtt{S}\})(A\{f,\mathtt{KS}\}))(A\{f,\mathtt{S}(\mathtt{KK})\mathtt{I}\}))\big)\big(\mathtt{S}(\mathtt{KK})(\mathtt{KI})\big)\big)\\
=&\mathtt{S}\big(\mathtt{S}(\mathtt{KS})\big(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{KS})))(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{KK})))(\mathtt{S}(\mathtt{KK})\mathtt{I}))\big)\big)\\
&\big(\mathtt{S}\big(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{KS})))(\mathtt{S}(\mathtt{S}(\mathtt{KS})(\mathtt{S}(\mathtt{KK})(\mathtt{KK})))(\mathtt{KI})))\big)\big(\mathtt{S}(\mathtt{KK})(\mathtt{KI})\big)\big)
\end{aligned}
$$

図 1.1: $\lambda f.\lambda g.\lambda x.f(gx)$ を SKI に変換するでゲソ!

1.2.5 SKI でチャーチ数しなイカ?

当然、SKI でもチャーチ数を作れて計算できるでゲソ! 小さいチャーチ数をいくつか作ってみるでゲソ!

まずは 0 でゲソ!

$$0fx \to x$$

なんか $0=\mathtt{K}$ っぽいでゲソ! しかし、残念ながら違うでゲソ!

$$\mathtt{K}fx \to f$$

じゃぁ次は $0=\mathtt{S}a$ みたいな形を考えてみるでゲソ!

$$\mathtt{S}afx \to ax(fx)$$

$a=\mathtt{K}$ なら x になるじゃなイカ!

$$0=\mathtt{SK}$$

実は $0=\mathtt{K}a$ という形でもいけるでゲソ!

$$\mathtt{K}afx \to ax$$

これが x になればいいので、$a=I$ じゃなイカ!

$$0=\mathtt{KI}$$

次に 1 でゲソ!

$$1fx \to fx$$

これはそのまま I でいいんじゃなイカ!

$$1=\mathtt{I}$$

次に 2 でゲソ!

$$2fx \to f(fx)$$

$2=\mathtt{S}a$ みたいな形になっているんじゃなイカ?

$$2fx=\mathtt{S}afx \to ax(fx)$$

f が (fx) という形だけになってしまったので 2 回使えないじゃなイカ! じゃあ $2=\mathtt{S}ab$ じゃなイカ?

$$2fx=\mathtt{S}abfx \to af(bf)x$$

よさそうなので a の中身について考えるでゲソ。もう一回 S が必要そうでゲソ。

$$a=\mathtt{S} \to af(bf)x \to fx(bfx)$$

ダメじゃなイカ!

$$a=\mathtt{S}c \to af(bf)x \to c(bf)(f(bf))x$$

これもダメじゃなイカ!

$$a=\mathtt{S}cd \to af(bf)x \to cf(df)(bf)x$$

$c=\mathtt{KS}, d=\mathtt{K}, b=\mathtt{I}$ ならいいんじゃなイカ? つまり

$$2=\mathtt{S}(\mathtt{S}(\mathtt{KS})\mathtt{K})\mathtt{I}$$

できたじゃなイカ!

1.2.6 SKI でチャーチ数を自動生成しなイカ?

いちいち手で書いていたら日が暮れてしまうでゲソから、プログラムでチャーチ数を生成するでゲソ。制限なしの SK で最短のものとか考えると難しいでゲソから、ここではより小さいチャーチ数から大きいチャーチ数を組み立てることを考えるでゲソ! イカのような SKI 式でチャーチ数どうしの演算ができて大きいチャーチ数を作れるでゲソ! n,m は自然数、n, m は SKI 式で書かれたチャーチ数でゲソ。実際に正しいかどうかは読者の演習問題とするでゲソ!

$n+1$ `S(S(KS)K)n`
$n+m$ `n(S(S(KS)K))m`
nm `S(Km)n`
$n(n+1)$ `M(S(S(KS)K))n`
n^m `mn`
n^n `Mn`

256 イカの数とかなら、これらの組み合わせでできるチャーチ数を全列挙して一番短いものを選ぶくらい余裕でゲソ。

SKI 式まとめでゲソ!

関数合成	`S(KS)K`$fgx \to f(gx)$
イカ娘	`IKMSM` $\to$ `MM`
0	`SK` または `KI`
1	`I`
2	`S(S(KS)K)I`
$n+1$	`S(S(KS)K)n`
$n+m$	`n(S(S(KS)K))m`
nm	`S(Km)n`
$n(n+1)$	`M(S(S(KS)K))n`
n^m	`mn`
n^n	`Mn`

1.3 有効活用しようじゃなイカ!

1.3.1 実用しなイカ?

そろそろ λ や SKI で実際にプログラムを書きたくなってきたはずでゲソ! そんなお前たちのためにオススメの言語を紹介するでゲソ! λ 式や SKI 計算の仕組みを取り入れつつ、便利で効率よく計算ができるようにしたプログラミング言語は色々あってあちこちで使われているでゲソ。

LISP: λ 計算を元にしたプログラミング言語でゲソ。今日はチャーチ数を使った純粋な λ 計算を紹介したでゲソが、LISP ではそれ以外にも普通に + とかの演算子や普通の整数や小数などを使った普通のプログラミングができるでゲソ。もちろん、純粋 λ 計算してもいいんじゃなイカ?

() ばっかと言われるでゲソが、それがいいんじゃなイカ! LISP には亜種がたくさんあって、その中では Scheme がわたしのお気に入りでゲソ♪

Unlambda, Lazy K: SKI を元にしたプログラミング言語でゲソ。Unlambda では文字を表示する関数があるのでそれを使って Hello World できるでゲソ。

Lambda: the Gathering: これはプログラミング言語というより、ゲームでゲソ。「S のカード」「K のカード」「I のカード」をはじめとして、攻撃用カード、数を計算するためのカードなどがあって、それを組み合わせて SKI 式のような関数を組み立てて評価することで副作用で相手を倒すでゲソ!

ICFP Programming Contest 2011 (ICFPC 2011) `http://www.icfpcontest.org/` という、関数型言語の学会 (ICFP: The International Conference on Functional Programming) 併設のプログラミングコンテストの 2011 年のお題でしたでゲソが、普通に面白いでゲソ! チャーチ数や SKI 計算の力が試されるでゲソ!

`@nushio` さんによる非公式対戦サーバー (Yet Another Unofficial Judge for ICFP Contest 2011 `http://www.paraiso-lang.org/Walpurgisnacht/`) もあるので、対戦成績を競えるでゲソ!

1.3.2 λ 計算は何がいいでゲソ?

お前たちは λ 式や SKI 計算が何の役に立つのかと思うかもしれないでゲソ。そうだとしたら、もしかして副作用と Syntax Sugar に毒されているのではなイカ?

まず第一に、覚えることが少なくて済むから書くのが簡単じゃなイカ[*6]。シンプル・イズ・ベスト、でゲソ!

構成要素が少ないことは理論的にも便利でゲソ。何かを証明したりしたい時、SKI なら S と K の二種類についてだけ証明すればコンピュータ上の計算について全部証明できるでゲソ。

他にも、データとプログラムを統一的に扱うことができるでゲソ。λ 式は β 簡約によって実行されるプログラムでもあるし、β 簡約によって操作されるデータでもあるでゲソ。

構造が単純な分、計算も速いと嬉しいでゲソ。λ 計算専用ハードウェアとか胸が熱くならなイカ! さすがに純粋 λ 計算や SKI 計算にしてしまうと、チャーチ数を使うことになって現実的でないでゲソが…。

そして、λ 計算や SKI 計算は、関数型プログラミング、型理論、プログラミング言語理論、計算可能性理論等等の豊かで深い世界の一部でゲソ。さらに関数型の世界を極めて手続き型を侵略しなイカ!

[*6] あくまでイカ娘個人の感想であり、効能を保証するものではありません

第2章

数をつくらなイカ？

— @nushio

読者は、この本のそこかしこに SKI が飾り付けられているのに気がついたんじゃなイカ？ これらの SKI 式はメッセージをエンコードしているので、できるものなら解読してみるがいいでゲソ。え？ 手書きな訳がないじゃなイカ！ 私は１０本の触手があるとはいえとことん lazy(褒め言葉) なプログラマなのでゲソ。なるべく簡潔に書ける言語を選んで文字数最短の SKI 式を探索させたに決まってるじゃなイカ！

こうなったら、この SKI 文字列を生成した楽しいコードを解説することで、読者の脳を関数型に侵略してやるでゲソ?! lazy(褒め言葉) な私は重要なポイントの解説しかしなイカら、コードの全文を http://www.paraiso-lang.org/ikmsm/ からダウンロードし、あわせて読むでゲソ！

まず、このプログラムにおける SKI 式はイカのデータ型で表現されているでゲソ。

```
data Expr = M | S | K | I | Ap Expr Expr
          | Succ | N Int
          | Bottom String
          | Church Integer
          | Literal String
            deriving (Eq, Ord, Show, Read)
```

- このうち M, S, K, I はコンビネータでゲソ。前にも言ったように本当は S と K だけで用は足りるのでゲソが、輝かしい地上の支配者である IKMSM の名をしるすために I と M もアルファベットに加えるのでゲソ！ Ap は関数適用を表すでゲソ。Ap と S, K, I(, M) は SKI 式の根幹でゲソ。
- つづいて、N は整数、Succ は整数に 1 を加える 1 引数関数でゲソ。チャーチ数に対して Succ と (Num 0) を引数に与えると、そのチャーチ数がなんの整数を表していたのか評価できるじゃなイカ？ だからこれらは検算に使わせていただくでゲソ。
- Bottom は計算がエラーになったことを表すでゲソ。エラーの原因を示す文字列を引き連れているのは便利のためでゲソ。
- Church n は、チャーチ数 n を部分式として利用することを表すでゲソ。再帰的に部分式を利用していくうちに、可能な部分式の場合の数は指数関数的に膨らんでいくでゲソが、Church n があればそんな状況をも少ない資源で表現できるでゲソ。
- Literal は、好きな名前を持ち、動作は未定義のコンビネータでゲソ。足など飾りじゃなイカ？

これら SKI 式の eval は次のように書けるでゲソ。

```
eval :: Expr -> Expr
eval expr = case expr of
  (Ap(Ap(Ap S f)g)x) -> let fx = eval (Ap f x)
                            gx = eval (Ap g x)
                        in eval (Ap fx gx)
  (Ap (Ap K x) _) -> eval x
  (Ap I x) -> eval x
  (Ap M x) -> eval (Ap (S&!I&!I) x)
```

```
  (Ap Succ x) -> case x of
                   (N n') -> N (n'+1)
                   _      -> Bottom "cannot increment non-integer"
  (Ap (N _) _) -> Bottom "cannot apply an integer"
  (Ap bottom@(Bottom _) _) -> bottom
  (Ap (Literal x) _ ) -> Bottom $ "cannot evaluate literal :" ++ x
  x -> x
```

実に素直に書けるじゃないカ！

また、イカのコードのように、apply として (&) と (&!) という２種類の演算を用意するでゲソ。(&!) はその場で出来る限りコンビネータを評価する eager な性格なのに対し、(&) は構文木 Ap を組立てるだけで何も評価を行わない lazy な奴でゲソ。

```
evalAp :: Expr -> Expr -> Expr
evalAp x y = eval $ Ap x y

(&), (&!) :: Expr -> Expr -> Expr
infixl 1 &
infixl 1 &!
(&) = Ap
(&!) = evalAp
```

これらの道具をつかえば、小さなチャーチ数を手書きするのはとっても楽しいでゲソ！

```
csucc :: Expr
csucc = S & (S & (K&S) & K)

rei, ichi, ni, san, yon::Expr
rei = K & I
ichi = I
ni = csucc & I
san = csucc & ni
yon = M&ni
```

ではいよいよチャーチ数の生成を実装していくでゲソ！ 私こと海からの使者、イカ娘は欲しい者は真っ先に手に入れる主義でゲソ。だからまず次の行を書くでゲソ！

```
candidate :: Vector [Expr]
candidate = undefined
```

candidate は Expr のリストの配列でゲソ。Data.Vector はイカデックスが 0 から始まる、Haskell の使いやすい配列ライブラリでゲソ。candidate の n 番目には、チャーチ数 n の表現のうち最短のものすべてのリストを入れるでゲソ！

undefined とはイカに？ undefined :: a とは、何者にもなれる型を持つ代わりに何者にもなれない、お前達のような物体でゲソ。ひとまずこう書いておけばコンパイルは通り、undefined が評価されるまでプログラムは何事もなく実行されるでゲソ。私のような lazy(褒め言葉) なプログラミングスタイルにはぴったりじゃないカ！

この candidate が完成していることを前提にすれば、n 番目のチャーチ数の表現を何か 1 つ返す関数 getSolution は次のように書けるでゲソ。

```
getSolution :: Integer -> Expr
getSolution = head . (candidate !) . fromInteger
```

このままでは、帰って来た式は別のチャーチ数 Church m を部分式として含んでいるんじゃなイカ？ だから再帰的に Church を取り除いて SKI 式を作る関数 unchurch を用意するでゲソ！

```
unchurch :: Expr -> Expr
unchurch expr = case expr of
  (Ap x y) -> Ap (unchurch x) (unchurch y)
  Church x -> unchurch $ getSolution x
  x -> x
```

SKI 式の重みは、部分チャーチ数を取り除いた (unchurch) 結果を綺麗な文字列に変換した (pprint) 結果の文字列の長さ (length) としようじゃなイカ！ これは、文字通りそのまま実装できるでゲソ。何とも素晴らしい言語じゃなイカ！

```
weight :: Expr -> Weight
weight = length . pprint . unchurch
```

ではいよいよ candidate を実装するでゲソ。

```
solveSize = 256

candidate0 :: Vector [Expr]
candidate0 = V.replicate solveSize [] // [(0, [rei]), (1, [ichi]), (2,[ni])]

candidate :: Vector [Expr]
candidate = V.accum better candidate0 $
  concat (map (shrink . expand) [2..solveSize-1])
    where
      better :: [Expr] -> Expr -> [Expr]
      better [] x = [x]
      better xs@(x1:_) x2
         | elem x2 xs              = xs
         | weight x2 < weight x1   = [x2]
         | weight x2 == weight x1 = x2:xs
         | otherwise               = xs
```

このコードのキモは次の２行でゲソ！

```
candidate = V.accum better candidate0 $
  concat (map (shrink . expand) [2..solveSize-1])
```

candidate は、初期の候補 candidate0 をもとに、[2..solveSize-1] の整数を expand して得られた新しい候補たちを、better というより良いものを選ぶ関数により accumulate して作られるのでゲソ。better は、既存の候補リスト xs にたいして新しい候補 x2 が見つかったときの新しい候補リストを記述しているでゲソ！

- 新候補が既存候補リストにすでに含まれていれば変更なし
- 新候補のほうが weight が小さければそいつが単独トップでゲソ！
- weight が等しかったらみんなで集まって仲間でゲソ！ 独りぼっちは寂しいもんな！
- 新候補のほうが weight が大きかったとしたら、そんな候補に存在価値なんてないんだよ・・・

では expand とはイカに？ expand n は、チャーチ数 n の作り方は既知としたときに、それを応用すればチャーチ数 m は expr という式で書けるよ、という候補のリスト [(m,expr)] を返すでゲソ！

```
expand :: Integer -> [(Integer, Expr)]
expand n = (n^n, M & nexpr) : (n+1, csucc & nexpr)
         : (n*(n+1), M&csucc&nexpr)
         : (bySum ++ byProd ++ byPow)
    where
      nexpr = Church n
      others = [(m, Church m) |m<-[2..n]]
      bySum  = generate (+) (\mx nx -> mx & csucc & nx)
      byProd = generate (*) (\mx nx -> (S&(K & mx) & nx))
      byPow  = generate (flip (^)) (&)
      generate fint fexpr =
          [(fint n m, fexpr nexpr mexpr) |(m, mexpr)<-others]++
          [(fint m n, fexpr mexpr nexpr) |(m, mexpr)<-others]
```

前章のセクション 1.2.6 でも解説したように、n 単独で作れる $n^n, n+1, n(n+1)$ のほか、`[2..n]` の範囲のもうひとつの整数 m と組み合わせて足し算、掛け算、冪乗によって作れる可能性を探求しているでゲソ。

`shrink` は列挙された候補のうち `solveSize` イカの物だけを選ぶなど瑣末な処理をしているでゲソ。

```
shrink :: [(Integer, Expr)] -> [(Int, Expr)]
shrink = map (\(n,x) -> (fromIntegral n,x)) .
         filter (\(n,_) -> 0<=n && n < solveSize)
```

以上のコードにより無事、すべてのチャーチ数の候補が求まるでゲソ。めでたしめでたしでゲソ！

栄子「異議有り！ うちの目を欺けるとでも思ったか？ 今の話には自己撞着があるじゃない！ `Expr` の配列変数 `candidate :: Vector Expr` の構築には長さの比較のために `weight` を利用しており、`weight` は部分 `Church` 式除去のために `unchurch` を利用しており、その為には `candidate` 配列が完成して `getSolution` が引ける状態になっていることが前提だったじゃないか！」

大丈夫、大丈夫でゲソ！ `Vector` は遅延評価される純粋計算な配列でゲソ。実際の計算がいつ行われるのか誰にも分からないし、いつ計算を行っても、だれにも影響を与えない。始まりも、終わりもない状態にあるでゲソ。`candidate` で `candidate` を定義するようなコードを書いても、円環の理の導きにより、`candidate` は必要に応じて小さいほうから埋まっていき、ちゃんと値を返してくれるでゲソ。

この、$O(1)$ でアクセスでき、更新も最小限で済む純粋な配列を作るのには、並ならぬ努力が払われているのでゲソ[*1]。純粋さを願った代償は決して小さくなかったでゲソ。私みたいな頭のよいイカでもなければ、純粋関数的なデータ構造のライブラリを設計するのは無理になってしまったのでゲソ。だがいったんライブラリができてしまえば、その恩恵はすごいなんてもんじゃない、途方も無いのでゲソ。

「時間を空間に」という Haskell の標語を聞いたことがなイカ？ 遅延評価と純粋計算の組み合わせにより、データ構造 X に対し、X の要素を X の他の要素を使って再帰的に定義するようなコードを書くだけで、必要な箇所だけが計算され、動的プログラミング (DP) のようなコードがいとも簡単に作れてしまうでゲソ。

恐怖と感動で言葉も出ないようでゲソね。でも心配いらないでゲソ。最初はつまづいてばかりで使うのがやっとだった読者も、すぐに PFDS [*2]を読みこなし、純粋なデータ構造を作れるまでになれるでゲソ。やがて関数型 Wizard になる君たちのことは、関数型魔法少女と呼ぶべきじゃなイカ？

[*1] Coutts, Leshchinskiy and Stewart2007, *Stream fusion: from lists to streams to nothing at all* `http://portal.acm.org/citation.cfm?id=1291199`

[*2] ChrisOkasaki, *Purely Functional Data Structures*

QpY

第3章

関数型イカガール

— @tanakh

3.1 関数型イカガール第一話

高校一年の夏。

日も暮れてがらんとした「海の家れもλ」、そのテーブルで僕はノートを広げ一人考えに耽っていた。昼間の賑わいが嘘のように寂しげな砂浜を眺めながら、イカ墨パスタを口に運び、合間合間にペンを走らせる。至福のひとときだ。

「ポイントフリーでゲソか？」

いつの間にか、僕のノートを覗き込んでいる小さな頭があった。この海の家でアルバイトをしているイカさん。白くて妙な形をした帽子をかぶり、そこから青く長い髪が覗いている。いや、髪なのだろうか。髪のようなそれは風もないのにゆらゆらと空中を揺らめいている。まるで意識を持って動いているような...。

「仮引数を一つずつ落として、ラムダ式に書き換えるでゲソ。それから flip を使って束縛変数を後ろに移動させて変数を削除する、覚えるまでもないじゃなイカ？」

いいんだよ、練習しているだけなんだから。僕のノートにはいくつかの有名関数の定義が書いてあって、それらをポイントフリーに書き換えようとしている最中だった。

イカさんは髪で僕のペンを奪い、さらさらとノートに式を書き込む。矢印で左右から関数を挟み込んだような式だ。

「ほら、これはなんでゲソ？」

Squish パターンだ、僕は心のなかで答えた。Squid に掛けているのだろうか。突っ込むべきか迷ったが、結局口には出さない。二匹のイカが関数を侵略しているようにも見えるその式を、じっと見つめていた。

「分からないでゲソか？ Squish パターンじゃなイカ」

そう言ってイカさんは少し体を起こす。微かにイカ臭い香りが漂った。

イカさんはしたり顔で話し始める。「Squish パターンは f >>= a . b . c =<< g の形のポイントフリー式でゲソ。二つのモナドの計算結果をひとつの関数でまとめるのに使えるでゲソ。二つのものを一つのモナドに変換する、合成すると言ってもいいでゲソ。複数のモナドを合成できるようになったとき、とても素敵なことが起こるでゲソ」

イカさんの声を聞きながら、僕は別のことを考えていた。イカの女の子。侵略者。その二つの姿が、一人の少女であると気づいたら、どんな素敵なことが起こるんだろう。

でも、もちろん僕は何も言わず、黙ってイカ墨パスタを食べていた。

3.2 関数型イカガール第二話

夏休み。

いつものように僕は日の暮れた海の家れもλのテーブルでノートを広げていた。ノートには一列に並んだ数字。そしてそれらを書き換える操作。

「配列操作でゲソか？」

いつものようにイカの少女がノートを覗き込んできた。彼女はイカさん、関数の海からの侵略者だそうだ。今は訳あってここ、関数型海の家れもλでウェイトレスをしている。小柄な少女の頭にちょこんと乗った白い特徴的な帽子から伸びる十本の触手がせわしなくうごめいている。

僕は配列について考えていた。関数型言語における配列。参照透明な言語における immutable な配列。こ

れをうまく扱う方法について考えあぐねていた。mutable な配列を使うという選択肢もあるにはあるが、関数型言語としては美しくない。immutable でかつ読み書きを定数時間で行うことのできるデータ構造は無いものか――――。

「ふーん」イカさんは中空を見つめながら「大きく分けて 3 つの方法があるでゲソ」勿体ぶってそう言った。

3.2.1 永続データ構造

「まず一つ目は、永続データ構造[*1]による配列操作の実現でゲソ」

purity と immutable とくれば persistent data structure に行き着くのはある意味自然だ。しかしこれは効率が犠牲になる。最も自明なのは、単純なデータの列として配列を表現する、つまり単なる配列の immutable 版だ。でもこれは更新の効率が極端に悪い。更新するたびに全てをコピーして新しい配列を作らなければならない。

「更新操作をまとめて行えば、効率は上げられるでゲソ。あるいは」少女はいたずらっぽく言った「構築に対して読み込みが多ければ問題ないじゃなイカ」

例えば静的な検索インデックスのような、構築一回に対して大量のリードオンリークエリが生じるのであればこのような単純な手法でも十分実用になる。しかしそういうケース以外だと採用は難しい。

「一般のケースを考えると、バランス木[*2]、フィンガーツリー[*3]あたりでゲソか」

バランス木を配列として用いるのもある意味自明だ。計算量が読み書きともにワーストケース O(logn) で可能だし、何よりシンプルだ。だけど、メモリ使用量、実行時間ともに命令型言語の mutable な配列を用いた場合に比べて激しく劣る。フィンガーツリーは Haskell には Data.Sequence[*4]として存在するがこちらもそんなに速いわけではない。これらは永続データ構造なのだ。永続性が必要ない場合にこれらはオーバーヘッドにしかならない。

イカさんは「永続性を捨てればいいじゃなイカ」またもいたずらっぽくそう言った。

3.2.2 Linear Type

それはそうだろう。だけど純粋関数型言語でそれをどうやって実現するのか。

まずひとつはモナドを使う方法だ。配列操作を副作用とみなして、すべての操作を IO モナドに包む。これは Haskell で言う所の IOArray[*5]だ。

「STArray[*6]もあるでゲソ」

そうだ。副作用の及ぶ範囲を特定のメモリ操作に限定したモナド、ST モナド。これを使うと外部からは純粋な関数に見える操作を命令的に書くことができる。さらに IO と ST モナドを統合した PrimMonad モナド[*7]というものを用いるとこれらの配列操作をポリモーフィックに扱うこともできる。

だけど、これらは基本 mutable な操作になり、コードは極めて命令的になる。そういうコードを書くのであれば命令型言語のほうがすっきりとしたコードになってしまうのが僕にはなんとなく許せなかった。

「型システムの拡張を考えるでゲソ」イカさんは言った。

型システムの拡張？ 僕が答えに詰まってると「Clean には、Linear Type[*8]というものがあるでゲソ」少女は僕からペンをひったくり、さらさらとコードを書いた。

```
geso :: !Int -> *{#Int} -> *{#Int}
geso ix arr = { arr & [ix]=100 }
```

何やら型に変な記号が付いている。

「!は Bang パターンでゲソ。正格性を表してるでゲソね」ふむふむ。

「{} は配列、#は Unboxed な型を表しているでゲソ。そして*が一番重要なんでゲソが...」イカさんはふっふっふと勿体ぶって言った。

[*1] http://en.wikipedia.org/wiki/Persistent_data_structure

[*2] 子の高さに偏りのない木のこと

[*3] http://en.wikipedia.org/wiki/Finger_tree

[*4] http://hackage.haskell.org/package/containers

[*5] http://hackage.haskell.org/packages/archive/array/0.3.0.2/doc/html/Data-Array-IO.html

[*6] http://hackage.haskell.org/packages/archive/array/0.3.0.2/doc/html/Data-Array-ST.html

[*7] http://hackage.haskell.org/package/primitive

[*8] http://en.wikipedia.org/wiki/Linear_type_system

「Linear Type でゲソ！」

Linear Type とは初めて聞いたが、要はその変数が参照される回数が “高々一回” であることを保証する型らしい。例えば次のようなコード

```
gesho :: *Int -> Int
gesho n = n + n
```

はコンパイルエラーになる。Linear Type の変数が二回参照されているからだ。

「一回しか参照されない型。これがどういうことかわかるでゲソか？」

しばし考える僕。そうか。一回しか参照されない型、実行時には変数だけど、そういう変数は破壊的更新が可能になるんだ。なぜなら、破壊前の変数が参照されることが絶対にないということが、“型レベル” で保証されているのだから。

破壊的更新を行っても、参照透明性を維持することはできる———。僕は改めて型システムの力の強力さを思い知った。

「型の力に驚いたでゲソか」イカさんはキリッとした表情でこちらを覗き込んでいる。そしてそのままこちらにぐいと顔を近づけてきた。少しドキリとした。

「まだまだ、こんなもんではないでゲソ」

3.2.3 Stream Fusion

「型システムの拡張とは別のアプローチで攻めるでゲソ。これは vector[*9] パッケージで採用されている手法なんでゲソが」

まだ何かすごいものが飛び出してくるのだろうか。僕の中の好奇心がざわめき立つ。

「Stream Fusion というテクニックがあるでゲソ」

はて、どういうテクニックなんだろうか。僕が戸惑っていると「じゃあその前に」イカさんは言った。

「Haskell の書き換え規則は知ってるんじゃなイカ？」

それはさすがに知っている。特定の形の式を書き換えるルールを記述できるというやつだ。これは Prelude にも使われている。例えば map 関数の場合。

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

これが GHC の標準ライブラリでの定義だ。とても素直な定義だが、これがこのまま実行されるわけではない。書き換え規則によって、コード変換が行われる。

```
{-# RULES
"map"       [~1] forall f xs.   map f xs
  = build (\c n -> foldr (mapFB c f) n xs)
-}
```

map f xs は build (\c n -> foldr (mapFB c f) n xs) に変換されるということだ。mapFB とはなんだというと、map のそばに次のように定義されている、内部的な関数だ。

```
mapFB ::  (elt -> lst -> lst) -> (a -> elt) -> a -> lst -> lst
mapFB c f = \x ys -> c (f x) ys
```

これは実際には map をη展開した形になっている。まあ細かいことは今はいい。重要なのは、次の書き換え規則だ。

```
{-# RULES
"mapFB"     forall c f g.       mapFB (mapFB c f) g
```

[*9] http://hackage.haskell.org/package/vector

```
  = mapFB c (f.g)
  #-}
```

つまり、map f . map g のような式は map (f.g) のような形に変形され実行される。これはすなわちリストの中間データが生成されないより効率的なコードに変換されることを意味する。

「Purity はすごいでゲソね」

そうだ。これは参照透明だからなせる業だ。値が変わらないことと副作用が無いことがわかるから、自由に式を変形することができる。変形後の式が元の式と同じ値になることを保証することはさすがに今のところプログラマの仕事ではあるのだけど。

「ま、それは置いておいて」イカさんは楽しそうに言う。「Stream Fusion っていうのは、リストなどの操作を Stream の操作に一旦変換して融合変換を行うテクニックでゲソ」

そう言ってスラスラとペンを走らせ始めた。うねうねと揺らめく触手が僕の目を惑わせる。

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

「map の宣言はさっきと同じでゲソ。違うのは書き換え規則の方でゲソ」そう言ってイカさんは式を書き足した。

```
{-# RULES
"map -> fusible" [~1] forall f xs.
    map f xs = unstream (Stream.map f (stream xs))
  #-}
```

「map の際に一旦 Stream という型に変換するんでゲソね。そして終わった後 unstream でもとに戻すでゲソ」

それで何が嬉しいんだろうか。操作 f . g . h は unstream . Stream.f . stream . unstream . Stream.g . stream . unstream . Stream.h . stream になって...

「stream . unstream は明らかに無駄じゃなイカ。当然そういうのを削除するルールは入っているでゲソ」

```
{-# RULES
"STREAM stream/unstream fusion" forall s.
    stream (unstream s) = s
  #-}
```

じゃあさっきのは unstream . Stream.f . Stream.g . Stream.h . stream になるわけだ。リストに対する操作が stream に対する操作に変わったことで何が変わるのだろう。

「じゃあ、Stream の具体的な実装を見てみることにするでゲソ」

僕は Data.Stream モジュールのソースを眺めて、Stream の定義を探した。

```
data Stream a = forall s. Unlifted s =>
                          Stream !(s -> Step a s)  -- a stepper function
                                 !s                -- an initial state


data Step a s = Yield a !s
              | Skip    !s
              | Done
```

なんだこれは。Stream は実際にはデータの列ではなくて、列を生成する関数に過ぎないのか。これに対する map 操作は、

```
map :: (a -> b) -> Stream a -> Stream b
map f (Stream next0 s0) = Stream next s0
  where
    {-# INLINE next #-}
    next !s = case next0 s of
        Done       -> Done
        Skip    s' -> Skip        s'
        Yield x s' -> Yield (f x) s'
{-# INLINE [0] map #-}
```

このように定義されている。map f . map g はどう最適化されるんだろう。

```
map f (map g xs)
  ==> map f (map g (Stream next0 s0))
  ==> map f (Stream next s0
        where
          next !s = case next0 s of
              Done       -> Done
              Skip    s' -> Skip        s'
              Yield x s' -> Yield (g x) s')
```

うーむ、ややこしい。式がごちゃごちゃしてきた。

「ちょっと貸してみるでゲソ」イカさんは僕からまたもペンをひったくり、さらさらと式を展開していく。

```
==> Stream next' s0
      where
        next' !s = case next s of
            Done       -> Done
            Skip    s' -> Skip        s'
            Yield x s' -> Yield (f x) s')

        next !s = case next0 s of
            Done       -> Done
            Skip    s' -> Skip        s'
            Yield x s' -> Yield (g x) s')
==> Stream next' s0
      where
        next' !s =
          case (case next0 s of
                  Done       -> Done
                  Skip    s' -> Skip        s'
                  Yield x s' -> Yield (g x) s')) of
            Done       -> Done
            Skip    s' -> Skip        s'
            Yield x s' -> Yield (f x) s')
```

「ここで何か気づかなイカ？」

最後の式は case がネストしている。内側の case で Done の場合は Done に、Skip の場合は Skip に、Yield の場合は Yield を返しているな。あっ、これは外側の case はテストする必要が無いじゃないか。

「その通りでゲソ！」イカさんは満足気に続けた。「このような case 式のネストは単純な case 式に書き換え

ることができるでゲソ。これを case-of-case transformation と呼ぶでゲソ。かっこいい名前でゲソ」

```
==> Stream next' s0
      where
        next' !s =
          case next0 s of
            Done       -> Done
            Skip    s' -> Skip        s'
            Yield x s' -> Yield (f (g x)) s'
```

二重のループが潰れてしまった。そしてこれはまさしく map (f.g) の形ではないか！ リストを Stream という最適化のかかりやすい形に変換して、それに対する操作にする。これが Stream Fusion というものか。僕は感心してしばらく呆然としていた。

3.2.4 Recycling

Stream Fusion を使うと、様々なリスト様の操作、例えば map、filter、fold などのあらゆる組み合わせを融合して、中間のデータを作らずに高速に計算することができる。これを配列にも応用すれば、配列に対して map や fold のような操作が連続したとしても中間の配列を作らずに済む。

だけど、これじゃあ足りない。Stream Fusion がうまくいくのはリストがシーケンシャルなデータ構造だったからだ。ランダムアクセスが入るととたんにだめになる。

「そのために Recycling というテクニックがあるでゲソ」イカさんはまたも得意げに胸を張った。

「例えば配列のある要素を書き換える操作があったとするでゲソ」

```
a // (1, 2)
```

「Haskell では // 演算子を用いて配列の更新を表すことが多いでゲソね」

これは immutable な配列に対して行うと、配列のコピーが発生する。嬉しくないことだ。

「更新が連続するとどうなるでゲソ？」

```
a // (1, 2) // (3, 4) // ...
```

イカさんの触手がスラスラと長い式を書いていく。こんなに何度も更新したら大量の一時的な配列が生成されてとても多くの無駄な処理が発生しそうだ。

しかし、中間で生成される値は次の更新された配列を生成するのに使われるのみで、二度と参照されることはない。これは Linear Type の場合と同じだけど、今回の場合は型による助けを借りることはできない。

「実際のところ、複数の更新処理をまとめて行う操作は提供されているでゲソ」

```
a // [(1, 2), (3, 4) ... ]
```

そうだ。これは自明な拡張だ。たくさんの更新をまとめてするだけなら、そういうプリミティブがあればいい。だが、中間の配列の生成を削除したい要求はもっと他の処理にも現れるはずだ。さっきの Stream Fusion のように一般的な削除が可能にはならないのだろうか。

「そこで Recycling でゲソ」事もなげにイカさんは言う。

Recycling というからには、中間的な配列が今後使われないと分かる場合に、それを破壊的に更新して再利用するのだろう。しかしどうやって？

「内部的に mutable な配列を使用するでゲソ」

あっ、と僕は声を出してしまった。Stream の代わりに mutable 配列に対する操作に変換すればいいのか。分かってしまえばあまりにも単純なことだ。

「例えばさっきの配列の更新の例でゲソが」

```
{-# INLINE (//) #-}
v // us = update_stream v (Stream.fromList us)
```

イカさんはサラサラと式を書きだす。この少女は頭の中にライブラリのコードが全部入っているのだろうか。そんな疑問を挟む暇もなく、

「us を Stream に変換しているのは些事でゲソ。それよりも、update_stream が重要でゲソ」さらにイカさんは続ける。

```
update_stream :: Vector v a => v a -> Stream (Int,a) -> v a
{-# INLINE update_stream #-}
update_stream = modifyWithStream M.update
```

実際に処理を行っているのは modifyWithStream という関数のようだ。modifyWithStream の定義を辿る。

```
modifyWithStream :: Vector v a
                 => (forall s. Mutable v s a -> Stream b -> ST s ())
                 -> v a -> Stream b -> v a
{-# INLINE modifyWithStream #-}
modifyWithStream p v s = new (New.modifyWithStream p (clone v) s)
```

「この clone というのが、配列を mutable なものとして扱うための変換でゲソね。new がその逆。さっきの stream/unstream に対応するものでゲソ」

なるほど、New.modifyWithStream がその mutable なものに対する操作というわけか。

「実際にはここで生成されるものは New という特殊なデータ構造で、単なる mutable な配列ではないのでゲソが、まあそんなことはここでは置いておくでゲソ」

大雑把にはそういう理解でいいらしい。操作が複数挟まると clone/new の列が生成されるはずで、これも Stream Fusion のと同様に削除されるべきだろう。コードを漁ると思ったとおり、次のような書き換え規則が見つかった。

```
{-# RULES
"clone/new [Vector]" forall p.
  clone (new p) = p
 #-}
```

「配列に対しても Stream Fusion は実装されているでゲソ。これで map、fold のようなシーケンシャルな操作も最適化されるでゲソ」

それで配列のコピー回数を最小限に保つというわけか。immutable な配列に対する操作を、データ構造的な改善ではなく、コンパイラの最適化と型システムによって効率化してしまったということになる。O(1) 読み書きの永続配列をずっと考えていた僕にとっては、まさに発想の転換だった。

「海のゴミも配列もリサイクルが大事でゲソ。ポイ捨てはイカンでしょ。イカンでしょ」

フンフンと鼻を鳴らしながらイカの少女は海の家の奥へと消えていった。

3.2.5 Deforesting と Supercompilation

帰りの電車にて。

乗客の殆ど居ない列車で僕はノート PC を広げていた。Stream Fusion と Recycling。あまりにも美しく、綺麗な手法だ。Purity とコンパイラの最適化を活用するとこんなことができてしまうのか。数多のポリモーフィックな配列から Stream という唯一のデータ型への変換。Stream の世界での操作。そして元のポリモーフィックな世界へ。そしてそれらがコンパイラの最適化でノーコストでできてしまう。頭では理解しつつも、にわかには信じがたいことだ。僕は実際にコードを書いてみることにした。

```
main :: IO ()
main = do
  file <- head <$> getArgs
  s <- B.readFile file
  print . mostFreq . bsToVector $ s
```

```
bsToVector :: B.ByteString -> V.Vector Word8
bsToVector s =
  V.generate (B.length s) (B.index s)

mostFreq :: V.Vector Word8 -> Int
mostFreq s =
  V.maximum $
  V.accumulate (+) (V.replicate 256 (0 :: Int)) $
  V.map (\c -> (fromEnum c, 1)) s
```

これはファイルを引数に取り、そのファイルの中に出現する最頻文字の出現回数を表示するプログラムだ。文字の配列を、文字コードと 1 のタプルに変換し、一つ目の添字で足しあわせ、最後に最大値をとっている。C++ で書けば次のようになるだろう。

```
int main(int argc, char *argv[])
{
  vector<int> cnt(256, 0);
  ifstream ifs(argv[1]);

  for (char c; ifs.get(c); )
    ++cnt[(unsigned char)c];

  int maximum = cnt[0];
  for (size_t i = 0; i< cnt.size(); ++i)
    maximum = max(maximum, cnt[i]);

  cout << maximum << endl;

  return 0;
}
```

さて、実行してみよう。適当に大きなファイル（565MB）を食わせてみる。ちなみにファイルはすべてキャッシュに載った状態での計測なのでディスクの速度がボトルネックになるとは考えなくて良い。

```
$ ghc -O2 MostFreq
[1 of 1] Compiling Main             ( MostFreq.lhs, MostFreq.o )
Linking MostFreq ...
$ time ./MostFreq Fedora-15-i686-Live-Desktop.iso
3387261

real  0m2.427s
user  0m2.127s
sys   0m0.290s
```

速い。実行中のメモリ使用量を測ってみると、ファイルサイズとちょうど同じぐらいのサイズだ。確保されたメモリは最初に strict に読み込んでいる文字列のメモリだけで、中間の配列は一切作られていないことが分かった。C++ の方も動かしてみようか。

```
$ g++ -O2 MostFreq.cpp
$ time ./a.out Fedora-15-i686-Live-Desktop.iso
```

```
3387261

real  0m6.321s
user  0m6.203s
sys   0m0.093s
```

C++ の方は一文字ずつ stream から読み込むという酷い実装ではあるが、少なくともこの実装では Haskell のほうが速い。Haskell の方も straight forward な実装で、immutable な配列を使っていることを考えるとこれは充分な速度だ。

immutable な配列の操作がこんなに高速になり得ることが分かって、僕は少し興奮した。やはり Purity は制約ではない。我々にひとつ上のレベルのコーディングを可能にさせてくれるものなのだ。

あまりに技巧的である。素晴らしい技術だ。だけど世の中には愚直なコードもたくさんある。リストをそのまま使ったコードだってたくさんある。こういうコードを高速化することはできないのか？ 理想的には map f . map g のような操作はコンパイラが自力で判断して map (f . g) に変換されるべきだ。書き換え規則は正しいかどうかをコンパイラが検証できないし、組み合わせが多くなってくると全てを記述するのは事実上不可能になる。

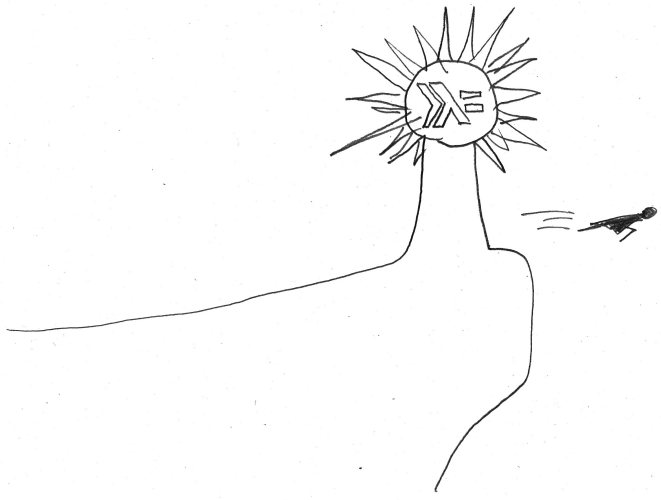
僕は WiMAX[*10]でインターネットに接続し、そういう研究が無いか論文を検索してみた。こういう時に Google Scholar[*11]は便利だ。すると ICFP[*12]2010 の論文に Supercompilation というものを見つけた。プログラムを字面のまま実行することにより最適化されたコードを生成できるものらしい。それによる中間データの削除を Deforesting と呼ぶらしい。

「次は、○○〜、○○〜」論文をダウンロードしようとしたとき、到着を告げるアナウンスが流れた。家に帰って寝る前に読もう。僕はまだ、関数プログラミングの世界に広がる海にほんの少し漕ぎ出したばかりなのだ。

3.3 参考文献

- 結城浩「数学ガール」http://www.amazon.co.jp/dp/4797341378
- ミルカさん http://www.hyuki.com/story/miruka.html.
- Pointfree(HaskellWiki)[http://www.haskell.org/haskellwiki/Pointfree]
- 「本物のプログラマは Haskell を使う」第 40 回〜第 42 回
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20100706/349960/
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20100803/350961/
  - http://itpro.nikkeibp.co.jp/article/COLUMN/20101013/352848/
- Duncan Coutts and Roman Leshchinskiy and Don Stewart. Stream Fusion. From Lists to Streams to Nothing at All. ICFP' 07
- Neil Mitchel. Rethinking Supercompilation. ICFP' 10

[*10] 高速モバイル通信の一つ http://www.uqwimax.jp/
[*11] 論文検索エンジン http://scholar.google.co.jp/
[*12] 関数型言語の国際学会 http://www.icfpconference.org/



第4章

加速しなイカ？

— @nushio

4.1 夢のようなライブラリがあったじゃなイカ！

人類よ、よく聞け！ フリーランチは終わったでゲソ！ これからは誰もが~~ボッタクry~~良心的価格のランチを得るために並列プログラミングという重労働を強いられる時代でゲソ。CPU もシンディーちゃんにブルドーザーちゃんに、どんどんベクトル化／マルチコア化してきているし、GPU なら既に何万というスレッドを並列に動作させることができるでゲソ。さらに将来イカに異様なハードウェアが現れても、我々はプログラムを書いてイカざるを得ないでゲソ。ならばプログラムを操る程度の能力を持った言語が是非とも必要でゲソ。いまこそ Haskell の力を見せつけて、闇の言語どもを侵略しようじゃなイカ！

Haskell には Accelerate `http://www.cse.unsw.edu.au/~chak/project/accelerate/`なる高速配列演算ライブラリがあって GPU 計算ができるでゲソ。最新のコードは彼らのレポジトリ `https://github.com/mchakravarty/accelerate` からチェックアウトでゲソ。チェックアウトしたら、そのフォルダで

```
> cabal configure
> cabal install
> cabal haddock
```

とやるとインなんとかされるでゲソ。`cabal haddock` を行えば `dist/doc/`イカにドキュメントが生成されるのでゲソ。Haskell のライブラリはいつだって、初見は意味不明不明じゃなイカ？ それでも型を合わせて動かしてみたり、Haddock のハイパーリンクでいつも同じ所に飛ばされたりしているうちに、おぼろげながらライブラリの構造が見えてくるでゲソ。Haddock はとても役に立つじゃなイカ！

いくつか注意点があるでゲソ。まず執筆時点で、Accelerate が依存している CUDA という Hackage が、CUDA3.2 までしか対応していないため、最新の CUDA4.0 を入れた環境には Accelerate がインストールできない模様でゲソ。おなじく執筆時点の情報でゲソが、Hackage にある Accelerate(0.8.1.0) は github の (0.9.0.0) と比べてだいぶ遅れているので注意が必要でゲソ。この解説は Accelerate の開発者らによる論文 *Accelerating Haskell Array Codes with Multicore GPUs*, Chakravarty et. al. (2011) を基にしているので、そちらも参照でゲソ。

さて、Accelerate は、ただの GPU 向けコードジェネレータではなく、あくまでも Haskell プログラムに GPU 計算の速度を兼ね備えさせたいという発想でゲソ。そのため GPU 計算をモナドなどとして表現**せず**、純粋な計算にしていることが大きな特徴でゲソ。IO にとらわれず、Haskell プログラムの任意の箇所から GPU 計算を呼び出せるのでゲソ。実行中 GPU 計算が要求されると、その場で CUDA プログラムを生成し、コンパイルし、自身にリンクすることで結果を得る。素晴らしい技術じゃなイカ！

~~変態~~紳士たる皆様は Accelerate の中身が知りたイカ？ Accelerate が生成するコードは次のようにして確認できるでゲソ。まず cabal インストール時に`-fpcache` オプションを指定することで Accelerate が生成した CUDA コードが保存されるようになるでゲソ。

```
> cabal configure -fpcache
```

デフォルトでは`$HOME/.cabal/share/accelerate-x.x.x.x/cache/`に連番ファイル名で保存されるようでゲソ。これではどのファイルがどれなのか分からないじゃなイカ！ そこで例えば、Accelerate のソースにイ

カのように一行を追加することで、

```
-- Data/Array/Accelerate/CUDA/Compile.hs
compile table key acc fvar = do
  . . . .
  pid <- liftIO $ do
    hPutStrLn stderr $ "filename:" ++ cufile ++ "\n" ++ show acc
```

どんな式がどの.cu ファイルに翻訳されたかわかるでゲソ！ cache/に既に存在するコードは使い回され、生成は行われないでゲソ。賢くなイカ？

4.2 GPU も、FPGA も、あるでゲソ！

いよいよ使ってみるでゲソ。イカのコードを Dotp.hs という名前で保存して

```
#!/usr/bin/env runhaskell
{-# OPTIONS -Wall #-}
import Data.Array.Accelerate (Acc, Vector, Scalar, (:.)(..))
import Data.Array.Accelerate.CUDA (run)
import qualified Data.Array.Accelerate as A

myShape :: A.DIM1
myShape =  A.Z :. 10

xs, ys :: Vector Float
xs = A.fromList myShape [83,72,73,78,82,89,65,75,85,33]
ys = A.fromList myShape [73,75,65,32,77,85,83,85,77,69]

accXs, accYs :: Acc (Vector Float)
accXs = A.use xs
accYs = A.use ys

dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
dotp as bs = A.fold (+) 0 (A.zipWith (*) as bs)

accAns :: Acc (Scalar Float)
accAns = dotp accXs accYs

ans :: Scalar Float
ans = run accAns

main :: IO ()
main = do
  print ans
```

実行するでゲソ。

```
> ./Dotp.hs
Array Z [53171.0]
```

GPU 計算ができたじゃなイカ！

このコードは Accelerate の計算モデルを端的に表しているでゲソ。基本的に、Accelerate は CPU 側とアクセラレータ側それぞれでの多次元配列の計算を表現しているのでゲソ。アクセラレータはなにも CUDA な GPU に限らず、Chakravarty et. al. (2011) によれば、将来的には LLVM や OpenCL、FPGA もサポートしうるとのことでゲソ。頼もしいじゃなイカ！

Accelerate では配列変数を `Array sh e` という型で表現するでゲソ。ここで `sh` は型クラス `Shape` に属する型で、配列の次元と添字を、`e` は型クラス `Elt` に属する型で、配列の要素を表すでゲソ。たとえば `Array (Z:.Int:.Int) Float` は `Float` を要素とし、`Int` の添字でアクセスできる二次元配列を表すことになるのでゲソ。まどろっこしイカ？ なら `Vector Float` とか `Array DIM2 Float` などの型シノニムも用意されているでゲソ。

つぎに、型構築子 `Acc` は GPU 上で行いたい計算を表しているでゲソ。`Acc` な `Array` とふつうの `Array` の

相互変換には、use と run という関数を使うのでゲソ。

```
use :: (Shape sh, Elt e) => Array sh e -> Acc (Array sh e)
run :: (Shape sh, Elt e) => Acc (Array sh e) -> Array sh e
```

Acc の正体もカラクリも、私は知っているでゲソ！ Acc は実際には、「こういう計算をしたい」という要求を表す抽象構文木で、Data.Array.Accelerate.Smart モジュールで定義されているでゲソ。main 関数をイカのように書き換えて実行すると、

```
main :: IO ()
main = do
  putStrLn $ "xs = "     ++ show xs
  putStrLn $ "accXs = "  ++ show accXs
  putStrLn $ "accAns = " ++ show accAns
  putStrLn $ "ans = "    ++ show ans
```

イカのように、中身の構文木が確認できるでゲソ。

```
> ./Dotp2.hs
xs = Array Z :. 10 [83.0,72.0,73.0,78.0,82.0,89.0,65.0,75.0,85.0,33.0]
accXs = use
  (Array Z :. 10 [83.0,72.0,73.0,78.0,82.0,89.0,65.0,75.0,85.0,33.0])
accAns = fold
  (\x0 x1 -> (+) (x0, x1))
  0.0
  (zipWith
     (\x0 x1 -> (*) (x0, x1))
     (use
        (Array
           Z :. 10 [83.0,72.0,73.0,78.0,82.0,89.0,65.0,75.0,85.0,33.0]))
     (use
        (Array
           Z :. 10 [73.0,75.0,65.0,32.0,77.0,85.0,83.0,85.0,77.0,69.0])))
ans = Array Z [53171.0]
```

この構文木は Acc を使った計算を積み重ねるたびに構築されてゆき、run したときに初めて、まとめて実行されるのでゲソ。run という同じ名前と型の関数が、ふたつのモジュール Data.Array.Accelerate.Interpreter と Data.Array.Accelerate.CUDA で定義されており、それぞれインタプリタによる CPU 上での実行、CUDA プログラムを生成しての GPU 上での実行を意味するでゲソ。

4.3　実アプリの性能と向き合わなイカ？

遊びは終わりでゲソ！ 何か実用的なプログラムを書いて、Accelerate がどこまでやれるのか、見届けてみようじゃなイカ！ 実用アプリといえば真っ先に出てくるのは流体計算でゲソ。・・・納得イカないという顔でゲソね…美しい参照透明な海を守る為には、人間共にもっと海のことを理解してもらわないとイカんじゃなイカ！

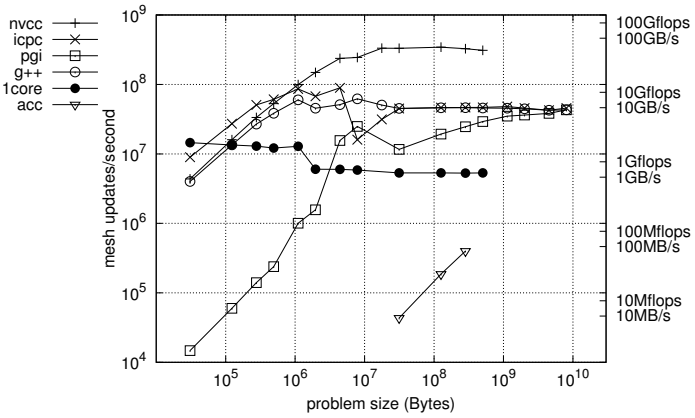
今回は Lattice Boltzmann 法という、コンパクトで安定なアルゴリズムを実装するでゲソ。Lattice Boltzmann 法の詳細については、説明を省略するでゲソ。興味のある方は参考文献[*1] でもイカがと思うが、知らなければ知らないで、何の不都合もないでゲソ。

いくつかの言語やコンパイラで Lattice Boltzmann 法のソルバを実装して、ベンチマークを取ってみた結果が図 4.1 でゲソ。って、どういうことだおいィ？ こいつ、遅いじゃなイカ！ 最速の場合でも、CUDA コードの 1/500 しか出なかったでゲソ…こんな性能、断じて満足するめイカ?!

これには理由があるでゲソ。私の実装はほぼあらゆる計算に A.use . run :: Acc (Array sh e) -> Acc (Array sh e) という関数を噛ませるようになっているでゲソ。この関数、意味は id :: a -> a にすぎないでゲソが、構文木を寸断して、計算を強制させる作用があるでゲソ。そんな事したら遅いし、折角

問題の最適化については元論文の 3.3 節を参照でゲソ！ バグについては現在、開発者に報告して返事待ち中でゲソ。

[*1] http://physics.ndsu.edu/fileadmin/physics.ndsu.edu/Wagner/LBbook.pdf



命令	単価	数量	合計
add	1f	14	14f
sub	1f	16	16f
neg	1f	3	3f
mul	1f	35	35f
div.rn	4f	2	8f
fma.rn	2f	27	54f
mad.hm	priceless		
ld.global	4B	36	144B
st.global	4B	18	72B
総計			130f
			216B

図 4.1: 統計よ。GPU は 1 枚の M2050(1.15GHz 14MP x 32Cores/MP=448Cores), CPU は 2 枚の Intel Xeon X5670(2.93GHz, 6Cores x HT = 12Cores) を利用したわ。nvcc は nvcc 3.2 V0.2.1221 でコンパイルしたもの、icpc,g++,pgi はそれぞれ icpc 11.1 20100414, g++ 4.3.4, pgCC 10.6-0 で SSE や OpenMP 等の最適化を有効にして 24 スレッド並列で実行したもの、1core は g++ をシングルスレッドで実行したもの、acc は Accelerate でコードを生成しながら計算したものよ。参考までに、右側には nvcc の吐いた ptx を根拠に換算した、演算量 (f) とレジスタ-device メモリの転送量 (B) を併記したわ。

確かに、アクセラレータが純粋計算として使えるのは素晴らしい。でも、この用途には致命的ね。代入のない純粋な構文木は計算の重複を産むから、共通項を回復する最適化が必要になる。そして、どんな最適化にもバグはつきものよ。だから Accelerate は性能を落とした。でも、無責任を承知で言わせてもらうと、画像処理や行列だけじゃなく、様々なテストをしておくべきだったわ。一度言語を公開してしまったら、もうバグレポートから救われる望みなんてないのよ…

の最適化が働かなくなるじゃなイカ！ でもこうでもしないとコード生成段階でエラーが出て動かないのでゲソ。id を挟んだら挙動が変わるって、それってつまり最適化がバグってるってことじゃなイカ！ ほとんど死んでるコードを動かして、生きてるフリをしてるだけなのでゲソ！ 性能出なくて当たり前じゃなイカ！

4.4 そんなの、私が許さないでゲソ！

馬鹿かと思うかもしれないでゲソが、私は本当に Haskell では性能が出ないのか、それを確かめるまでは、諦めたくないでゲソ！ こうなったら、この短期間で Accelerate を理解して改造してや r[*2]ここは github に issue を投げ、信じて待つしかないでゲソね…

すべてを諦めかけていた締め切り当日。なんと Accelerate 開発チームから修正パッチが届いたでゲソ。A.use . run を抜いても動くようになった結果、約 5.6 倍の速度向上をみたでゲソ。確かにこれは、CUDA との 500 倍の差を覆すには小さな一歩にすぎなイカもしれないが、ある種の最適化がきちんと機能していることは実験できたでゲソ。Chakravarty たちも「fusion transformation などの本格的な最適化は future work だ」と言っているでゲソ。purity に基づく華麗な最適化に、期待してもいいんじゃなイカ？

私も、偏微分方程式をシミュレーションで解く、という自分の仕事を支援するために、ステンシル計算が扱えるコードジェネレータ http://www.paraiso-lang.org/wiki/を作りはじめたところでゲソ。すでにフロントエンドと、1CPU 向けのコードを吐けるバックエンド http://d.hatena.ne.jp/nushio/20110515 までは作ってあるでゲソ。Paraiso は Accelerate とは違って、Haskell の中から GPU 計算の結果を利用する機能はなく、コード生成に特化しているでゲソ。また Accelerate よりドメインを絞っている分、物理屋さんに読みやすい、数式処理システム的なフロントエンドを用意しているし、もしかしたら性能も出しやすいんじゃなイカ？ GPU 向けバックエンドが完成した暁には、また報告させてくれなイカ！

[*2] そんなの不可能に決まってるじゃなイカ／人◕‿‿◕人＼

畑違いの私がコードジェネレータを作ろうとするなんて、Haskell がなかったら想像もつかなかった事でゲソ。プログラムを自在に操り、パースし、生成し、最適化する…そういったメタプログラミングは、純粋、関数型、強力な型システムなどという Haskell の性質が最もイカせる分野の 1 つでゲソ。またその性質は Haskell を、実に多様な並列／平行パラダイムをサポート[*3] できるプラットホームたらしめているでゲソ。

さまざまな並列計算を簡潔に書けて、速度の出る言語。数多の研究者を束ね、言語の特異点となった Haskell なら、そんな途方もない望みも、叶えられるんじゃなイカ？

[*3] http://www.slideshare.net/skillsmatter/parallel-haskell

$$IM(S(S(KS)K)I)(S(S(KS)K))(S(S(KS)K)(M(S(S(KS)K))(S(K(S(S(KS)K)I))(M(S(S(KS)K)I)))))$$

$$K(S(S(KS)K)(S(S(KS)K)I(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I))))))K$$

$$M(S(S(KS)K))I(S(S(KS)K))(K(S(S(KS)K)(S(S(KS)K)I(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I))))))M)$$

$$S(S(KS)K)(K(S(K(M(S(S(KS)K))(S(S(KS)K)I))))(M(S(S(KS)K)I)(S(S(KS)K)I)))S)$$

$$M(S(S(KS)K)I)(S(S(KS)K))(K(S(S(KS)K)I(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I)))))M)$$

$$IS(S(KS)K)(K(M(S(S(KS)K))(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I)))))K)$$

$$K(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(M(S(S(KS)K)(S(S(KS)K)I)))))W$$

$$S(S(KS)K)(S(K(M(S(S(KS)K))(M(S(S(KS)K)I))))(S(S(KS)K)(S(S(KS)K)I)))$$

$$M(S(S(KS)K)I)(S(S(KS)K))(M(S(S(KS)K))(S(S(KS)K)I)(S(S(KS)K)I))$$

$$M(S(S(KS)K)I)(S(S(KS)K))(M(S(S(KS)K))(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I)))))$$

$$S(S(KS)K)(K(S(K(M(S(S(KS)K)I))))(S(S(KS)K)(S(S(KS)K)(M(S(S(KS)K)(S(S(KS)K)I))))))K)$$

$$M(S(S(KS)K)(S(S(KS)K)I))(S(S(KS)K))(K(S(S(KS)K)(M(S(S(KS)K)I)(S(S(KS)K)(S(S(KS)K)I))))M)$$

$$IS(K(S(S(KS)K)(S(S(KS)K)I)))(K(S(S(KS)K)(S(S(KS)K)(M(S(S(KS)K)I))(S(S(KS)K)I)))S)$$

$$K(S(S(KS)K)(S(K(M(S(S(KS)K))(S(S(KS)K)I)))(M(S(S(KS)K)I)(S(S(KS)K)I))))M$$

$$M(S(S(KS)K))(K(S(K(S(S(KS)K)I))(S(S(KS)K)(M(S(S(KS)K)I))))K)$$

$$S(S(KS)K)(M(S(S(KS)K)I))(S(S(KS)K)I)(S(S(KS)K))(K(M(S(S(KS)K)(S(S(KS)K)I)))W)$$

$$M(S(S(KS)K)I)(S(S(KS)K))(S(S(KS)K)(M(S(S(KS)K))(S(K(M(S(S(KS)K)I)))(S(S(KS)K)I))))$$

$$IS(K(M(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I)(S(S(KS)K)I)))$$

$$K(S(K(S(S(KS)K)(S(S(KS)K)I)))(IKMSM)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))$$

$$S(S(KS)K)(K(M(S(S(KS)K)I)(S(S(KS)K))(M(S(S(KS)K))(S(K(M(S(S(KS)K)I)))(S(S(KS)K)I))))K)$$

$$M(S(S(KS)K))(S(S(KS)K)I)(S(S(KS)K))(K(S(S(KS)K)(M(S(S(KS)K))(S(S(KS)K)I)(S(S(KS)K)I)))M)$$

$$M(S(S(KS)K))(S(K(M(S(S(KS)K)I)))(S(S(KS)K)I))(S(S(KS)K))(K(S(S(KS)K)(S(S(KS)K)I))S)$$

$$S(S(KS)K)(K(S(K(M(S(S(KS)K))(M(S(S(KS)K)I))))(S(S(KS)K)(S(S(KS)K)I)))M)$$

$$M(S(S(KS)K))(S(K(M(S(S(KS)K)I)))(S(S(KS)K)I))(S(S(KS)K))(K(S(S(KS)K)(S(S(KS)K)I))K)$$

$$IS(S(KS)K)(K(S(K(M(S(S(KS)K))(S(S(KS)K)I)))(M(S(S(KS)K)I)(S(S(KS)K)I)))W)$$

$$K(S(S(KS)K))(IKMSM)(S(K(M(S(S(KS)K)I))))(M(S(S(KS)K)(S(S(KS)K)I))))$$

$$M(S(S(KS)K)I)(S(S(KS)K))(S(S(KS)K)(S(S(KS)K)I(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I))))))$$

$$S(S(KS)K)(S(K(M(S(S(KS)K))(S(S(KS)K)I)))(M(S(S(KS)K)I)(S(S(KS)K)I)))$$

$$M(S(S(KS)K))(K(S(K(S(S(KS)K)I))(S(S(KS)K)(M(S(S(KS)K)I))))K)$$

$$IS(K(S(S(KS)K)(M(S(S(KS)K)I)))))(K(S(S(KS)K)(M(S(S(KS)K))(M(S(S(KS)K)I))))M)$$

$$K(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(M(S(S(KS)K)(S(S(KS)K)I)))))S$$

$$S(S(KS)K)(K(S(S(KS)K)I(S(K(S(S(KS)K)I)))(S(S(KS)K)(M(S(S(KS)K)I)))))M)$$

$$M(S(S(KS)K))I(S(S(KS)K))(K(S(S(KS)K)(M(S(S(KS)K)I)(S(S(KS)K)(S(S(KS)K)I))))K)$$

$$M(S(S(KS)K))(K(S(K(M(S(S(KS)K)I)))(S(S(KS)K)I))W)$$

$$S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)(M(S(S(KS)K))(M(S(S(KS)K)I)))))$$

$$M(S(S(KS)K)(S(S(KS)K)I))(S(S(KS)K))(M(S(S(KS)K))(S(S(KS)K)I))$$

第5章

OSを侵略しなイカ？

— @master_q

もうユーザ空間は参照透明な海で征服しつくしたでゲソ! 今度は libc や kernel を関数型で侵略する作戦を練るでゲソ!!!

5.1 HFuse でファイルシステムを侵略しなイカ？

HFuse <http://hackage.haskell.org/package/HFuse>

を使えば Linux 上でファイルシステムを Haskell で書けるでゲソ。

まずは使ってみるでゲソ。環境は 2011 年 6 月 30 日頃の Debian sid amd64 でゲソ。Debian sid はインストールするタイミングによって ghc と haskell-platform のバージョンに不一致があるので、Debian パッケージの ghc まわりでインストールエラーが起きたら気長に修正されるのを待つか古いパッケージを選択インストールするでゲソ。

```
$ uname -a
Linux casper 2.6.39-2-amd64 #1 SMP Tue Jul 5 02:51:22 UTC 2011 x86_64 GNU/Linux
$ sudo apt-get install fuse libfuse-dev haskell-platform
$ sudo adduser あなたのユーザ名 fuse
... 再ログインするでゲソ ...
$ cabal install HFuse
```

これで HFuse パッケージのインストールが完了したでゲソ。github にサンプルがある [*1] ので試しに使ってみるでゲソ。

```
$ git clone git://github.com/realdesktop/hfuse.git
$ cd hfuse/examples
$ make
$ mkdir mountdir
$ ./HelloFS mountdir
$ mount | tail -n 1
HelloFS on /home/hogehoge/src/hfuse/examples/mountdir type fuse.HelloFS
(rw,nosuid,nodev,user=kiwamu)
$ ls -l mountdir
合計 1
-r--r--r-- 1 kiwamu kiwamu 20 1970-01-01 09:00 hello
$ cat mountdir/hello
Hello World, HFuse!
$ fusermount -u mountdir
```

うん、うまく動いているでゲソ。

[*1] https://github.com/realdesktop/hfuse/blob/master/examples/HelloFS.hs

ところで HelloFS を起動して “fuse: failed to open /dev/fuse: Permission denied” と言われなイカ？ そういう場合はたぶん fuse グループへの自ユーザ追加を忘れているので、再チェックでゲソ。

5.2 ヲレヲレファイルシステムを作ってみるでゲソ!

さあお待ちかねでゲソ。ヲレヲレファイルシステムを作ってみるでゲソ!

ちょっと草植えときますね型言語 `Grass <http://www.blue.sky.or.jp/grass/doc_ja.html>`

なんてものを目にしたでゲソ。Grass 言語インタプリタとして振る舞うファイルシステムを作ってみなイカ？ まずは仕様を決めるでゲソ。

```
$ ./GrassFs mount
$ cat helloworld.txt
wvwWwwwwWwwwwwwWWWWwwWWWWWwwwwvwwwwWWWwwWwwWWWWWwwwwWwwvwWwWwwvwwWWwvwWWwwWW
WWWwwwWwwWWWWWWwWWWWWWWwWWWWwWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
wwwwwWWWWWwwwwwwwwWWWWWWwwwwwwwwWWWWWWWWWWWWWWWWWWWWWWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwWwwwwwwwwwwwwwwwWwwwwwwwWWwwwwwwwWWWwwwwwwwWWWWWwwwwwwwwwwwwwwwwwwwwwwwW
wwwwwwwwwwwWWwwwwWWWwwwwwWWWWwWWWWwwwwwwwwwwwwwwwwwwwwwWWWWWWWWWWWWwwwWWwWWWwWWWWWw
WWWWWWWWWWWWWWWWWwwwwwwwwwwwwwwwwwwWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwWWwwwwwwwwwwwwwWWWwwwwwwwwWWWWwWWWWWwwwwwwwwwWWWWWWwwwwwwwwwwwwwwwwwwwwwWWWWW
WWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwWWWWWWWWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wWWwwwwwwwwwwwwwwWWWwwwwwwwwwwwwwwWWWWwwwwwwwwwwWWWWwwwwwwwwwwwwwwwwwwwwwwwwwWWWW
WWwwwwwwwwwwwwwwwwwwwwwwwww
$ cat helloworld.txt > mount/grassvm
$ sync;sync;sync
$ cat mount/grassvm
Hello, world!
```

のようにファイルに書くだけで実行結果が得られるようにしたいでゲソ。インタプリタへの入力はとりあえずエラーにしてしまって手を抜かなイカ？ Grass 言語インタプリタ自体は State モナドを使って作ってみたでゲソ。[*2] コマンドラインとして使うには以下のように簡単でゲソ。

```
main :: IO ()
main = do
  s <- getContents
  case toGrassCode s of
    Left e -> do putStrLn "Error parsing input:"
                 print e
    Right r -> putStr $
               S.evalState stateGrass $
               initGrassState r
```

HFuse の API は以下のようでゲソ。

`http://hackage.haskell.org/packages/archive/HFuse/latest/doc/html/System-Fuse.html`

要は data FuseOperations を決めて、fuseMain 関数に食わせる main を書けばいいだけでゲソ。ということで data FuseOperations で何をするか考えなイカ？ 次の表に Operation 関数それぞれの役割をまとめたでゲソ。

各 Operation 関数の実装はほとんど HFuse の examples/HelloFS.hs をパクったでゲソ。もちろん fuseWrite は追加で必要になるゲソが、さらに fuseSetFileSize も必要でゲソ! これは HFuse ではなく FUSE 側の仕様で

[*2] https://github.com/master-q/Haskell-Grass-interpreter/blob/master/Grass.hs

FuseOperations	Implementation
fuseGetFileStat	lstat(2) - ファイルの状態を取得する
	“/grassvm” ファイルの長さを返す
fuseSetFileSize	truncate(2) - 指定した長さにファイルを切り詰める
	内部保持している Grass ソースコードの長さを切り詰める
fuseOpen	open(2) - ファイルのオープン、作成
	“/grassvm” ファイルのみ成功
fuseRead	pread(2) - 指定したオフセットでファイルディスクリプタを読む
	“/grassvm” を読むと内部保持している Grass ソースコードを実行し
	結果文字列を返す
fuseWrite	pwrite(2) - 指定したオフセットでファイルディスクリプタを書く
	“/grassvm” ファイルに書き込むと内部保持している Grass
	ソースコードを修正
fuseOpenDirectory	opendir(3) - ディレクトリをオープン
	“/” ディレクトリのみ成功
fuseReadDirectory	readdir(3) - ディレクトリを読み込む
	“/” ディレクトリを読むと “.”,“..”,“grassvm” を返す
fuseGetFileSystemStats	statfs(2) - ファイル・システムの統計を得る
	HelloFS.hs からてきとーにコピペ

表 5.1: Grass 言語ファイルシステムの FuseOperations

ゲソ。

> No creation (O_CREAT, O_EXCL) and by default also no truncation (O_TRUNC) flags will be passed to open(). If an application specifies O_TRUNC, fuse first calls truncate() and then open(). [a]
>
> [a] http://fuse.sourceforge.net/doxygen/structfuse_operations.html#14b98c3f7ab97cc2ef8f9b1d9dc0709d から引用

とあるように O_TRUNC 付きで open(2) を呼ぶとその前に truncate(2) が呼出されるでゲソ。この truncate(2) に相当するのが fuseSetFileSize 関数でゲソ。

5.3　fuseGetFileStat はファイルサイズを有限時間で返答しなければイカん

あとは FuseOperations を HelloFS.hs を参考にして作ればいいだけなんでゲソが注意があるでゲソ。

fuseGetFileStat 関数は FileStat 型を返すんでゲソ。この中には指定されたファイルの長さがあるでげそ。非効率な実装で許されるならば、プロセス内に保存されている Grass ソースコードを実行してみて、結果のサイズを返せばいいんでゲソ。しかし Grass は停止しないコードも作れてしまうでゲソ。

```
$ cat infw.in
wWWwwwwWWww
$ ./GrassCommand < infw.in
wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww... 無限に"w"を出力
```

こんなコードの出力サイズを調べたらファイルシステムが凍ってしまうでゲソ!!! 今回は指定サイズしらべてもリストが続いていたら諦める length 関数を作って回避したでゲソ。ちょっとずるイカ？

```
mylength :: [a] -> Int
mylength l =  len l 0
  where
    len :: [a] -> Int -> Int
    len [] n = n
    len (_:xs) n | n > (1024 * 1024) = n -- 無限に続きそうなら諦める
                 | otherwise = len xs (n + 1)
```

本来はキャラクタデバイスとして扱うべき対象なんでゲソが、FUSE はレギュラーファイルしか扱えないでゲソ。Linux ではまともに kernel に組み込むしか本質的な解がないでゲソ。。。

5.4 ということで完成したでゲソ

https://github.com/master-q/Haskell-Grass-interpreter/blob/master/GrassFs.hs

に HFuse 対応のコードを置いたでゲソ。ちゃんと最初の仕様通りに動くでゲソ。さすがに “wWWwwwwWWww” を write すると動作が遅くなるでゲソが。。。

5.5 この後は何を侵略するでゲソ？

今回はファイルシステムを侵略したでゲソ。FUSE を使うかぎりではファイルシステムしか侵略対象にできないでゲソ。じゃあ POSIX な OS はファイルシステムしか侵略できないんでゲソか？

そんなことはないでゲソ。Linux ではなく NetBSD には rump というプロジェクトがあるでゲソ。

http://www.netbsd.org/docs/rump/

これを使えば NetBSD kernel(やそれ以外の kernel) の多くの部分をユーザ空間で動かすことができるでゲソ。事実 TCP/IP プロトコルスタックをユーザ空間で動かした実績があるでゲソ! [*3]

次は TCP/IP プロトコルスタックを Haskell で書いて、NetBSD 上で動作させなイカ？ (次回に続く...λ)

[*3] http://journal.mycom.co.jp/articles/2009/03/30/asiabsdcon4/index.html

第II部

簡約!? 入力娘 (二期) (2011年冬)

第6章

クラスタリングしなイカ？

— @dekosuke

最近、ついったーのタイムライン・・・ゲフンゲフン、美しい海を見ていると、○○クラスタという単語をよく聞くでゲソ。例えば「関数型クラスタ」というと、関数型言語が好きな人たちの集まりを指すでゲソ[*1]。一般に、クラスタリングとは集団を似た者同士の小さなグループに分けることを言うでゲソ。クラスタリングは、その集団の性質を知るということにつながるでゲソ。言い換えれば、クラスタリングしてしまえばその領地を侵略したも同然でゲソ！！

というわけで、クラスタリングをやってみなイカ？

6.1 アルゴリズムとライブラリを決めなイカ？

やることは決まったので、クラスタリングのライブラリを探すでゲソ。Hackage のパッケージリスト

```
http://hackage.haskell.org/packages/archive/pkg-list.html
```

を、”Clustering”で検索すればいいんじゃなイカ？ ６件ぐらいヒットするんじゃなイカ？

このうち、”hierarchical-clustering” (階層クラスタリング）系のライブラリは興味深いけど話が複雑になるので今回は避けるでゲソ。hsgsom (自己組織化マップ) は高精度のクラスタリング手法でゲソが、ライブラリ作者がやる気を出しすぎて扱いが大変なので避けるでゲソ。

そして、残った希望、いやライブラリは・・K-Means のライブラリでゲソ。K-Means はクラスタリングにおける一番シンプルなアルゴリズムでゲソ。ソートで言うとクイックソートみたいなアルゴリズムでゲソ[*2]。

というわけで、kmeans ライブラリをインストール・・・するのが許されたのは 2011 年 10 月頭まででゲソ。なぜベストを尽くさなイカ？ 2011 年 10 月に kmeans より効率的な kmeans-vector ライブラリが作られたでゲソ。[*3]。List を使っている kmeans ライブラリと違い、kmeans-vector ライブラリは Data.Vector[*4]を使っているので 4 倍高速[*5]でゲソ！！！

```
cabal install kmeans-vector
```

で、必要なものはインストールできたんじゃないイカ？ え、ライブラリの内容が分からないでゲソか？大丈夫、K-Means は簡単なライブラリなので、中身を全部読めるでゲソ。短いのでここに転載してもいいんじゃなイカ？[*6]

```
type Vec = V.Vector Double
data Cluster = Cluster {
  cid :: !Int,
  center :: !Vec
  }
```

[*1] もちろん私も関数型言語クラスタに含まれているでゲソ

[*2] K-Means に関しては計算量がデータに対して NP 困難なので、大規模なデータに対しては近似手法が取られるでゲソ。今回はそんなことは気にしないことにするでゲソ

[*3] ちなみに Hackage のパッケージリストには載ってないでゲソが、理由は筆者の知識では分からないでゲソ

[*4] vector はランダムアクセスが定数時間で行えるデータ構造でゲソ。数値計算の人が使うのはだいたいランダムアクセスでゲソ

[*5] kmeans-vector ライブラリ作者調べ

[*6] kmeans-vector のライセンスは BSD3 に準じるでゲソ。章末に記載するでゲソ。

```
distance :: Vec -> Vec -> Double
distance u v = V.sum $ V.zipWith (\a b -> (a - b)^2) u v

partitionPoints :: Int -> [Vec] -> [[Vec]]
partitionPoints k vs = go vs
  where go vs = case L.splitAt n vs of
          (vs', []) -> [vs']
          (vs', vss) -> vs' : go vss
        n = (length vs + k - 1) `div` k

computeClusters :: [[Vec]] -> [Cluster]
computeClusters = zipWith Cluster [0..] . map f
  where f (x:xs) = let (n, v) = L.foldl' (\(k, s) v' ->
                     (k+1, V.zipWith (+) s v')) (1, x) xs
                   in V.map (\x -> x / (fromIntegral n)) v

regroupPoints :: [Cluster] -> [Vec] -> [[Vec]]
regroupPoints clusters points = go points
  where go points = map (map snd) . L.groupBy ((==) `on` fst) .
             L.sortBy (compare `on` fst) $ map (\p -> (closest p, p)) points
        closest p = cid $ L.minimumBy
             (compare `on` (distance p . center)) clusters

kmeansStep :: [Vec] -> [[Vec]] -> [[Vec]]
kmeansStep points pgroups = regroupPoints (computeClusters pgroups) points

kmeansAux :: [Vec] -> [[Vec]] -> [[Vec]]
kmeansAux points pgroups = let pss = kmeansStep points pgroups in
  case pss == pgroups of
  True -> pgroups
  False -> kmeansAux points pss

-- | Performs the k-means clustering algorithm
--   using trying to use 'k' clusters on the given list of points
kmeans :: Int -> [V.Vector Double] -> [[V.Vector Double]]
kmeans k points = kmeansAux points pgroups
  where pgroups = partitionPoints k points
```

説明をするでゲソ。K-Means は下記のアルゴリズムでゲソ。

1. 最初に、分けるクラスタの数 k を決めるでゲソ
2. 各データをランダムにクラスタに割り振るでゲソ
3. クラスタの中心を、そのクラスタに含まれる要素の座標の中心として求めるでゲソ
4. 各データを、最も近い中心のクラスタに再割り当てするでゲソ
5. 3 と 4 をクラスタの割り当てが変化しなくなるまで繰り返すでゲソ。割り当てが変化しなかったらクラスタリング完了でゲソ。

具体的には、先ほどのソースのの partitionPoints 関数が 2 に、computeClusters 関数が 3 に、regroupPoints 関数が 4 に、kmeansAux 関数が 5 の処理にあたるでゲソ。

熱心な読者は、数時間程度で K-Means を別のデータ構造で再実装できるであろう・・・でゲソ。

6.2　CSV をパーズしなイカ？

だいたいの場合、入力データは CSV で与えられるでゲソ。kmeans 関数に入れるためには Vector のリストにする必要があるんじゃなイカ？

というわけで、入力文字列をトークンに分解する関数 sepBy を定義するでゲソ。

```
sepBy :: Eq a => a -> [a] -> [[a]]
sepBy key str = sepBy' key str
  where
    sepBy' k [] = [[]]
    sepBy' k (c:cs)
      | k == c = [] : sepBy' k cs
      | otherwise =
        case sepBy' key cs of
          [] -> [[c]]
          (s:ss) -> (c:s):ss
```

この関数の適用結果はこんな感じでゲソ。

```
sepBy ',' "4.9,3.1,1.5,0.1" -- ["4.9" ,"3.1" ,"1.5" ,"0.1"]
```

6.3　実際にやってみるでゲソ

手始めに、あの海辺にある適当なデータを侵略するでゲソ。”UCI Machine Learning Repository”から、”Iris Dataset”をダウンロードするでゲソ。

```
wget http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
```

”Iris Dataset”はクラスタリングにおける標準的なデータセットでゲソ。アヤメ (Iris) を、花びらとがくのサイズから”Setosa”, ”Versicolor”, ”Virginica”の 3 種類に分類するでゲソ！

```
main = do
  cs <- readFile "iris.data" --ファイルを読んで文字列して
  let ls = lines cs          --改行で分けてリストにして
  --カンマで分けてリストのリストにするでゲソ
  let sepss = [seps | seps<-map (sepBy ',') ls, seps/=[""]]
  --分類ラベルを除去するでゲソ
  let xss = map (take 4) sepss
  --後で正解かどうかを確認するために、キー・値のマップでそれぞれのデータの正しいラベルを
  --記憶するでゲソ
  let irisMap = Map.fromList $
    map (\seps->(readVec $ take 4 seps, last seps)) sepss
  --データを Data.Vector にするでゲソ
  let vecs = readVecs xss
  --クラスタリングするでゲソ！ 引数 3 は望むクラスタの数でゲソ！！
  let cls = kmeans 3 vecs
  --正解数と全データ数でゲソ
  let (correct,all) = (correctNum cls irisMap, sum $ map length cls)
  putStrLn $ "Correct/All=" ++ show correct ++ "/" ++ show all
  putStrLn $ "Accuracy=" ++ show (fromIntegral correct / fromIntegral all)
  putStrLn "Putting clusters into iris.cls[0-2]"
```

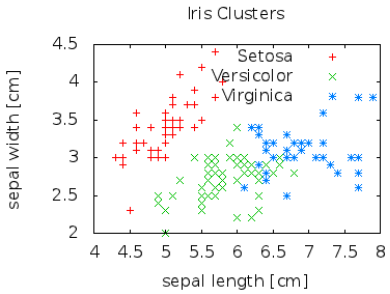
```
--結果をファイルに保存するでゲソ
mapM_ (writeCluster cls irisMap) [0..2]
```

それぞれの関数の詳細はソースコード[*7]を見ればいいんじゃないカ？
実行結果はイカのようになったでゲソ。

```
Correct/All=133/150          --正解数/全データ数
Accuracy=0.8866666666666667 --正解率（約 9 割）
Putting clusters into iris.cls[0-2]
```

Versicolor と Virginica の間でデータの交差があるので、正解率は 9 割程度でゲソ。
え、なんで３つのクラスタが”Setosa”, ”Versicolor”, ”Virginica”のどのラベルに対応しているかどうかわかるかって？ それは決め打ち・・・データ数が少ないときは手作業の誘惑には耐えられないもの・・・[*8]
無事データをクラスタリングできたので、データを表示する作業に入るでゲソ！ データのプロットには gnuplot を使うでゲソ。gnuplot の一行スクリプトを書くでゲソ！

```
gnuplot -e "set terminal png size 400, 300; set output 'iris.png';
 set title 'Iris Clusters';
 plot 'iris.cls0' using 1:2 with points title 'Setosa'...(略
```



という感じで、きれいにデータがクラスタに分けられたでゲソ！
上図のプロットでは４次元のデータ中２次元しか使ってないので一見外れ値があるように見えるでゲソが、実際は４次元空間の一番近いクラスタ中心の周りにデータが集まっているはずでゲソ。これにてクラスタリング完了でゲソ！
実行結果で示したように、正解率は 9 割弱でゲソ。つまり上のグラフの 1 割ぐらいは間違って・・・ゲフンゲフン！

6.4 まとめでゲソ

今回は手始めに Iris Dataset の分類を行い、クラスタリングの威力を見たでゲソ。今後はソーシャルグラフなどのさまざまな方向への侵略が期待されるでゲソ。
クラスタリングは、距離の計れる任意のデータに応用可能でゲソ。クラスタに対する意味づけは読者次第・・・なので無限の世界が広がっているでゲソ！

6.5 参考文献でゲソ

- 神嶌敏弘「クラスタリングとは」http://www.kamishima.net/jp/clustering/
- Wikipedia「K 平均法」http://ja.wikipedia.org/wiki/K 平均法
- Hackage - kmeans-vector http://hackage.haskell.org/package/kmeans-vector-0.1.1
- UCI Machine Learning Repository ”Iris Dataset”

[*7] http://github.com/dekosuke/iriscluster
[*8] 最も正解率が高くなるようにクラスタを各ラベルへ割り当てるためのアルゴリズムは、賢明な読者なら容易に思いつくであろう

http://archive.ics.uci.edu/ml/datasets/Iris

- 今回のクラスタリングのソース http://github.com/dekosuke/iriscluster

6.6 kmeans-vector の license でゲソ

```
Copyright (c)2011, Alp Mestanogullari <alpmestan@gmail.com>

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

    * Neither the name of Alp Mestanogullari <alpmestan@gmail.com>
      nor the names of other contributors may be used to endorse or
      promote products derived from this software without specific
      prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

第7章

Copilotへの希望と絶望の相転移

— @master_q

みんないつも参照透明な海を泳いでるでゲソ？ 海のめぐみは大きいでゲソ。素晴しいでゲソ。でもそのめぐみは大きな力によって保たれているでゲソ。例えば簡単な Haskell 製のプログラムを ghc でコンパイルしてみるでゲソ。

```
$ ldd HaskellApp
  linux-vdso.so.1 =>  (0x00007fff4ad7c000)
  libgmp.so.10 => /usr/lib/x86_64-linux-gnu/libgmp.so.10 (0x00007f250dd37000)
  libffi.so.5 => /usr/lib/x86_64-linux-gnu/libffi.so.5 (0x00007f250db2a000)
  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f250d8a7000)
  librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f250d69f000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f250d49b000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f250d116000)
  libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f250cefa000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f250dfd6000)
```

依存ライブラリが多いでゲソ、、、特に厳しいのが、libgmp,libc,libpthread あたりでゲソ。これらのライブラリに強く依存していてなかなかこれらのライブラリのない環境では参照透明な海のめぐみを受けられないでゲソ。。。

7.1 Copilot はイカが？

そんなことに悩んでいたおり

```
Copilot <http://leepike.github.com/Copilot/>
```

を使うと Haskell の DSL で書いたプログラムが C 言語に落ちるという情報を tanakh 殿から仕入れたでゲソ。 Copilot を使えば、Haskell 製プログラムから libgmp,libc,libpthread などへの依存をなくせて、左記ライブラリのない組み込み用途の CPU でも Haskell プログラミングできるんじゃなイカ？ ちょっと使ってみるでゲッソ! 使い方は Copilot tutorial [*1] が詳しいでゲソ。

Copilot のしくみを簡単に解説すると、Copilot はある種の DSL で、

1. Copilot 外部の変数や関数を呼び出して、Stream という型に収め
2. Stream をフィルタしたり Stream 同士を演算したりして
3. 条件によって外部へ副作用を与える (trigger)

ような繰り返しを表現するでゲソ。このようにして書かれた Copilot DSL は C 言語に変換できるでゲソ。この Copilot の繰り返しを進めるために、Copilot DSL 由来の C 言語とは別に main() 関数を含む C 言語ソースを準備して、その main() 関数内で Copilot の step() 関数を繰り返し呼び出すようにするでゲソ。変換された C 言語はせいぜい libm ぐらいにしか依存していない [*2]ので、組み込み環境でも容易使えるでゲソ。

Copilot のインストールは簡単でゲソ。 Debian sid amd64(2011/12/03 時点) では、以下のような感じでゲソ。

[*1] https://github.com/niswegmann/copilot-discussion/blob/master/tutorial/copilot_tutorial.pdf

[*2] Copilot DSL 側で sqrt 関数などを使うと C 言語側の sqrt() 関数が必要になるでゲソ

```
$ sudo apt-get install haskell-platform
$ cabal update
$ cabal install copilot
```

フィボナッチ数列の例があったので、そのまま写経してみたでゲソ。[*3] まずはフィボナッチ数列を表示してみるでゲソ。

- ソースコード: Fibonacci.hs

```
{-# LANGUAGE RebindableSyntax #-}
module Fibonacci where

import qualified Prelude as P
import Copilot.Language.Prelude
import Copilot.Language

fib :: Stream Word64
fib = [0, 1] ++ fib + drop 1 fib

fibSpec :: Spec
fibSpec = do
  trigger "fib_out" true [arg fib]
```

- ソースコード: Run.hs

```
module Main where

import Copilot.Language
import Fibonacci

main :: IO ()
main = interpret 50 [] fibSpec
```

- ソースコード: Compile.hs

```
module Main where

import Language.Copilot hiding (even, odd)
import Copilot.Language
import Copilot.Compile.C99
import Fibonacci

main :: IO ()
main = reify fibSpec >>= compile defaultParams
```

- ソースコード: main.c

[*3] https://github.com/master-q/study_copilot/tree/master/Fibonacci

```
#include <stdio.h>
#include <stdint.h>
#include "copilot.h"

void fib_out(uint64_t var)
{
  printf("(%llu)\n", var);
}

int main()
{
  int i;

  for (i = 0; i < 50; i++) {
      step();
  }
  return 0;
}
```

- ソースコード: Makefile

```
run:
  runghc Run.hs

copilot.c: Compile.hs Fibonacci.hs
  runghc Compile.hs

compile: copilot.o main.o
  gcc -o Fibonacci $^

clean:
  rm -f copilot.c copilot.h Fibonacci
  rm -f *~ *.o *.hi

.PHONY: run clean
```

これで準備ができたでゲソ。実行してみるでゲソ! 実行形式にはインタープリタ実行とコンパイル実行の 2 つがあるでゲソ。まずはインタープリタ実行を試すでゲソ。

```
$ make run
runghc Run.hs
fib_out:
(0)
(1)
(1)
(2)
(3)
(5)
......
```

うむ。ちゃんとフィボナッチ数列が表示されるでゲソ。 Run.hs で “interpret 50 [] fibSpec” としてフィボナッチ数列の先頭から 50 個を表示しているでゲソ。

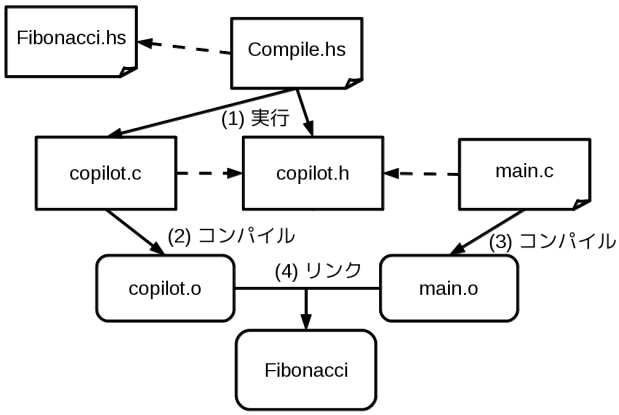
お次はコンパイル実行でゲソ。

```
$ make compile
runghc Compile.hs
Rule Scheduling Report
--snip--
cc    -c -o copilot.o copilot.c
cc    -c -o main.o main.c
gcc -o Fibonacci copilot.o main.o
$ ldd Fibonacci
        linux-vdso.so.1 =>  (0x00007fff3188d000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f696c273000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f696c628000)
$ ./Fibonacci
(0)
(1)
(1)
(2)
(3)
(5)
......
```

Haskell で書いたコードをコンパイルしたら libc にしか依存してないじゃなイカ! しかも実行結果はインタープリタ実行と変わらないでゲソ。これはいいじゃなイカ!

7.2　Copilot DSL と C 言語コードの融合でゲソ

ところで Copilot DSL をコンパイルして実行バイナリにするフローがわかりにくいのでここで解説しておくでゲソ。今回作った Fibonacci.hs のコンパイルは下図のようなプロセスでコンパイルできるでゲソ。



上図の実線はコンパイルによるファイル変換の流れ、破線はファイル間の include や import 関係でゲソ。ここで Fibonacci.hs のドライバになる main.c が必要になるでゲソ。このファイルの作り方は簡単でゲソ。

- main 関数を作りループで step 関数 (copilot.c に実体が作られる) を繰り返し呼び出す

- Fibonacci.hs から呼び出される関数、ここでは fib_out 関数をスタブとして作る

これだけでゲソ。main 関数から step 関数を while(1) で呼び出し続ければステートマシンのようなものも作れるでゲソ。あとは Fibonacci.hs 本体の書き方でゲソ。

7.3 Fibonacci.hs を例にして Copilot DSL の書き方を伝授するでゲソ!

Fibonacci.hs のソースコードを例に取って、Copilot DSL の書き方を伝授するでゲソ!

まず、"RebindableSyntax" というのは "if" のような既存の構文を書き換え可能にする拡張でゲソ。[*4] これがないと Copilot の DSL が使えないでゲソ。fib 関数の定義で出てくる "Stream" という型は Copilot のキモで、Haskell のリストに似た何かでゲソ。上記 Fibonacci.hs の fib 関数を Haskell のリストを使った同等な fib_ll 関数に書き直すと以下のようになるでゲソ。

```
fib_ll :: [Word64]
fib_ll = [0, 1] ++ zipWith (+) fib_ll (drop 1 fib_ll)
```

fib と fib_ll の違いがわかるでゲソ？ フィボナッチ数列は $f_{n+2} = f_n + f_{n+1}$ なので、1 つ要素のずれたリストを 2 つ作り、それぞれのリストの要素を足しこめばいいでゲソ。ところが Haskell のリストだとそのまま (+) 演算子で要素を足しこむことはできないので、zipWith を使って (+) をリスト同士の演算に持ち上げるでゲソ。

```
Prelude> :t zipWith (+)
zipWith (+) :: Num c => [c] -> [c] -> [c]
```

ところが Stream はそのようなコンテナではないでゲソ。Stream 同士で基本的な演算ができるでゲソ! Stream 同士の (+) は定義されていているでゲッソ。

```
instance (Typed a, Num a) => Num (Stream a) where
  (Const x) + (Const y)   = Const (x + y)
  (Const 0) + y           = y
  x + (Const 0)           = x
  x + y                   = Op2 (Core.Add typeOf) x y
```

で、結局 Stream というのはなんなんでゲソ？

```
*Copilot.Language.Stream> :i Stream
data Stream $a where
  Append :: Typed a =>
            [a] -> (Maybe (Stream Bool)) -> (Stream a) -> Stream a
  Const :: Typed a => a -> Stream a
  Drop :: Typed a => Int -> (Stream a) -> Stream a
  Extern :: Typed a => String -> Stream a
  ExternFun :: Typed a => String -> [FunArg] -> Stream a
  ExternArray :: (Typed a, Typed b, Integral a) =>
                 String -> (Stream a) -> Stream b
  Local :: (Typed a, Typed b) =>
           (Stream a) -> (Stream a -> Stream b) -> Stream b
  Var :: Typed a => String -> Stream a
  Op1 :: (Typed a, Typed b) =>
         (Core.Op1 a b) -> (Stream a) -> Stream b
  Op2 :: (Typed a, Typed b, Typed c) =>
```

[*4] http://www.sampou.org/cgi-bin/w3ml.cgi/haskell-jp/msg/595

```
        (Core.Op2 a b c) -> (Stream a) -> (Stream b) -> Stream c
Op3 :: (Typed a, Typed b, Typed c, Typed d) =>
        (Core.Op3 a b c d) -> (Stream a) -> (Stream b) -> (Stream
                                                          c) -> Stream d
```

なるほどでゲソ。Stream 同士を計算すると Op2 のような変換を経由してまた Stream になるんじゃなイカ。全部 Stream にまとめてしまえば C 言語に落とすのが楽でゲソ。

残る疑問は Spec 型でゲソ。これは trigger 関数を列挙するためのモナドでゲソ。trigger 関数は下記ような型で、、、

```
trigger :: String -> Stream Bool -> [TriggerArg] -> Spec
```

1. String: step 毎に呼び出す C 言語関数の名前
2. Stream Bool: 指定した C 言語の関数を実行するかどうか
3. [TriggerArg]: 指定した C 言語への引数リスト

のような引数の意味を持つでゲソ。Fibonacci.hs での trigger 関数は以下のような意味になるでゲソ。

1. “fib_out”: fib_out 関数を
2. true: どの step でも常に
3. [arg fib]: 当該 step での fib Stream の要素を第一引数にして呼び出す

Spec はモナドなので trigger を列挙できるんでゲソが、列挙された trigger がどの順序で実行されるかは列挙順とは限らないことが注意でゲソ。順序制御をしたければ次の step で trigger が起動されるように Stream Bool を作る必要があるでゲソ。

7.4　Wii リモコンで遊ぶでゲソ

だいたい使い方が解ったところで何かアプリケーションを書いてみるでゲソ。どんな題材にしようでゲソ。Stream を使った情報の加工をしたいので何かセンサーがたくさん付いたデバイスの制御とか、、、ちょうど手元にあるので Wii リモコン [*5] にしなイカ？ まずは実行例で作成した Wii リモコンアプリの使い方を説明するでゲソ。

```
$ uname -a
Linux casper 3.1.0-1-amd64 #1 SMP Tue Nov 29 13:47:12 UTC 2011 x86_64 GNU/Linux
$ ./Main
Put Wiimote in discoverable mode now (press 1+2)...
found!
|・ω・｀）イザ...
(｀・ω・´ ) ドギャ! (5.042165)
(｀・ω・´ ) ドギャ! (5.069622)
|彡サッ！
|・ω・｀）イザ...
|彡サッ！
```

B ボタンを押しながら Wii リモコンを強く振ると “ドギャ! (振った強さ)” というメッセージがコンソールに表示されるでゲソ。B ボタンを押していない状態でいくら Wii リモコンを振っても何も表示されないで注意でゲソ。

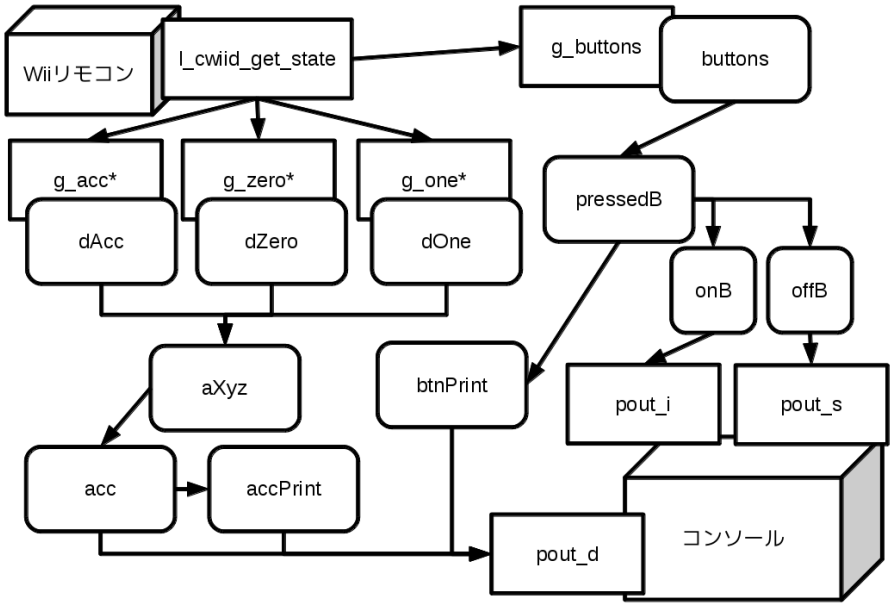
このアプリケーションのソースコードを全部載せるとページ数が無駄になってしまうので、Copilot DSL の部分 (Wiimote.hs) だけ掲載/解説するでゲソ。

[*5] CWiid <http://abstrakraft.org/cwiid/>

```
{-# LANGUAGE RebindableSyntax #-}
module Wiimote where

import qualified Prelude as P
import Copilot.Language.Prelude
import Copilot.Language

type StreamTdouble = (Stream Double, Stream Double, Stream Double)

inited :: Stream Word32
inited = [0,1,2] ++ 3

buttons :: Stream Word16
buttons = externW16 "g_buttons"

pressedB :: Stream Bool
pressedB = (buttons .&. 4) /= 0

onB :: Stream Bool
onB = n && not p
  where n = pressedB
        p = [False] ++ n

offB :: Stream Bool
offB = not n && p
  where n = pressedB
        p = [False] ++ n

funCastW8 :: String -> Stream Double
funCastW8 s = externFun "cast_double" [funArg $ externW8 s]

dAcc, dZero, dOne :: StreamTdouble
dAcc = (funCastW8 "g_acc0", funCastW8 "g_acc1", funCastW8 "g_acc2")
dZero = (funCastW8 "g_zero0", funCastW8 "g_zero1", funCastW8 "g_zero2")
dOne = (funCastW8 "g_one0", funCastW8 "g_one1", funCastW8 "g_one2")

aXyz :: StreamTdouble -> StreamTdouble -> StreamTdouble -> StreamTdouble
aXyz (a1,a2,a3) (z1,z2,z3) (o1,o2,o3) =
  ((a1 - z1) / (o1 - z1), (a2 - z2) / (o2 - z2), (a3 - z3) / (o3 - z3))

acc :: StreamTdouble -> Stream Double
acc (ax, ay, az) = sqrt $ ax ** 2 + ay ** 2 + az ** 2

accPrint :: Stream Bool
accPrint = foldl1 (&&) $ fmap (> 5.0) [accB0, accB1, accB2]
  where accB0 = acc $ aXyz dAcc dZero dOne
        accB1 = [0] ++ accB0
        accB2 = [0] ++ accB1
```

```
btnPrint:: Stream Bool
btnPrint = pressedB && (b1 && b2)
  where b1 = [False] ++ pressedB
        b2 = [False] ++ b1

wiiSpec :: Spec
wiiSpec = do
  trigger "l_cwiid_open" (inited == 0) []
  trigger "l_cwiid_get_acc_cal" (inited == 1) []
  trigger "l_cwiid_set_rpt_mode" (inited == 2) []
  trigger "l_cwiid_get_state" (inited == 3) []
  trigger "pout_d" (accPrint && btnPrint) [arg $ acc $ aXyz dAcc dZero dOne]
  trigger "pout_i" onB []
  trigger "pout_s" offB []
```

アプリケーション全体のソースコードは github [*6] に上げてあるので実行してみたい方は参照でゲソ。で、肝心の Wiimote.hs の中身の解説でゲソが、ちょっとややこしいので図で説明するでゲッソ。



まず上の図に示されていないデバイスの初期化について解説するでゲソ。0,1,2,3,3,3,3,3,.... のような inited Stream を準備しておくでゲソ。この inited Stream を使って trigger を書けば以下のように初期化関数を順番に呼び出した後、Wii リモコンの状態取得を行なう l_cwiid_get_state 関数を定期的に実行し続けるでゲソ。

- step 1 回目: l_cwiid_open 関数を実行
- step 2 回目: l_cwiid_get_acc_cal 関数を実行
- step 3 回目: l_cwiid_set_rpt_mode 関数を実行
- step 4 回目以降: l_cwiid_get_state 関数を実行

これで上の図の説明のスタートに辿り着いたでゲソ。図中の四角は C 言語の関数や変数、丸四角は Copilot DSL の Stream でゲソ。データの流れとしては、Wii リモコンから加速度などの情報をポーリングで取り出

[*6] https://github.com/master-q/study_copilot/tree/master/Wiimote

し、Copilot DSL で加工して、最終的にコンソールにプリントされるでゲソ。図の Stream の接続を順番に説明するでゲソ。

1. l_cwiid_get_state 関数は定期的に呼び出されて Wii リモコンの状態を以下のグローバル変数に格納するでゲッソ
   - g_buttons: ボタンの状態
   - g_acc*: 加速度情報
   - g_zero*: 加速度ゼロ点情報
   - g_one*: 加速度初期値情報
2. 上記のグローバル変数は externFun 関数でラップされてそれぞれ以下の Stream に変換されるでゲソ
   - g_buttons → buttons
   - g_acc* → dAcc
   - g_zero* → dZero
   - g_one* → dOne
3. 加速度情報: dAcc,dZero,dOne はキャリブレーションの関数である aXyz を通って一つの Stream にまとめられるでゲソ
4. 加速度情報: aXyz を通った Stream はタプルなので、さらに acc 関数を通って xyz 軸を平均化、Stream Double 型に変換されるでゲッソ
5. 加速度情報: acc 関数を通った加速度 Stream は accPrint 関数によって加速度の強度が step 3 回連続で 5.0 以上かチェックされるでゲソ
6. ボタン情報: buttons Stream には全てのボタンの押下状態が入っているのでボタン B のみの押下状態 pressedB Stream を作るでゲソ
7. ボタン情報: B ボタンが OFF → ON になったことを検知する onB Stream を pressedB Stream から作るでゲソ
8. ボタン情報: B ボタンが ON → OFF になったことを検知する offB Stream を pressedB Stream から作るでゲソ
9. ボタン情報: B ボタンが step 3 回連続で ON であったことを検出する btnPrint Stream を pressedB Stream から作るでゲッソ!
10. もし Wii リモコンを強く振っていて (accPrint) なおかつ B ボタンを押下していた (btnPrint) ら現在の加速度の xyz 平均値 (acc $ aXyz dAcc dZero dOne) を表示するでゲソ
11. もし B ボタンが OFF → ON になったことを検知した (onB) ら “|・ω・｀) イザ…” と表示するでゲソ
12. もし B ボタンが ON → OFF になったことを検知した (offB) ら “| 彡！ ” と表示するでゲソ

Copilot DSL の表現力は trigger では得られないので、必然的に Stream を重ねることでプログラミングをするんでゲソ。

7.5 libc のないところでも操縦できるでゲッソ!

さぁそろそろお遊びの時間は終わりでゲソ。当初の目的を覚えているでゲソか？「libgmp,libc,libpthread のようなライブラリのない環境でも参照透明な海を泳ぎたい」だったでゲソ。願ったからには作らなければならないでゲッソ! 対象として STM32 のような組み込み用途の CPU 上で動かしても良かったでゲソが、色々な制約上 [*7] NetBSD の bootloader 上で Copilot DSL を使ったプログラミングをしてみるでゲソ。。。。ということでなんとかできたゲソ。 [*8] NetBSD の bootloader は src/sys/arch/i386/stand/boot にソースコードが置かれていて、その中の boot2.c にある boot2() 関数が bootloader の main 関数でゲソ。この boot2() 関数の中に Copilot を呼び出す step 関数をループで書けば、 Copilot DSL を bootloader 上で使えるでゲソ。 [*9]

まずは今回作った Copilot DSL 内蔵 bootloader の使い方でゲソ。ビルド手順は少しややこしいので、github の README [*10] を読んで欲しいでゲソ。環境を整えたら qemu.sh で Copilot な bootloader が起動するでゲソ。

[*7] 単に筆者の根性がないだけでゲソ……
[*8] https://github.com/master-q/study_copilot/tree/master/NetbsdBoot
[*9] カーネル／ VM Advent Calendar 7 日目 http://d.masterq.net/?date=20111207#p01
[*10] https://github.com/master-q/study_copilot/blob/master/NetbsdBoot/README.md

```
>> NetBSD/x86 BIOS Boot, Revision 5.3 (from NetBSD 5.1)
>> for Book:Lambda Ka Musume
(1)
(2)
l(3)
z(4)
d(5)
x(6)
r(0)
r(0)
s(1)
s(2)
d(3)
l(4)
```

bootloader が起動するとバナーが表示されてキーボード入力待ちになるでゲッソ。てきとーにキーボードから文字を入力すると入力文字とカウンタが表示されるでゲソ。“r” キーだけは特別で、2 回 “r” キーを入力するとカウンタがリセットされるでゲソ。[*11]

今回も Copilot DSL だけソースコード (Bootmenu.hs) を掲載/解説するでゲソ。

```
{-# LANGUAGE RebindableSyntax #-}
module Main where

import qualified Prelude as P
import Language.Copilot hiding (even, odd)
import Copilot.Compile.C99

counter :: Stream Bool -> Stream Word64
counter reset = y
  where zy = [0] ++ y
        y  = if reset then 0 else zy + 1

lGetChar :: Stream Int32
lGetChar = externFun "l_getchar" []

lPrintable :: Stream Bool
lPrintable = (lGetChar >= 32) && (lGetChar < 127)

resetCounter :: Stream Bool
resetCounter = lGetChar == 114

menuSpec :: Spec
menuSpec = do
  trigger "l_putchar" lPrintable [arg lGetChar]
  trigger "counter_out" true [arg $ counter resetCounter]

main :: IO ()
```

[*11] ｡｡｡ え｡｡｡ それだけ？？？ 副操縦士さんに疲れたんでゲソ｡｡｡ #orz

```
main = reify menuSpec >>= compile defaultParams
```

DSL 側自体は Wii リモコンの時より簡単でゲソ。簡単に解説すると、、、counter Stream は step 毎に +1 される Stream でゲソ。 lGetChar Stream は BIOS コールを呼び出す l_getchar 関数を呼び出して 1 文字キーボード入力を受けるでゲソ。lPrintable はその lGetChar Stream が印字可能文字かチェックするだけで、resetCounter は lGetChar で受けた文字が “r” の時に True になるでゲソ。l_putchar trigger で入力文字が印字可能なら画面にプリントするでゲソ。counter_out trigger は step 毎に現在のカウンタの内容を表示するでゲソ。というかソースコードを読んだ方が自然言語よりわかりやすいでゲソ。参照透明の力でゲソ! [*12]

あれ？ ところでさっきの bootloader の実行結果がおかしいじゃなイカ？ なぜ “(1)” が画面に表示された際にキーボード入力文字は表示されてないんでゲソ？ そもそも “(1)” が表示されるまでに何度かキーボードをたたいた覚えがあるでゲソ。凍ってるのかと思えば動いている。。。これは何でゲソ？ 気になるでゲソ。

bootloader 上で動かしていた Copilot DSL を libc の上で動かしてみるでゲソ。以下のような main.c でキーボード入力はてきとうなスタブにしといたでゲソ。

```
#include <stdio.h>
#include <stdint.h>
#include "copilot.h"

void pout_w64(uint64_t var)
{
  printf("(%llu)\n", var);
}

int32_t l_getchar(void)
{
  static int32_t ret = 'A';
  return ret++;
}

void l_putchar(int32_t c)
{
  putchar((int) c);
}

void counter_out(uint64_t var)
{
  printf("(%llu)\n", var);
}

int main()
{
  int i;

  printf(">> Fakeboot for Book:Lambda Ka Musume\n");
  for (i = 0; i < 10; i++) {
      step();
  }
  printf("\n");
```

[*12] たぶんそうではないですね。やってる内容が簡単なだけです。。。#orz

```
  return 0;
}
```

さっそく実行でゲソ。

```
$ make -f Makefile.onlibc
$ ./Bootmenu
>> Fakeboot for Book:Lambda Ka Musume
(1)
(2)
E(3)
J(4)
O(5)
T(6)
Y(7)
^(8)
c(9)
h(10)
m
```

うむ？ ツッコミどころは色々あるでゲソが、キーボード入力は “ABCDEFG…” のような ASCII コード順の Stream になっているはずでゲソ。ところが “EJOTY…” しか表示されていないでゲソ! ASCII 表だと

- ‘A’ → 65
- ‘E’ → 69
- ‘J’ → 74
- ‘O’ → 79
- ‘T’ → 84

のように対応しているはずなので、 Copilot が生成した copilot.c の lGetChar Stream にはサイズ 5 の内部バッファがあって、そのバッファが満たないと文字が出てこないように推測できるでゲソ。Copilot DSL から生成された copilot.c を見てみるでゲソ。

```
static void __r0() {
  bool __0 = true;
  int32_t __1 = tmp_ext_fun_l_getchar__0;
  if (__0) {
    tmp_ext_fun_l_getchar__0 = l_getchar();
    __coverage[0] = __coverage[0] | (1 << 0);
  }
  state.copilot.ext_fun_0_l_getchar = __1;
}
--snip--
void copilot()
{

  {
    static uint8_t __scheduling_clock = 2;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r0();  /* copilot.sample_fun_l_getchar_0 */
      __assertion_checks(); __r1();  /* copilot.sample_fun_l_getchar_1 */
```

```
      __assertion_checks(); __r2();  /* copilot.sample_fun_l_getchar_2 */
      __assertion_checks(); __r3();  /* copilot.sample_fun_l_getchar_3 */
      __assertion_checks(); __r4();  /* copilot.sample_fun_l_getchar_4 */
      __scheduling_clock = 6;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
--snip--
void step()
{
  copilot();copilot();copilot();copilot();copilot();copilot();copilot();
}
```

こ、これは何でゲソ!一回の step であるにもかかわらず、copilot 関数の中で__r0 関数を通じて 5 回も l_getchar() が呼び出されているでゲソ! これはヒドいじゃなイカ？

Copilot の DSL は汎用的なアプリケーションを書くために作られた訳ではないんでゲソ。マイコンに接続されたセンサー類を観測しながら、なんらかのアクションを起こすようなタスクマシンを設計するには非常に適しているでゲソ。センサー類が入力なのであれば、読み出し回数が複数回になってもなんの問題もないでゲソ。さらに、もっと汎用的なものを作るには Stream だけでプログラミングするのは非力すぎるゲソ。。。

7.6 けっきょく使えそうなんでゲソ？

Copilot を製品開発には採用できそうもないことがわかってしまったでゲソ。ここから教訓が得られるのではなイカ？

問題の本質は Copilot DSL が Haskell のような汎用プログラミング言語そのものと比較してあまりにも非力であるということでゲソ。DSL というものは限定用途のためにのみ設計されていて、その用途から外れてしまうと使いづらくなるのではなイカ？ どのような用途もある程度満たすような何かは DSL ではなく、結局汎用プログラミング言語と同じようなものになってしまうのかもしれないでゲソ。もしそうなら解決策は DSL ではなく Haskell 言語で記述されたコードをそのまま libc のない空間で動かすこと、それって結局 GHC が libc に頼らないコードも掃き出せるようにしなきゃいけないってことじゃなイカ？ ！どうして GHC には組み込み用途のコンパイル機能がないんでゲソ？

実はあるのでゲソ。かるく検索するだけでも以下のような試みが見つかるでゲソ。

- Lighthouse `http://web.cecs.pdx.edu/~kennyg/house/`
- HaLVM `http://halvm.org/wiki/`
- Kernel Modules `http://www.haskell.org/haskellwiki/Kernel_Modules`

ところがどのプロジェクトも GHC 本体にはマージされていないでゲソ。しかも GHC のバージョン 6 時代のままじゃなイカ。理由があるにしても、やる気がないんじゃなイカ？ [*13]

これはきっと POSIX のインターフェイスが悪いんでゲソ。POSIX はコタツのようなものなんでゲソ。いったん使いはじめるたらコタツの中で生活することの方がたやすい。それではいつまでたっても libc 以下のレイヤーは汚れた海のままでゲソ! 悪の根源がたった今わかったでゲソ。

「全ての POSIX 互換インターフェイスを生まれる前に消し去りたい。すべての宇宙、過去と未来の全ての POSIX 互換インターフェイスを、この手で」
「そんな祈りが叶うとすれば、それはインターフェイス変更なんてレベルじゃない。C 言語そのものに対する反逆だ」

[*13] GHC 本体と統合するのはいろいろ難しいみたいです。。。

「希望を信じたプログラマを、私は泣かせたくない。最後まで笑顔でいて欲しい。それを邪魔する C 言語なんて、壊して見せる。変えて見せる」

ということで Lighthouse プロジェクトで作られた GHC6 への変更内容を検証しなイカ？ (次回に続く...λ)

第8章

加速しなイカ？！

— @nushio

8.1 これまでのあらすじ

■夢のようなライブラリがあったじゃなイカ！ これからは並列計算の時代だと聞いて闇の言語侵略に闘志を燃やす関数型イカ娘。そこにキュゥべえアイコンの人が現れて、避けようのない綻びも驚きもすべて君が抽象化してしまえば良い、そのための力が Haskell には備わっているし、彼女も出来ると説くのだった。

■GPU も、FPGA も、あるでゲソ！ イカ娘は Accelerate をつかった GPGPU コード生成を試してみる。Accelerate ちゃんは、これからの並列計算のコードを生成しまくっちゃいますからねーと意気込むのであった。

■実アプリの性能と向きあわなイカ？ 緑色っぽい CUDA 使いに性能の差を見せつけられた Accelerate ちゃんは、存在意義を見失い、その魂を黒く黒く染め上げていくのだった。

■そんなの、私が許さないでゲソ！ イカ娘は涙ながらに、自分自身の手でちゃんと速度の出るコードを生成してみせることを誓うのだった。いいよ、Accelerate ちゃんのためなら、私・・・。

■最後に残った半蔵門線でゲソ！ ついに東京をワルプルギスの夜が襲う。台風により交通網が麻痺する中、イカ娘たちは ICFP 会場から PFI 社までサイモン先生を護送する必要に迫られていた。

— そして、現状はというと・・・

8.2 進化しなイカ？

「フゥーハハハハハハハ！！ 数々の未来ガジェットを生み出してきた我がラボの最新作がついに完成したでゲソ！」

— イカちゃんは時間遡行者の真似をしている。個人的には、そっちじゃないほうの時間遡行者が好みなのだが。

「ぼやいたって仕方がないでゲソ。さあ、本題に行かなイカ？ この章では、拙作の偏微分方程式の陽的解法コードを GPU その他並列計算機向けに生成するためのドメイン特化言語、Paraiso(`http://paraiso-lang.org/wiki/`) を扱うでゲソ。Paraiso の実装の詳細については、函数プログラミングの集い 2011 などでも紹介したから説明を省略するとして、今回はそこからのアップデートをご紹介するでゲソ。なんと、Paraiso には遺伝的アルゴリズムベースの自動チューニング機構が備わったのでゲソ。すごいじゃなイカ！ これで、すべての偏微分方程式の陽解法を速くできるでゲソ。過去と未来、すべてのアーキテクチャで、すべてのコードを、この手で！」

— と言いたいところだが、今のところ対応しているのは OpenMP もしくは CUDA のみなのは相変わらずだ。それにしても、Paraiso の詳細を完全に飛ばしたら、さすがに以後の説明に差し障るのじゃなイカ。三行くらいでまとめられなイカ？

「だるいでゲソ！ それに三行じゃとても足りないでゲソ！」

「私からもお願いできるかしら」

「Orthotope Machine は、任意のサイズの配列変数を確保し、並列に操作する命令を持った、偏微分方程式の陽解法を表現するための仮想マシンです。」

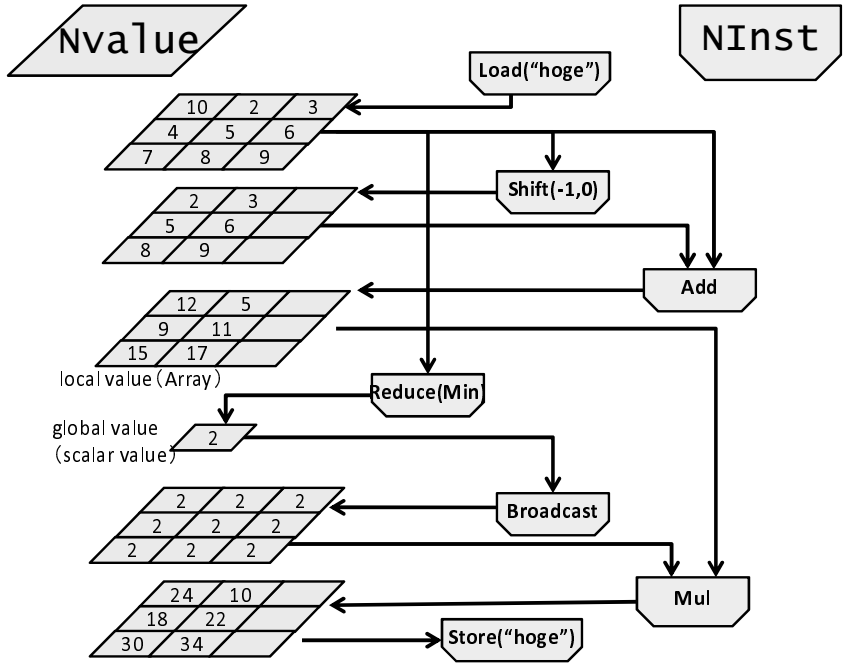


図 8.1: Orthotope Machine(OM) のデータフローグラフ。OM には複数の Kernel がある。Kernel とは一塊の計算 (ふつうにメソッドみたいなもん) であり、ひとつの Kernel がひとつのデータフローグラフに対応する。OM の変数には、Kernel の中にスコープが限られる Temporal 変数と、名前がついていて Kernel 間を生き延びる Static 変数とがあり、各 Kernel の目的は Static 変数を Load 命令で読み込んで、Store 命令でその新しい状態を書き出すことである。OM の他の命令の意味は、イメージしろ！

「行いたい計算を表現するデータフローグラフを、Builder モナドというフロントエンドを使って、Orthotope Machine という仮想機械の命令列として生成します。(c.f. 図 8.1)」

「データフローグラフを適切なところで分割して、CUDA カーネル呼び出しまたは C の関数呼び出しの集まりに変換しコードを生成します。」

— おお、ちゃんと三行で説明できた。でも、データフローグラフを**適切**に分割って、簡単に言うけど、具体的には何をすればいいのだろう。

「まさにそこがポイントなのでゲソ！ 図 8.2 のような簡単なプログラムをとってもわかるように、計算の途中結果を配列としてメモリに置く (Manifest) か、必要に応じて再計算する (Delayed) か、実装の選択肢があるじゃなイカ？ このへんは Haskell の多次元配列ライブラリ、REPA [4] を参考にしたでゲソ。」

— 図 8.3 のように、この Manifest or Delayed の選択はグラフの分割結果、そして生成されるコードを大きく左右する。これが Paraiso が自動チューンしてくれるポイントの一つだ。

「もう一つは、各演算命令の前に同期命令`__syncthreads()`を挿入するかどうかでゲソ。メモリの読み書きの前など、同期を挟めば次の命令が揃うから高速化することがあるでゲソが、あまり同期を頻繁にすると性能の低下につながるでゲソ。

さらに、CUDA コードを生成する場合はカーネルトポロジー (GPU 内スレッドの分割数) も最適化してくれるでゲソ。全部合わせて、今回の実験では遺伝子は 6000 ビット強のサイズになっているでゲソ。可能な実装の数、実に 2^{6000}！ Paraiso には、一旦組み立てたプログラムからこれらのチューニングポイントをシリアライズして遺伝子に書き出す機能や、遺伝子をプログラムに合成する機能が備わっているでゲソ。これにより、遺伝子からプログラムを生成してはベンチマークをとり、そのベンチマークをスコアとして遺伝子を進

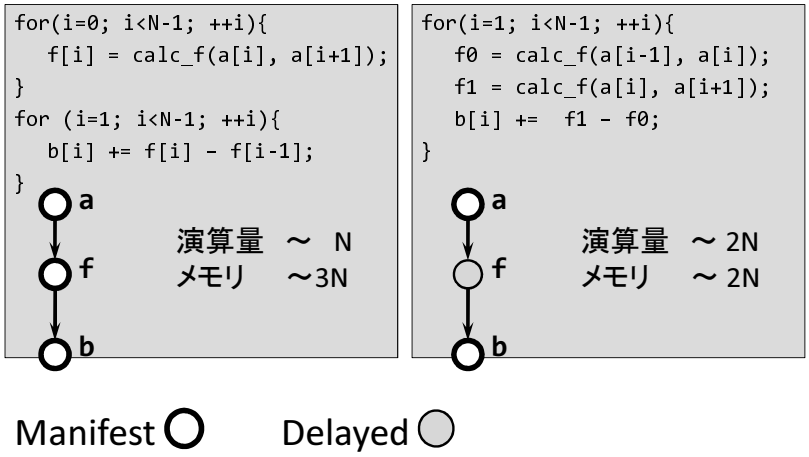


図 8.2: メモリを使ってでも演算量を減らすか、演算量を増やしてでもメモリの容量や帯域を節約するか？ Manifest or Delayed トレードオフ。

化させることが可能になるのでゲソ。」

— このようなビッグサイズな探索空間の中から、なるべく良い候補を探したいとなると、たしかに遺伝的アルゴリズム (GA) の出番だろう。とつぶやいた所、

「え、焼きなまし最強じゃなイカ？」

「 GA が焼きなましよりもうまくいったことが無いでゲソ」

「『GA? 結構うまくいくよ』派と『GA? 全然うまくいかないよ (><)』派の断絶がある稀ガス」

— などの反応が帰ってきた。持つべきものは友である。良い論文を書く手助けをしてくれるからだ！ 実は、今回使っているのは焼きなましと遺伝的アルゴリズムを結合したオリジナルのレシピなのだ。[*1]

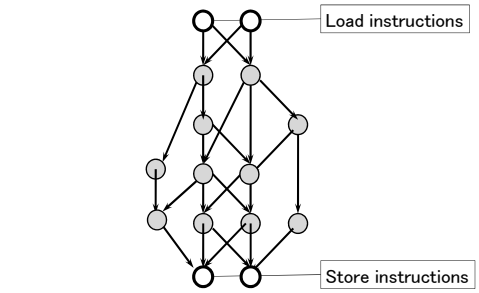
8.3 生存戦略しなイカ？

「通常の焼きなまし法はチューニング対象を一個しか保持しなイカら、解の候補を複数持って競わせる遺伝的アルゴリズムより省スペースで高速だし、焼きなましスケジュールにしたがって温度を下げていけば、かならず終了するというのが利点でゲソ。しかし、これは裏を返せば並列化に向かないし、焼きなましスケジュールの調整に失敗すると局所解に陥りやすいという欠点にならなイカ？

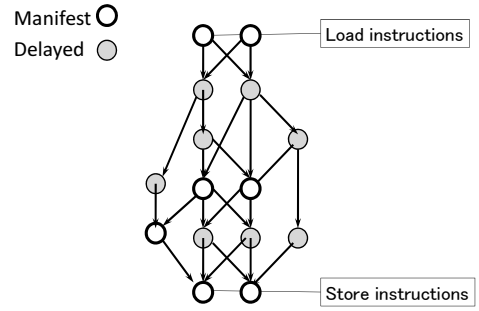
これらの欠点を解決したのが replica-exchange Monte Carlo 法 [3] でゲソ。この手法は、通常の焼きなまし法では一つの熱浴を用意し温度を下げてゆくところ、あらかじめ熱湯から氷水まで幅広い温度の熱浴を並列に用意し、隣同士で解の交換を許すという手法でゲソ。これも時間を空間に TM のひとつじゃなイカ？ 焼きなましスケジュールの候補のリスト `[Time -> Temperature]` を、`Time -> [Temperature]` という一つ上の概念にシフトし、かつてあったスケジュールも、いつかあり得るかもしれないスケジュールも、みんな同時に実現しているのでゲソ！」

— まだ問題がある。我々はクラスター計算機の空きノードで自動チューニングを行うことを強いられているんだ！ 通常の連結熱浴法は、熱浴の並列数が固定で、一世代ごとに同期を取る。これに対し「遺伝子」の

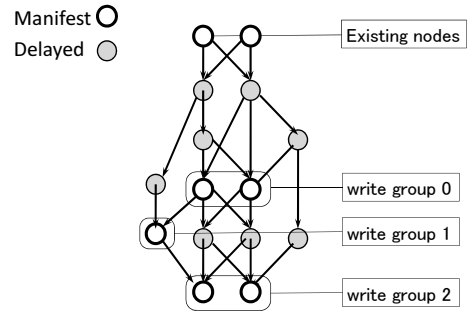
[*1] こんなのを論文より先に同人誌で出しちゃっていいのだろうか・・・だが、それがいい。



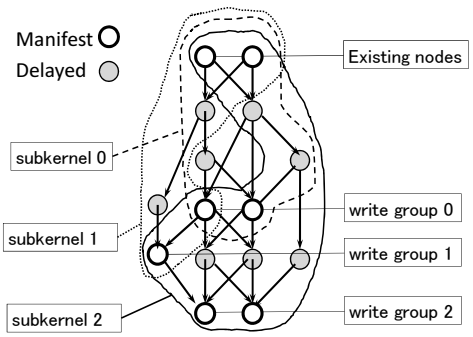
(1) Kernel には初期状態で Manifest にすべきノードが存在する。



(2) データフローグラフの残りのノードについて、Manifest or Delayed を自由に選択する。



(3) ひとつのループで一括処理できる演算を greedy に結合し、write group を作る。



(4) 各 write group への書き込みを担当する subKernel を作ることで、Kernel を複数の subKernel に分割する。この後、一つの subKernel が一つの`__global__`関数もしくは C の関数に対応したコードに変換される。

図 8.3: Paraiso のコード生成メカニズム。

スコアは生成したコードのベンチマーク。スコアが出るまでの所要時間は遺伝子によって違うから、同期をとっていては待ち時間が生じる。おまけに空きノードの数も刻々変化するわけで、熱浴の並列数が固定というのも悩ましい。何とかして、個々の遺伝子の評価を非同期処理にできなイカ？

「そこで我々は、一種のマスター／ワーカーモデルに基づく自動チューニング機構を新たに開発したのでゲソ！ ちなみにマスターは ruby で書いて、Paraiso によるコード生成からジョブの投入までを制御しているでゲソ。適材適所じゃなイカ！

マスターは過去に出現した全ての個体の遺伝子やそのスコアをデータベース (DB) に持っているでゲソ。マスターの役目は、DB からいくつかの遺伝子を取り出し、組み合わせて新個体を作り出してはワーカーを作って実行させることでゲソ。これに対しワーカーの役目は、与えられた遺伝子をもとにコードを生成し、ベンチマークの統計を取って DB に書き込むことでゲソ。」

— なるほど。これなら、DB さえきっちり同期が取れれば、ワーカーは非同期に実行できるし、ワーカーの数が変動しても構わない。しかも、旧世代の情報は上書きされてアクセスできなくなるという、焼きなまし法・遺伝的アルゴリズムに共通する欠点も消えているじゃなイカ。

「ひとつの個体 S についてベンチマークは 30 回程度走らせることになっているでゲソ。こうして得られたスコアの平均 $\mu(S)$ と標準偏差 $\sigma(S)$ を DB に記録しているでゲソ。」

— 標準偏差を記録しておくのって、そんなにも大切な事なの？

「十分な並列度があれば、出し惜しみせずに統計を溜めることだってできる。それが並列焼きなましの強みでゲソ。」

「実際に、平均と標準偏差をどう使っているか見れば分かるんじゃなイカ？ DB から遺伝子を一つランダムに取りだす操作が『ドロー』。本手法において焼きなましの肝となる操作でゲソ。ドローには温度 T があるでゲソ。温度が T のドローでは、個体 S の遺伝子が選ばれる確率は

$$\exp\left(\frac{\mu(S_0)-\mu(S)+\sigma(S_0)+\sigma(S)}{T+\sigma(S_0)+\sigma(S)}\right)$$

に比例するのでゲソ。ここで S_0 は現時点で成績がトップの個体でゲソ。」

— 高温のドローは、スコアに余り頓着せず幅広い個体から遺伝子を選択するし、低温のドローほど、スコアが高い個体だけを好むというわけだ。そして、$\sigma(S_0)+\sigma(S)$ と比べて細かな違いは、その影響が小さくなるようになっている。

標準偏差を記録せず、一発録りのスコアだけで個体を評価していると、たまたま良い成績が出たやつが優良個体だと錯覚されて拡まってしまうことがある。そうなると、それ以上のスコアの個体が出るまで何度でも遺伝子をドローし、墓地に捨てるハメになってしまうのだ。

「ここで、マスターは新個体をつくるたびドロー温度 T をランダムに変えることで、replica-exchange Monte Carlo 法と同様、温度分布のある熱浴群を実現するでゲソ。

このように、焼きなまし法を並列化した replica-exchange Monte Carlo 法を、マスター／ワーカーモデルにより非同期化して動的なリソース変動に対応、さらに近傍生成手段として遺伝的アルゴリズムを採用することで、焼きなまし法の『大局最適化が見つかりやすい』という利点と、遺伝的アルゴリズムの『独立に発見した改善どうしを合成できる』という利点をうまくイカした手法が本手法なのでゲソ！」

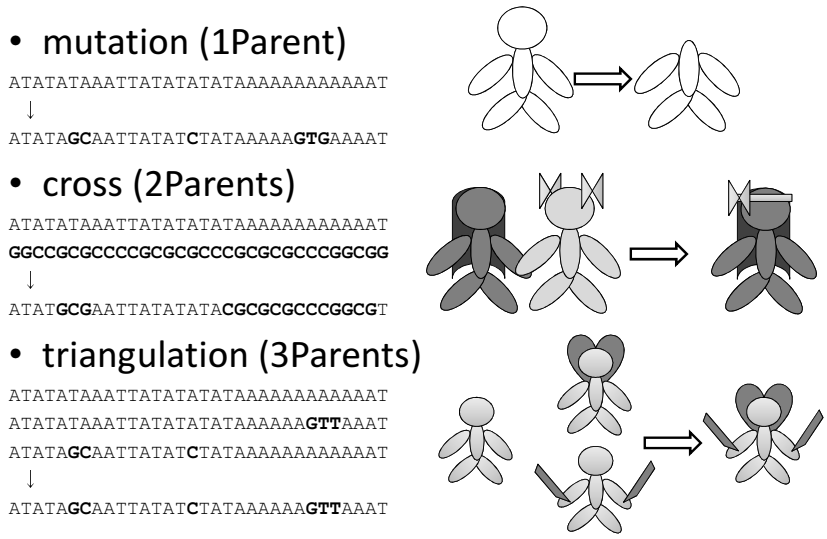


図 8.4: 新個体を創り出す三つの方法。絵が酷いですね。

— では次に、新個体を作り出す方法について説明してくれなイカ？

「図 8.4 のように、新個体を作る方法は三つ採用されているでゲソ。まず一つ目は突然変異。ひとつの遺伝子を『ドロー』し、ランダムに書き換えて、別の個体をつくり出す操作でゲソ。」

— 遺伝的アルゴリズムの基本となるやつだ。もちろん、ランダムに書き換えるだけなので、ほとんどの場合は性能の向上にはつながらず、致命的な結果になる場合もままある。そういった無数の変異体の屍を礎にして、進化は成り立っているのだ。

「二つ目は交尾。つまりまどほ m」

— おっと、それ以上いけない。そもそも crossing の訳は交尾じゃなくて交叉だ。交叉とは、ふたつの遺伝子を『ドロー』し、それらをいくつかのランダムな点で分割した上で入れ替えることで、新個体を作りだす操作だ。

「三つ目は三角合成。これは独自の概念でゲソ。」

— なぜ独自の概念を導入する必要があったのだろうか。

「困難を分割せよ、という格言があるじゃなイカ？ 大きなプログラムにはたくさんの最適化ポイントがあるから、沢山の小問題に分割してからその成果を統合できれば効率がよいでゲソ。FFTW などでは、大きなサイズの FFT を再帰的に部分問題に分割した上で、小さな問題から順に最速の実装を確定していくという方法がとられている [2] でゲソ。ところが、Paraiso の扱う問題では部分問題を切り出すのはイカにも難しくなイカ？」

— Paraiso の扱う問題は数千ノードのモノリシックなデータフローグラフで、FFT のような再帰的構造をもたず、どこで切り分ければよいのかは自明でない。かりに切り出したとして、その部分グラフを最適化するには、その部分だけをコードとして生成し、適切な数の入力を与え出力をテストするモジュールをとりつけなければならない。確かに骨のおれそうな作業だ。

「だから違うのでゲソ。そういうアプローチは取らないでゲソ。全体のグラフに埋め込んだまま、

Base	0	0	0	0	1	1	1	1
Primary	0	0	1	1	0	0	1	1
Secondary	0	1	0	1	0	1	0	1
Child	0	1	1	1	0	0	0	1

表 8.1: 三角合成のビット演算表でゲソ！

いわば in vivo で各部分グラフをチューンし、その結果を合成してやれればよいじゃなイカ？」

— なるほど。具体的にはどういう操作をするのだろう。

「基本的なアイデアはこうでゲソ。まず先祖 (Base) となるべき個体と、そいつから別々に進化した二つの個体 (Primary, Secondary) をランダムに選択するでゲソ。それから、『親に比べていづれかの子が変化していたら、その変化を取り入れる』という方針でそれらを合成。つまり表 8.1 のようなビット演算を施すでゲソ。これにより、Primary と Secondary に別々の改良点が生じていたら、それらを合成できるでゲソ。」

— でも、それだと家系図を辿るのが大変じゃなイカ？

「そこでもうひと工夫でゲソ。単に 3 つの遺伝子を『ドロー』し、スコア順にソートして低い方から Base, Secondary, Primary とすることにしたのでゲソ。」

— そんな手抜きで大丈夫か？

「それを否定したとして、君は僕の言葉を信じるカイ？『流体計算』という実アプリを前にして、Paraiso がどこまでやれたのか、その目で見届けるがいいでゲソ！」

8.4 種をまかなイカ？

「 今回は 2 次精度、HLLC 方式 [7] の流体シミュレータを Paraiso で実装したので、これの自動チューニングを試みるでゲソ。

シミュレータは流体の状態をマス目にわけて表現しているでゲソ。このマス目を空間補間して壁での量を求めたり、壁の量を入力に Riemann 問題をといて壁を通過する流量を求めたり、それを次元の数だけ足し合わせたり、さらに時間精度を高めるために以上の手順を繰り返したりと、流体シミュレータのやることはたくさんあるでゲソ。

まして、それを異なるハードウェアに移植したり、リファクタリングして速くしたりするのは大変じゃなイカ？ そこで、コード生成と自動チューニングの出番でゲソ！」

「自動チューニングを始めるにあたって、まずは初期個体を与える必要があるでゲソ。そこで、Manifest 指定をまったく行わないデフォルト個体 (*Izanami*) をもとに、」

「人間がコード書いたらこのへんメモリに置くだろ常考」

「といえる個体を 3 つばかし作ってみたでゲソ (c.f. 表 8.2)。その中でも一番速かったのに *Shinatsuhiko* という名前をつけたでゲソ。

Paraiso のソースコードとして *Shinatsuhiko* と *Izanami* を比較するなら、追加されている Manifest アノテーションはたった一つでゲソ。このたった一つの変更で、遺伝子は 16 文字変わり、生成されるコードの行数は1/3に、サブルーチンの数は 7 つから 11 に、メモリ消費量は 1.3 倍に、そして速度は一桁向上しているでゲソ。」

— コピペだらけの壊れやすいコードのままで、これらのリファクタリングを全部やってね ♡ なんて、とてもお願いできない。そんなことをしたら、プログラマの肉体の痛覚がどれだけの刺激を受けるか、想像もつかないだろう。そもそもそんなコードでは、ロジックの存在なんて、最初から自覚できているわけがない。そこは命令列の集まりでしかないし、ここにも命令列の集まりがあるだけだ。そのくせ、ロジックの一貫性

個体名	行数	subKernel 数	メモリ消費量	速度 (単精度)	速度 (倍精度)
Izanami	13128	7	$52 \times N$	1.551 ± 0.0005	1.138 ± 0.0004
Iwatsuchibiko	17494	12	$68 \times N$	5.015 ± 0.002	2.475 ± 0.001
Shinatsuhiko	3010	11	$68 \times N$	42.682 ± 0.083	19.253 ± 0.044
Hayaakitsuhime	3462	15	$84 \times N$	34.100 ± 0.110	15.687 ± 0.022

表 8.2: 手でアノテートしたコードの性質と速度。生成されたコードの行数、その中の subKernel の数、流体マス目の数 N に対して何倍のメモリを消費するか、単精度と倍精度のコードの速度、など。速度の単位は一秒間に何百万メッシュ計算できたか (Mega cell update per second) で測っている。それにしても、神々産みし始原の女性に息長き風王神 (izanami / shinatsuhiko)。中々よい名前だ。日本神話を愛する人ごめんなさい。

RunID	精度	初期スコア	所要時間	最高個体の ID/総数	ハイスコア
GA-1	倍	1.138 ± 0.0004	3870	20756 / 20885	14.158 ± 0.002
GA-S1	倍	1.138 ± 0.0004	4120	33958 / 34328	16.247 ± 0.002
GA-4	倍	19.253 ± 0.044	5811	39991 / 40262	35.303 ± 0.035
GA-F	単	42.682 ± 0.083	2740	23019 / 23062	53.300 ± 0.078
GA-D	倍	19.253 ± 0.044	8770	59841 / 68138	34.968 ± 0.043
GA-F2	単	42.682 ± 0.083	4811	22242 / 24887	53.656 ± 0.078
GA-DE	倍	19.253 ± 0.044	7928	41250 / 41386	31.015 ± 0.032

表 8.3: 複数の自動チューニング実験の統計。初期に与えた個体は、GA-1 と GA-S1 では *Izanami* であり、そのほかは *Shinatsuhiko* である。各実験について、採用した精度、初期にあたえた個体のスコア、チューニングにかけた実時間（単位：分）生成した個体の総数とそのうちスコアの平均値 $\mu(S)$ が最も高かった個体の ID、そしてその最高個体のスコア（単位：Mcups）を示す。Mcups は一秒あたり 10^6 メッシュを計算する程度の能力である。

が維持できなくなると、コードの存在意義は消滅してしまう。

そうならないよう、Paraiso のソースコードには、コード生成器である Builder モナドとその演算、という、コンパクト[*2]で安全[*3]な姿が与えられている。だからただの一発で、ここまでの変更を一気に施せるのだ。

「呪わしきコピペ作業はすべて、Paraiso が引き受けてくれることで、プログラマは最後まで笑顔でいられるのでゲソ！

またそうすることで Paraiso は、変更点を力任せとばかり次々に試しての自動チューニングをも可能にしたのでゲソ！」

8.5　統計しなイカ？

「まずはお断りでゲソ。イカの測定は、東京工業大学の TSUBAME2.0 で、ノードあたり 2 枚の Intel Xeon X5670 CPU(2.93GHz, 6Cores x HT = 12Cores) 3 枚の M2050 GPU(1.15GHz 14MP x 32Cores/MP=448Cores) が刺さっている環境で行ったでゲソ。計算はとくに断りなき限りすべて 1 枚の GPU を使ったコードで、1024^2、2 次元、2 次精度の HLLC 流体ソルバーの性能でゲソ。This project was partially supported by JST, CREST through its research program: “Highly Productive, High Performance Application Frameworks for Post Petascale Computing.”」

— イカ娘が突然英語をシャベッタアアア。これも科学少女として契約した者の、逃れられない義務なのであろう。

[*2] DRY 原則がとことん貫かれているという意味で

[*3] 好きにアノーテートしても決して意味が変わらないという意味で

「さて、表 8.3 がこれまでに行った自動チューニング実験の統計でゲソ。一日あたりざっと一万個体を生成できるでゲソ。そして、数日放置することで、*Izanami* は *Shinatsuhiko* なみの速度になり、*Shinatsuhiko* はもとの２倍近い速度になったでゲソ。

この性能がイカほどのものか、本当のところは完全手書きでチューンしたコードと較べてみないと分からないでゲソが、同じ GPU で流体コード、というカテゴリーなら、Schive [6] らにより単精度で C2050 一枚あたり 30Mcups（ただし彼らは 3 次元かつ計算量は多いスキームを採用）、Asunciónа [1] らにより単精度で GTX580 一枚あたり推定 6.8Mcups（こちらは 2 次元）などの性能が報告されているので、いい線いってるんじゃなイカ？」

「並列計算言語を作りたいという願いを掲げてエラい人と契約 (c.f. 前作『簡約（かんやく）！ ラムダ カ 娘（むすめ）』) して約二年。ようやく自動チューニングを動かすことができたでゲソ。次なる最大の課題の一つは、分散計算への対応でゲソ。これに関しては C ライクな DSL で GPU 並列計算を記述できる Physis [5] や、C や Fortran の文法と OpenMP 的指示文で MPI 計算を記述できる XcalableMP [8] など渡りに舟の言語が生まれてきているでゲソ。ぜひこれらの言語にむけてもコードを生成してゆきたいでゲソ！

計算したいものを Haskell で表現し、OpenMP や CUDA、将来的には様々な手続き型言語のコードを生成。さらに自動チューニングを ruby で制御と、様々な言語のおかげで、今の Paraiso があるのでゲソ。

すべての言語開発者に、そして他の著者が良い紹介記事を書く中、拙作ライブラリの記事を書かせてくれた執筆陣と、読んでくれた読者に、ありがとうでゲソ！」

参考文献

[1] DE LA ASUNCINA, M., CASTROC, M., FERNNDEZ-NIETOB, E., MANTASA, J., ACOSTAC, S., AND GONZLEZ-VIDAD, J. Efficient GPU implementation of a two waves WAF method for the two-dimensional one layer shallow water system on structured meshes.

[2] FRIGO, M., AND JOHNSON, S. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216—231.

[3] HUKUSHIMA, K., AND NEMOTO, K. Exchange monte carlo method and application to spin glass simulations. *Journal of the Physical Society of Japan 65* (1996), 1604–1608.

[4] KELLER, G., CHAKRAVARTY, M., LESHCHINSKIY, R., PEYTON JONES, S., AND LIPPMEIER, B. Regular, shape-polymorphic, parallel arrays in haskell. *ACM SIGPLAN Notices 45*, 9 (2010), 261—272.

[5] MARUYAMA, N., NOMURA, T., SATO, K., AND MATSUOKA, S. Physis: An implicitly parallel programming model for stencil computations on Large-Scale GPU-Accelerated supercomputers.

[6] SCHIVE, H., ZHANG, U., AND CHIUEH, T. Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR. *Arxiv preprint arXiv:1103.3373* (2011).

[7] TORO, E. F. *Riemann Solvers And Numerical Methods for Fluid Dynamics: A Practical Introduction*, 3 ed. Springer-Verlag, June 2009.

[8] 李珍泌, 朴泰祐, AND 佐藤三久. 分散メモリ向け並列言語 XcalableMP コンパイラの実装と性能評価. **情報処理学会論文誌. コンピューティングシステム** *3*, 3 (Sept. 2010), 153–165.

第9章

Java VM上で tail call しなイカ？

— @yryozo, @xhl_kogitsune

お前たち、海の中でも海の家でも陸の上でも、Linux の上でも FreeBSD の上でも AIX の上でも x86 の上でも Power の上ででも、できることならいつでもどこででも関数型言語を使いたいと思うはずでゲソ。でも、言語処理系をマルチプラットフォーム対応にするのは大変な労力でゲソ。そこで、Java VM 上で関数型言語を実行したり実装したりしなイカ!?

9.1 Java VM と関数型言語は相性が悪いでゲソ？

最近は Java VM の上でも関数型言語を走らせている人が多くなってきているでゲソ。特に Scala や Clojure なんかが人気が高いようでゲソね。それもこれも私の侵略計画が順調に進んでいる証拠じゃなイカ？

Java VM の上での実装には、おおまかに 2 つの方法があるでゲソ。

インタプリタ　Java VM 上で実装したい言語のインタプリタを実行する方式でゲソ。

トランスレータ　実装したい言語のソースコードを、Java VM 上のプログラム (Java バイトコード) に変換し、その変換した Java バイトコードを Java VM 上で直接実行する方式でゲソ。実装したい言語の関数が、Java のメソッドに変換されるものが多いでゲソ。

ところが、Java VM と関数型言語 (のトランスレータ方式の実装) は相性があまりよくないという人たちもいるのでゲソ。というのも、関数型言語に必須と言ってもいい末尾呼び出しの最適化 (Tail Call Optimization, TCO) が Java VM ではすごくやりにくいのでゲソ。理由は下の表に書いた通りでゲソが、Java VM にはスタックフレームを積んだりポップしたりせずにメソッド外に飛ぶという命令がないのでゲソ[*1]。TCO がないと、Scheme で末尾再帰や tail call でループ書いただけで即死でゲソ!

分類	命令名
メソッド外に飛べるが、スタックフレームを積むかポップしてしまう	`invokevirtual`, `invokestatic`, `invokeinterface`, `invokespecial`, `return`, `areturn`, `ireturn`, `lreturn`, `freturn`, `dreturn`, `athrow`
スタックフレームの深さは変えないが、メソッド外には飛べない	`if_acmp`, `if_icmp`, `if`, `ifnonull`, `ifnull`, `tableswitch`, `lookupswitch`, `goto`, `goto_w`, `jsr`, `jsr_w`, `ret`

実際、Scala や Clojure の言語仕様を読んで見ると末尾呼び出しについては少し (というかかなり) 歯切れが悪いでゲソ...。とはいえ、関数型言語はこんな程度の問題ではびくともしないでゲソ。ちゃんと解決策はあるでゲソよ。

[*1] ちなみに、この問題は Java VM だけのものではないんでゲソ。例えば関数型の言語を C にトランスレートして gcc なんかでコンパイルしようとする時にも同じ問題が起こるでゲソ。

分類	解決策
末尾再帰だったら？ (呼び出し先が自分自身)	・goto に変換可能でゲソ
末尾再帰じゃなかったら？	・複数の関数をひとつにまとめるでゲソ ・トランポリンするでゲソ ・tail call を扱えるようなインタプリタとして実装するでゲソ ・そもそも、そんな処理系を使ってるのが悪いんじゃなイカ？

末尾再帰の場合は簡単でゲソね。自分自身を再帰的に呼び出すだけならメソッドの外には出ないから、メソッド外には出れないけどフレームは積まない命令 (goto とか) で対処できるでゲソ。

末尾再帰じゃない場合はちょっと面倒でゲソ。末尾再帰じゃない tail call の例として、イカのような相互再帰を考えながら解決策を紹介するでゲソ。これは偶数・奇数判定を odd? と even? の相互再帰で実現した Scheme プログラムでゲソ。

```
(define (even? m)
  (if (= m 0) #t (odd?  (- m 1))))
(define (odd? n)
  (if (= n 0) #f (even? (- n 1))))
```

普段はこんなコード書かないでゲソが、tail call による相互再帰を書きたい欲望を抑えられない時もあるでゲソ。Java で書くとこんな感じでゲソ。

```
public static boolean isEven(int m){
    if( m == 0 ) return true;
    else         return isOdd(m - 1);
}
public static boolean isOdd(int n){
    if( n == 0 ) return false;
    else         return isEven(n - 1);
}
```

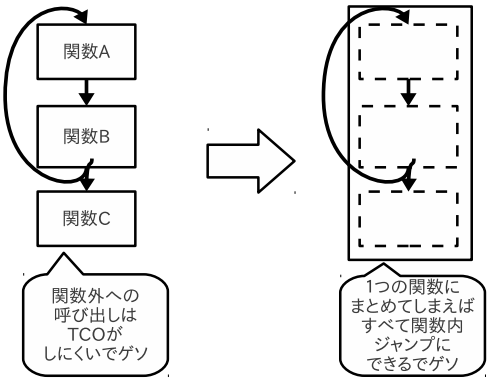
9.1.1 複数の関数をひとつにまとめるでゲソ

解決案の 1 つは、複数の関数を 1 つにまとめてしまうというものでゲソ。まとめてしまえば関数呼び出しも関数内でのジャンプになってしまうので、上の方法と同じように goto で対処できるでゲソ。

Java っぽく書くとこんな感じでゲソ。Java のソースコードで goto は使えないでゲソが、雰囲気を感じてほしいでゲソ。

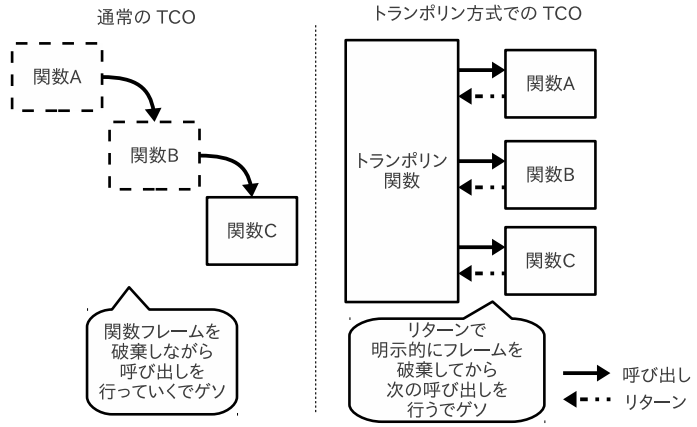
```
boolean isOdd(int n){
    int m;
isOdd:  // (odd? n)
    if( n == 0 ) return false;
    else { m = n - 1; goto isEven; } // call to even? (- n 1)
isEven: // (even? m)
    if( m == 0 ) return true;
    else { n = m - 1; goto isOdd;  } // call to odd?  (- m 1)
}
```

問題は、関連する関数を全部一度にコンパイルしないとイカんので、分割コンパイルとか外部ライブラリとかが使いにくくなる点でゲソ。あと、Java VM には 1 メソッドの大きさに制限 (64KB) があったりするのも辛いところでゲソ。Java VM 上の Scheme 実装の一つである Bigloo の論文 [7] 4.4.2 には、JIT のコンパイルするメソッドサイズ制限にひっかかった話が載っているでゲソ。



9.1.2 トランポリンするでゲソ

解決案のもう 1 つは、トランポリンという方式でゲソ。末尾呼び出しができないのは, スタックフレームの深さを変えずに呼び出しを行う (= スタックフレームの破棄と呼び出しを同時に行う) ことができないからでゲソ。同時にできないなら、スタックフレームの破棄と呼び出しを順に行ってやればいいじゃなイカ。トランポリン方式では、トランポリン関数という補助関数経由で関数を呼び出して、末尾呼び出しする代わりに一旦トランポリン関数にリターンし (これでフレームを破棄するでゲソ)、改めてトランポリン関数から次の関数を呼び出すのでゲソ。



Java で書くと例えばこんな感じになるでゲソ。

```
public class OddEvenByTrampoline{
    static interface Continuation{
        public Object invoke();
    }
    static class Odd implements Continuation {
        public Odd(int n) { _n = n; }
        public Object invoke(){
            if( _n==0 ) return new Boolean(false);
            else        return new Even(_n-1); // 元は tail call
        }
        private int _n;
    }
    static class Even implements Continuation {
        public Even(int m) { _m = m; }
        public Object invoke(){
            if( _m==0 ) return new Boolean(true);
```

```
            else        return new Odd(_m-1); // 元は tail call
        }
        private int _m;
    }
    public static Object trampoline(Object o){
        while(o instanceof Continuation)
            o = ((Continuation)o).invoke();
        return o;
    }
    public static void main(String[] args){
        System.out.print(trampoline(new Even(1000 * 1000 * 1000)));
    }
}
```

Tail call でメソッドを呼び出す代わりに、tail call するための情報を一旦返す (return する) でゲソ。一旦返した後で、再度トランポリン (trampoline 関数) から呼び出すでゲソ。こうすることで、スタックを消費せずに tail call できるでゲソ。

9.1.3 トランポリンを改良するでゲソ

トランポリン方式は毎回リターンがあったりして少し遅くなってしまうのが欠点でゲソが、Michel Schinz 氏と Martin Odersky 教授 [*2] が提案した改良型のトランポリン [5] もあったりするでゲソよ。こちらは毎回トランポリン関数にリターンする代わりに、何十回かフレームを貯めておいてまとめてトランポリン関数にリターンするのでゲソ。

Java だと例えばこんな感じになるでゲソ。スタックオーバーフローしないように tail call の段数を数えながら普通にメソッドコールを繰り返して、段数が上限 (このコードでは 1000 段) を超えたらトランポリンまで戻るでゲソ。論文では、トランポリンまで戻る時にメソッドリターンを繰り返す方法 (下のコード) と例外で戻る方法が比較されているでゲソ。

```
public class OddEvenByTrampoline2{
    static interface Continuation{
        public Object invoke();
    }
    static class Odd implements Continuation {
        public Odd(int n) { _n = n; }
        public Object invoke(){ return isOdd(0,_n); }
        private int _n;
    }
    static class Even implements Continuation {
        public Even(int m) { _m = m; }
        public Object invoke(){ return isEven(0,_m); }
        private int _m;
    }
    public static Object isOdd(int depth, int n){
        if( depth>=1000 ){ return new Odd(n); }
            // tail call 段数が上限超えたらトランポリンへ戻る
        if( n==0 ) return new Boolean(false);
        else       return isEven(depth+1,n-1);
    }
```

[*2] Martin Odersky 教授は Scala の設計者、Michel Schinz 氏はそのお弟子さんでゲソね。

```
    public static Object isEven(int depth, int m){
        if( depth>=1000 ){ return new Even(m); }
            // tail call 段数が上限超えたらトランポリンへ戻る
        if( m==0 ) return new Boolean(true);
        else       return isOdd(depth+1,m-1);
    }
    public static Object trampoline(Object o){
        while(o instanceof Continuation)
            o = ((Continuation)o).invoke();
        return o;
    }
    public static void main(String[] args){
        System.out.print(trampoline(new Even(1000 * 1000 * 1000)));
    }
}
```

9.1.4 処理系を改良するでゲソ

最後に、そもそもスタックフレームを積まずに関数呼び出しができるように処理系を改良するという手もあるでゲソ。例えば、C−− (しー・まいなす・まいなす) [3] という言語は、Haskell の設計者である Simon Peyton Jones 先生が作った C 言語の修正版でゲソ。それから、マイクロソフトで作られている.Net のランタイム (CLR) も Java VM に似た命令セットを持っているでゲソが、 Java VM の反省を生かしてちゃんと末尾呼び出しができるように改良が施されているでゲソ。

これでみんなにも関数型は Java VM ごときに負けたりはしないということが分かったんじゃなイカ？ それでもまだ納得できないという読者諸君のために、実際に関数型の侵略の前にはどんな抵抗も無意味だという所を見せつけてやるでゲソ。

9.2 Java VM 上の関数型言語実装を紹介するでゲソ!

地上には Java VM 上に実装された関数型言語のランタイムがたくさんあるでゲソ。

英語版 `Wikipedia: List of JVM languages`
`<http://en.wikipedia.org/wiki/List_of_JVM_languages>`

に関数型以外も含めて列挙されているでゲソ。 Java VM を侵略しなくとも、ここまで侵略できるでゲソ! 今日はその中からいくつか紹介するでゲソ! jar ファイルだけで動いたり、アーカイブを展開するだけでインストール完了なものが多いでゲソ。

それぞれのランタイムが tail call に対応しているかも軽く試してみたでゲソ。イカの 3 つのプログラムを書いて、ちゃんと落ちずに動かせるかどうかを見たでゲソ。いくつかのランタイムについては軽く実装を解説するでゲソ。

先に断っておくでゲソが、末尾再帰、末尾呼び出しするだけが関数型言語でのループの表現ではないでゲソ。末尾呼び出しが回数無制限にできなくても、他のナイスな方法で書ければいいんじゃなイカ。このあたり、どのように書くかは言語によって異なるでゲソ。でも、今日のわたしは末尾呼び出しがしたい気分なので末尾呼び出しで書くでゲソー♪

末尾再帰 末尾再帰でフィボナッチ数第 1,000,000,000 項を 100,000,007 で割った余りを計算するプログラムを各言語で書いて実行してみたでゲソ[*3]。

```
(define (fib n)
  (letrec
```

[*3] 余りを取っているのは、単に演算結果が巨大な多倍長整数になることを防ぐためでゲソ

```
((fib-iter (lambda (n a b)
    (if (<= n 0)
       a
       (fib-iter (- n 1) b (modulo (+ a b) 100000007))))))
  (fib-iter n 1 1)))
```

相互末尾再帰 前の節で使った even/odd の例の相互末尾再帰による実装を実行してみたでゲソ ((even? 1000000000))。

トランポリン even/odd の例をトランポリンで実装して実行してみたでゲソ ((even? 1000000000))。

結果をざっとまとめるとこんな感じになるでゲソ♪[*4][*5]

- 「×」は、スタックオーバーフローなどで落ちた場合でゲソ。
- 「○」はちゃんと実行できた場合でゲソ。
- 「-」は、コンパイルを通るプログラムの書き方が分からなかったので試してないでゲソ。

言語	ランタイム	末尾再帰	相互末尾再帰 [*a]	トランポリン
Scheme	Kawa –full-tailcalls	○	○/○ [*b]	○
	Kawa	○	○/× [*b]	○
	Bigloo	○	○/×	○
	SISC	○	○/○	○
	JScheme	○	○/○	○
	JAKLD (末尾再帰対応版)	○	○/○	○
	JAKLD (オリジナル)	×	× /×	×
Common Lisp	ABCL	×	×	○
	CLforJava	×	×	○
	Jatha	×	-[*c]	-[*c]
OCaml	OCamlJava	○	×	○ [*d]
Funnel	Funnel	○	×	-[*e]
	Funnel -tail-calls:ret	○	○	-[*e]
	Funnel -tail-calls:exn	○	○	-[*e]
Clojure	Clojure	○ [*f]	×	○
Scala	Scala	○	×	○

[*a] Scheme の左は letrec で、右は define でそれぞれ書いた場合でゲソ。

[*b] プログラムの書きかたによっては×だったでゲソ。

[*c] Jatha は、do とかがないので実装できなかったりコンパイルが通らなかったりしたでゲソ。

[*d] プログラムの書きかたによっては×だったでゲソ。環境 (Windows/cygwin, FreeBSD) によっても落ちることがあったでゲソ。

[*e] Funnel では、型が合わせられなくてトランポリンの書き方が分からなかったでゲソ…。

[*f] Clojure では、末尾再帰でループを書くために用意されている loop, recur を使って書いたでゲソ。

9.2.1 Scheme

まずはわたしの大好きな Scheme の話でゲソ! Scheme は言語仕様 [1] 上で追加でスタック領域を消費せずに tail call できることを規定しているので、JVM 上の各ランタイム実装もがんばって tail call のサポートをしているでゲソ! 素晴らしいじゃなイカ!

[*4] ただ、書き方の違いで結果が違ったりすることもあったでゲソ。あと、相互末尾再帰の例は tail call の例としてはかなり単純で小さいので、もっと複雑な tail call になると結果は違うかもしれないでゲソ

[*5] いくつかの処理系や言語は今回初めて使ったものだったゲソ。書き方が悪くて通らないだけだったらごめんなさいでゲソ。

Kawa

公式サイト　http://www.gnu.org/software/kawa/
バージョン　1.11 (2010-11-11)

トランスレータ方式による Scheme 実装でゲソ。

```
$ java -cp kawa-1.11.jar kawa.repl --full-tailcalls --main -C program.scm
  # Scheme プログラムから Java バイトコードにコンパイル
$ java -cp 'kawa-1.11.jar:.' program
  # 実行
```

みたいにすると実行できるでゲソ。Java クラスファイルが出力されるでゲソ。

Kawa 1.11 では letrec で相互再帰を書くと、相互再帰をしている関数をまとめて 1 つのメソッドにコンパイルするようでゲソ。

また、--full-tailcalls オプションをつけると、general tail call elimination が有効になるでゲソ[*6]。具体的にはイカのように define で相互再帰を書くとトランポリンを用いた実装に自動変換されたでゲソ。

```
(define (even? x) (if (= x 0) #t (odd?  (- x 1))))
(define (odd? x)  (if (= x 0) #f (even? (- x 1))))
```

ただ、書き方によってはコンパイルに失敗したり、適切に変換されずに実行時にスタックオーバーフローを起こすという場合があるようでゲソ。

Java のメソッドも呼び出せるし[*7]、Java から Scheme の式を評価したりもできるでゲソ[*8]。

Bigloo

公式サイト　http://www-sop.inria.fr/indes/fp/Bigloo/
バージョン　3.7a (2011-09)

Bigloo は、Java VM 以外にも C や.NET CLR などにもコンパイルできる Scheme 実装でゲソ。こんな感じでコンパイルと実行ができるでゲソ[*9]。

```
$ /path/to/bigloo/bin/bigloo -jvm program.scm
  # Scheme プログラムから Java バイトコードにコンパイル
$ java -cp ".:/path/to/bigloo/lib/bigloo/3.7a/bigloo_s.zip" program
  # 実行
```

Bigloo 自体は C 言語とかも使って書かれているでゲソが、bigloo を使ってコンパイルした後は、出力されたクラスファイルと bigloo_s.zip と Java 環境があれば bigloo 本体がない環境にも持っていって実行できるでゲソ。

マニュアルには、全部じゃないけどほとんどの末尾再帰は最適化されると書いてあるでゲソ[*10]。

Bigloo では、第一級の値として使われることのない、常に tail call される関数は 1 つのメソッドに変換されるようでゲソ [7]。試しに相互再帰を letrec で書いたら相互再帰している関数がまとめられて 1 つのメソッドにコンパイルされたでゲソ。define で書いたらスタックオーバーフローでダメだったでゲソ...。

[*6] http://www.gnu.org/s/kawa/Restrictions.html
[*7] http://www.gnu.org/s/kawa/Method-operations.html
[*8] http://www.gnu.org/s/kawa/Evaluating-Scheme-expressions-from-Java.html
[*9] bigloo.jheap がないというエラーが出た場合は、-lib-dir /path/to/bigloo/lib/bigloo/3.7a のようなオプションで bigloo.jheap のあるパスを指定する必要があるかもしれないでゲソ
[*10] "Bigloo is not able to compile all the tail recursive call without stack consumption (however, most of the tail recursive calls are optimized by Bigloo and don't use stack activation frames).", Bigloo User manual 27.3 http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo-28.html#Limitation-of-the-JVM-back-end.

SISC

公式サイト `http://sisc-scheme.org/`
バージョン 1.16.6 (2007-02-27)

JScheme

公式サイト `http://jscheme.sourceforge.net/jscheme/main.html`
バージョン 7.2 (2006-02-04)

JAKLD

公式サイト `http://ryujin.kuis.kyoto-u.ac.jp/~yuasa/jakld/index-j.html`
バージョン

SISC, JScheme, JAKLD は Java VM 上での Scheme インタプリタでゲソ。JAKLD にはオリジナル版と末尾再帰対応版があるでゲソ。SISC, JScheme と JAKLD の末尾再帰対応版は tail call に対応しているようでゲソ。

9.2.2 Common Lisp

Common Lisp は、Scheme と並ぶ Lisp 系の主要なバリアントでゲソが、こちらは言語仕様上では tail call optimization について規定していないでゲソ[*11]。Armed Bear Common Lisp, CLforJava, Jatha という Java VM 上での実装があるでゲソ。3 つとも、jar ファイルだけで動作するでゲソ。

Armed Bear Common Lisp (ABCL)

公式サイト `http://common-lisp.net/project/armedbear/`
バージョン 1.0.0 (2011-10-23)

CLforJava

公式サイト `http://clforjava.org/`
バージョン 20091212 (2009-12-12)

Jatha

公式サイト `http://jatha.sourceforge.net/`
バージョン 2.8 (2007-04-25)

これらの JVM 上での実装では末尾再帰でも相互末尾再帰でもスタックオーバーフローだったでゲソ。

9.2.3 OCaml

OCaml は ML 系言語の一つでゲソ[*12]。パターンマッチやバリアントが便利でゲソ。ツリー操作とかが書きやすいので、コンパイラの実装にも使われることが多いでゲソ。たとえば、C Intermediate Language (CIL) [4][*13] は OCaml で書かれた C 言語のパーザと中間表現上での解析器でゲソ。

本家ネイティブ実装である OCaml, OCamlopt は末尾再帰や tail call optimization に対応しているので tail call し放題でゲソ。

OCamlJava

公式サイト `http://ocamljava.x9c.fr/`
バージョン 1.4 (2010-02-06)

Java VM 上の実装としては OCamlJava があるでゲソ。OCamlJava では、直接の末尾再帰はループに変換されるでゲソ。一般の tail call はそのまま Java のメソッド呼び出しに変換されるのでスタックオーバーフローするでゲソ。これは、ドキュメント (アーカイブ内の `ocamljava-bin.pdf` など) に未対応と書いてあるで

[*11] SBCL など、ネイティブ実装のいくつかは tail call optimization を行うでゲソ。
[*12] OCaml 公式サイト: `http://caml.inria.fr/`
[*13] `http://cil.sourceforge.net/`

ゲソ。

9.2.4 Funnel

Funnel

公式サイト	`http://lamp.epfl.ch/funnel/`
バージョン	Release 9 (2002-03-14)

改良型のトランポリンが実装されているランタイムでゲソ。

デフォルトでは末尾再帰は最適化されるようでゲソが、一般の tail call は最適化されないので相互再帰とかを書くと落ちるでゲソ。

`-tail-calls:ret` または `-tail-calls:exn` オプションをつけると、9.1.3 節で説明した改良型のトランポリンが有効になるでゲソ。それぞれのオプションは、トランポリンまで戻る時にメソッドリターンを繰り返すか、例外で抜けるかに対応するでゲソ。これをつけると相互再帰でも落ちなくなるでゲソ!

9.2.5 Clojure

Clojure

公式サイト	`http://clojure.org/`
バージョン	1.3 (2011-09-23)

Clojure は Java VM 上で実行することを考えて設計された言語でゲソ。

Clojure では、末尾再帰を明示的に書いてループを表現するために、`loop`, `recur` というものが用意されているでゲソ。これを使って書けばちゃんとループが書けるでゲソ。普通に直接末尾再帰を書くとスタックオーバーフローで落ちてダメでゲソ。こんな感じで使うでゲソ。

```
(defn fib [n0 a0 b0]
  (loop [n n0 a a0 b b0]
    (if (<= n 0)
        a
        (recur (- n 1) b (mod (+ a b) 100000007)))))
```

`loop` で、末尾再帰で呼ばれる関数を定義するようなものでゲソ。`[n n0 a a0 b b0]` というのは、この `loop` の引数は `n`, `a`, `b` で、その初期値は `n0`, `a0`, `b0` である、という意味でゲソ。そして `recur` で `loop` を呼び出すでゲソ。`recur` の呼び出しが末尾再帰になっていることはコンパイラがチェックするでゲソ。スタックを消費せずにループが書けるのは `recur` だけでゲソ[*14]。

一般の tail call 最適化はサポートしていない[*15] ので、トランポリンを使うしかないでゲソ。トランポリンを楽に書くための記法も用意されているでゲソ [*16]。『プログラミング Clojure』によると、遅延評価やシーケンスなどのトランポリン以外の方法を推奨しているようでゲソ。

9.2.6 Scala

Scala

公式サイト	`http://www.scala-lang.org/`
バージョン	2.9.1 (2011-08-29)

Scala も Java VM 上での実行を考えて設計された新しい言語でゲソ。

末尾再帰は最適化される模様でゲソ。相互再帰とかは、Clojure 同様トランポリン等を使う必要があるでゲ

[*14] “Note that recur is the only non-stack-consuming looping construct in Clojure. ” – `http://clojure.org/special_forms`

[*15] “Since Clojure uses the Java calling conventions, it cannot, and does not, make the same tail call optimization guarantees.”, – `http://clojure.org/functional_programming`

[*16] 実例はこの章の末尾に載せてみたでゲソ

ソ [17]。

9.3 Java VM 自体も侵略するでゲソ

それでは次に Java VM の内側も侵略してみるでゲソ！

Da Vinci Machine Project というプロジェクトを知っているでゲソか？ これは OpenJDK を Java 以外の言語からも上手く使えるようにするためのプロジェクトでゲソが、実は Java VM で TailCall を可能にするためのパッチが既に作られているのでゲソ [6]。OpenJDK といえば、Java における事実上の標準で 2011 年 7 月には Java VM のリファレンス実装にも指定されたという、まさに Java VM 界の本丸でゲソ。つまり、このパッチさえあれば Java VM への侵略は完了したも同然じゃなイカ！

```
Da Vinci Machine Project
<http://openjdk.java.net/projects/mlvm/subprojects.html>
```

9.3.1 TailCall 機能付きの Java VM をコンパイルしてみるでゲソ

それではさっそく Java VM で TailCall できるようにしてみるでゲソ。なお、イカの手順は Ubuntu 11.10(x86-32) で試したものなので、他の環境の人は適宜読み替えてほしいでゲソ。(ちなみに、今のところ TailCall の実装は x86-32 版しか存在しないでゲソ…。)

0. まずは、ビルドに必要な環境を整えるでゲソ。とりあえず、イカのパッケージを入れておけばいいんじゃなイカ。

```
$ sudo aptitude build-dep openjdk-7
$ sudo aptitude install openjdk-6-jdk
## libjibx は 1.1 系列でないとイカんようでゲソ。(sourceforge から落とすときは
## 1.1.6a を選ぶでゲソ)
$ sudo aptitude install libjibx1.1-java
### パッケージを入れた場合はリンクが張られないので自力で張るでゲソ…。
$ cd /usr/share/java
$ sudo ln -s jibx-run-1.1.jar jibx-run.jar
$ sudo ln -s jibx-bind-1.1.jar jibx-bind.jar
##  Mercurial の設定も必要でゲソ。hgforest extension を入れるでゲソ。
$ HgForestRoot=~/hgext/   # ここのディレクトリ名は適当でいいでゲソ
$ hg clone https://bitbucket.org/pmezard/hgforest-crew/overview/ \
   ${HgForestRoot}
$ cat >> ~/.hgrc << EOF
    [extensions]
    forest = ${HgForestRoot}/forest.py
  EOF
```

1. 次にパッチ自体とパッチ対象のソースコードをダウンロードするでゲソ。jdk7-b59 というバージョンが必要でゲソ。(ReadMe には b50 と書かれていたりするでゲソが、これは間違いでゲソ)

```
$ mkdir davinci; cd davinci  # ここのディレクトリ名は適当でいいでゲソ！
$ hg fclone -r jdk7-b59 http://hg.openjdk.java.net/jdk7/jdk7 sources
$ hg fclone http://hg.openjdk.java.net/mlvm/mlvm patches
```

2. 次に、パッチディレクトリとソースコードディレクトリの間にリンクをはるでゲソ。

[17] こちらも実例を章末に載せてみたでゲソ

```
$ bash patches/make/link-patch-dirs.sh sources patches
## 一応確認しておくでゲソ。左端に出る数字が同じであれば問題ないでゲソ。
$ ls -il patches/hotspot/series sources/hotspot/.hg/patches/series
```

3. 適用するパッチを選ぶでゲソ。今回はもちろん TailCall パッチを選ぶでゲソよ。ちなみに、buildable と testable はダメなパッチを弾くためのルールでゲソ。つけておいた方が変なものが入らなくていいでゲソ。

```
$ export davinci=$(pwd) guards="buildable testable tailc-lazy tailc"
$ sh patches/make/each-patch-repo.sh hg qselect --reapply $guards
```

4. 実際にパッチを適用するでゲソ。

```
$ sh patches/make/each-patch-repo.sh '(! hg qunapplied | read something) || \
  hg qpush -a'
```

5. いくつか修正しないとダメそうな箇所があるので、適当に直しておくでゲソ。

```
## execvpe という関数名は最近の unistd.h の中身と被っているようなので変えておくでゲソ。
$ sed -i 's/execvpe/azunyan_kawaii/g' sources/jdk/src/solaris/native/java/\
  lang/UNIXProcess_md.c
## ALSA 関係のリンク設定がおかしいようなので直しておくでゲソ。
$ sed -i 's/LDFLAGS/EXTRA_LIBS/' sources/jdk/make/javax/sound/jsoundalsa/\
  Makefile
## awt 関係で使っている定数名がおかしいようなので直しておくでゲソ。
$ sed -i 's/X_ShmAttach/XShmAttach/' sources/jdk/src/solaris/native/sun/\
  awt/awt_GraphicsEnv.c
```

6. それでは、実際に make するでゲソ。

```
## === 環境変数を設定しておくでゲソ ===
## JDK6 があるパスを指定するでゲソ
$ export ALT_BOOTDIR=/usr/lib/jvm/java-1.6.0-openjdk/
## libjibx があるパスを指定するでゲソ
$ export ALT_JIBX_LIBS_PATH=/usr/share/java
## 以下は指定するとコンパイルが並列実行されて速くなるでゲソ（数字は CPU のコア数と
## 相談でゲソ）。
$ export HOTSPOT_BUILD_JOBS=4
$ export PARALLEL_COMPILE_JOBS=4
## 新しい Linux カーネル（3.x）には対応してないのでバージョンチェックを回避するでゲソ
$ export DISABLE_HOTSPOT_OS_VERSION_CHECK=ok
## === ここまでで環境変数の設定は完了でゲソ ===
$ cd sources/
## SAN 値をチェック、ではなくビルド準備が整っているかどうかをチェックするでゲソ
$ make sanity
## さあ、ビルドするでゲソ
$ make
```

7. ビルドは終わったでゲソか？ 完了したら sources/build/linux-i586/bin/に java コマンドや javac コマンドが出来ているはずでゲソ。

9.3.2 動くかどうか試してみるでゲソ

それでは、実際に動かしてみるでゲソ。TailCall パッチには、Java 言語の拡張 (というか Java コンパイラの拡張でゲソね) と、Java VM の拡張の 2 つが入っているでゲソ。

まず Java 言語の方でゲソが、'goto' という予約語 [*18][*19]を使って末尾呼び出しにしたい箇所を指定できるようになっているでゲソ。

```
public class TailCallExample {
  public static int tailcaller(int x) {
    if (x==0) return x+1;

    return goto tailcaller(x-1);  // この呼び出しが末尾呼び出しになるでゲソ
  }
}
```

どうでゲソ？ まぁ、あまり Java には興味がないから、ここはどうでもいいでゲソね。

次に Java VM についてでゲソが、バイトコード上では 196(=0xc4) という 1 バイトのバイトコードが末尾呼び出しを表すプレフィックスになるでゲソ[*20]。そして、この直後に置かれたメソッド呼び出し命令 (invoke 系命令) が末尾呼び出しになるんでゲソ。これは、.Net とほぼ同じ方式でゲソね。

```
$ javap -c TailCallExample  # バイトコードを逆アセンブルするコマンドでゲソ
...
public static int tailcaller(int);
  Code:
   0:  iload_0
   1:  ifne   8
   4:  iload_0
   5:  iconst_1
   6:  iadd
   7:  ireturn
   8:  iload_0
   9:  iconst_1
   10: isub
   11: bytecode 196  // この直後に置かれた invoke 系の命令が末尾呼び出しになるでゲソ
   12: invokestatic  #2; //Method tailcaller:(I)I
   15: ireturn
}
```

そして、実行する時は普通に java コマンドでいいでゲソ (当たり前でゲソが、さっき自分でビルドした java コマンドを使うんでゲソよ)。ただし、今のところはコマンドラインオプションで-XX:+TailCalls とつけないと TailCall 機能は有効にならないようでゲソ。

[*18] Scheme の設計者である Guy Steele 博士も、末尾呼び出しとは引数を渡せる goto のようなものだ、と言ってたでゲソね

[*19] 余談でゲソが、goto は昔から Java の予約語なのに今に至るまで全く意味が定義されていなくて誰にも使えない予約語でゲソ (言語仕様には、C++ と間違って goto と書いてしまった人にコンパイラがエラーを吐くために予約語にした、みたいなことが書かれているでゲソ)。そのおかげで、このパッチは互換性を壊さずに末尾呼び出し機能を導入できているでゲソが、名前空間という貴重なシンボルの海をこんな風に浪費するのはイカがなもんでゲソ？ ほとんど予約語を持たない Scheme の美しさを見習うべきじゃなイカ？

[*20] ちなみに、この 196 というバイトコードはパッチを当てていない Java VM でも利用されていて、wide 命令 (wide プレフィックス) と呼ばれているのでゲソ。後ろについたロード/ストア命令のオペランド長を 1byte から 2byte に伸ばす効果があるんでゲソが、invoke 系命令には効果がないので、パッチを当てない状態では invoke 系の前についた時はエラーになるでゲソ。

```
java -XX:+TailCalls TailCallExample
```

ちなみに、書いたプログラムによっては TailCallException という例外が出ることがあるでゲソ。この場合は-XX:+TailCallsStackCompression というコマンドラインオプションをつければ出なくなるでゲソ[*21]。

9.3.3 Java 以外の言語からも使ってみようじゃないカ

というわけで、とりあえず Java からは TailCall 機能が使えることは分かったでゲソ。とはいえ、Java VM の機能なら Java 以外の言語からでも使えるはずじゃないカ。というわけで、とりあえず手近にあった Scala と Clojure からも使えるようにしてみるでゲソ。今回は「普通に末尾呼び出しで書くと、普通に末尾呼び出し最適化してくれる」状況を目指すでゲソ。

Scala で「普通に末尾呼び出しで書いた」even/odd の相互再帰テストプログラム

```
object MutualTailCallTest {
  def isEven(n : Int) : Boolean = {  if (n == 0) true   else isOdd(n-1)   }
  def isOdd(n : Int)  : Boolean = {  if (n == 0) false  else isEven(n-1)  }

  def main(args : Array[String]) : Unit = {
    var start = System.currentTimeMillis
    for (i <- 1 to 10) {  isEven(1000 * 1000 * 1000)  }
    var end = System.currentTimeMillis
    print((end - start)/1000.0); println("[sec]")
  }
}
```

Clojure で「普通に末尾呼び出しで書いた」even/odd の相互再帰テストプログラム

```
(ns mutualtailcalltest
  (:gen-class))

(declare my-even? my-odd?)
(defn my-even? [^long n]  (if (zero? n) true  (my-odd?  (unchecked-dec n))))
(defn my-odd?  [^long n]  (if (zero? n) false (my-even? (unchecked-dec n))))

(defn -main [& args]
  (let [start (System/currentTimeMillis)]
    (do
      (dotimes [_ 10]
        (my-even? (* 1000 1000 1000)))
      (. System/out print (/ (- (System/currentTimeMillis) start) 1000.0))
      (. System/out println "[sec]"))))
```

といっても、Scala や Clojure のコンパイラに手を入れるのは行儀がよくない気がするでゲソ[*22]。というわ

[*21] これは Java VM の protection domain という概念と関連しているでゲソ。Java VM はそれぞれのクラスをどこからロードしたかを覚えていて、これを使って「アプレットのようなインターネットからダウンロードしたクラスにはセキュリティ上重要な機能は使わせない」みたいな成魚を実現してるんでゲソが、このせいで末尾呼び出しでも関数フレームを削除できないことがあるのでゲソ (セキュリティチェック時には、その時点でスタックに積まれている全関数フレームの protection domain の共通部分を確認するのでゲソ。フレームを消してしまうとそのフレームに関する protection domain 情報もなくなってしまうので、このチェックがきちんと行えなくなってしまうんでゲソ)。というわけで、実行時に protection domain が異なるクラス間でメソッドが末尾呼び出しされると、デフォルトでは TailCallException を投げて異常終了してしまうでゲソ。ただし、-XX:+TailCallsStackCompression を指定していると、例外を投げる代わりに末尾呼び出しを諦めて、普通に関数フレームを積んでメソッド呼び出しするようになるんでゲソ。

[*22] 作者注: いや面倒くさかっただけです。

けで、今回は scalac や clojurec が出力したクラスファイルの方をいじってみようじゃなイカ。

scalac や clojurec の出力したクラスファイルを見たところ、どちらも関数呼び出しとリターンについては素直に Java VM のメソッド呼び出し命令とリターン命令にしているようでゲソ。

scalac によるコンパイル結果

```
public boolean isEven(int);
  Code:
   ...
   11:  iconst_1
   12:  isub
   13:  invokevirtual   #18; //Method isOdd:(I)Z
   16:  ireturn

public boolean isOdd(int);
  Code:
   ...
   11:  iconst_1
   12:  isub
   13:  invokevirtual   #24; //Method isEven:(I)Z
   16:  ireturn
```

clojurec によるコンパイル結果

(正確には、以下は `my_even` **の方だけ。**
**** `clojurec` **では関数** `1` **つ毎に** `1` **つのクラスを生成する。それぞれの関数の処理は、**
生成されたクラスの `invoke()` **メソッドが呼ばれると実行される模様。**
以下では、 `invokePrim` **の** `29byte` **目の** `clojure/lang/IFn.invoke` **の呼び出しが**
`my_odd` **の呼び出しに対応。)**

```
public final java.lang.Object invokePrim(long);
  Code:
   ...
   19:  checkcast       #54; //class clojure/lang/IFn
   22:  lload_1
   23:  invokestatic    #60; //Method clojure/lang/Numbers.unchecked_dec:(J)J
   26:  invokestatic    #64; //Method clojure/lang/Numbers.num:(J)Ljava/lang/..
   29:  invokeinterface #68,  2; //InterfaceMethod clojure/lang/IFn.invoke:(...
   34:  areturn

public java.lang.Object invoke(java.lang.Object);
  Code:
   0:   aload_0
   1:   aload_1
   2:   checkcast       #74; //class java/lang/Number
   5:   invokestatic    #78; //Method clojure/lang/RT.longCast:(Ljava/lang/....
   8:   invokeinterface #80, 3; //InterfaceMethod clojure/lang/IFn$LO.invokePri
   13:  areturn
```

というわけで、先人の英知 [*23] を参考に、イカのような方針で末尾呼び出しを見つけて tailcall プレフィックスをつけるプリプロセッサを作ってみたでゲソ！

1. invoke 系の命令 [*24] の直後に return 系の命令 [*25] があれば、それが末尾呼び出しでゲソ。
2. ただし、synchronized 指定されているメソッドの中では、どんな呼び出しも末尾呼び出しにはならないでゲソ。(synchronized 指定がある場合、暗黙のうちにメソッドに入るとロックがとられ、メソッドから出るとアンロックされるでゲソ。ということで、事実上全ての return 命令の直前にロックを解除する monitorexit 命令があるのと同義なので、invoke 系命令の直後に return という実行シーケンスがありえなくなるでゲソ。)
3. それから、例外ハンドリングの対象になっている範囲中の呼び出しも、末尾呼び出しにはならないでゲソ。(例外処理の範囲内 (Java で言う所の try{} の内部) では、例外が起きたときには例外ハンドラに飛ばないといけないので、末尾呼び出しでフレームを削ってしまってはイカんでゲソ。Scheme で dynamic-wind 中で末尾呼び出しにならないのと同じでゲソ。)

今回は Javassist [2] というライブラリを使ってお手軽作成したでゲソ [*26]。とりあえずプレフィックスの挿入部分はこんな感じでゲソ。

```
...
for (final CodeIterator ci = ca.iterator(); ci.hasNext();) {
  final byte[] tailprefix = {(byte)0xc4};

  final int line1 = ci.next();
  final int opcode1 = codes[line1] >= 0 ? codes[line1] : codes[line1]+0x100;
  if (! ci.hasNext()) { break; }
  final int line2 = ci.lookAhead();
  final int opcode2 = codes[line2] >= 0 ? codes[line2] : codes[line2]+0x100;

  if ((opcode1 == Opcode.INVOKEINTERFACE ||
       opcode1 == Opcode.INVOKEVIRTUAL   ||
       opcode1 == Opcode.INVOKESPECIAL   ||
       opcode1 == Opcode.INVOKESTATIC)
      &&
      (opcode2 == Opcode.RETURN  ||
       opcode2 == Opcode.ARETURN ||
       opcode2 == Opcode.IRETURN ||
       opcode2 == Opcode.LRETURN ||
       opcode2 == Opcode.FRETURN ||
       opcode2 == Opcode.DRETURN)) {

      ci.next();
      ci.insert(line1, tailprefix);
  }
...
```

やってることは、バイトコードをイテレートしていって、invoke 系の命令の後ろに return 系の命令が続くシーケンスを見つけたらその前にプレフィックスを挿入するのでゲソ。見たまんまでゲソね。

[*23] 詳細は、[6] を見るでゲソ！

[*24] invokevirtual、invokestatic、invokeinterface、invokespecial の 4 つでゲソ。正確には invokedynamic というのもあるでゲソが、今回のソースコードのバージョン (jdk7-b59) ではそこまで対応していないでゲソ...。

[*25] return、areturn、ireturn、lreturn、freturn、dreturn の 6 つでゲソ

[*26] 作者注: というか、お手軽過ぎで上の 1~3 のうち 1 のルールしか実装してません。お手軽と言うか手抜きですね...。

ちなみに、実際に使う場合は -javaagent というオプションでプリプロセサを指定するのでゲソ [*27]。この方式では、Java VM がクラスファイルをロードするときにメモリ上のイメージだけを書き換えるので、ディスク上のクラスファイルは変更しないのでゲソ。勝手にクラスファイルを壊したりはしないので安心でゲソよ。

```
## scalac でコンパイルするでゲソ
$ scalac MutualTailCallTest.scala
## scala コマンドに、ビルドした java コマンドの位置を指定してやるでゲソ
$ export JAVACMD=~/workspace/davinci/sources/build/linux-i586/bin/java
## 同じく、プリプロセッサや他のオプションを指定するでゲソ
$ export JAVA_OPTS="-XX:+TailCalls -XX:+TailCallsStackCompression \
  -Xverify:none -javaagent:tailcallpreprocessor.jar=MutualTailCallTest"
## 実行してみるでゲソ！
$ scala MutualTailCallTest
```

9.3.4 性能を調べてみるでゲソ

それでは、Scala と Clojure がどれだけ上手く相互再帰できるようになったかを確かめてみようじゃなイカ。次の表が結果でゲソ [*28]。

言語	通常時	パッチおよびプリプロセス	(参考) 明示的にトランポリン化
Scala 2.9.0 (on Java 7b59)	死亡確認！	4.872[sec]	117.557[sec]
Clojure 1.3.0 (on Java 7b59)	〃	67.291[sec]	407.451[sec]
Java (Java 7b59)	〃	4.762[sec]	-
(参考) OCaml(OCamlopt) 3.12	6.048[sec]		

通常時の欄に書いてある通り、普通に末尾呼び出しで書いて普通に Java VM 上で動かすと、これまではスタックオーバーフローで死んでしまっていたのでゲソ。が、Java VM を侵略してしまえば、死なないどころかかなりの速度で even/odd の相互再帰ができるようになったでゲソ。

参考までに、OCamlopt でネイティブコンパイルした OCaml プログラムとも競争させてみたでゲソ。x86 のマシン語を直接出力できる OCamlopt は関数型言語コンパイラの中でも速い方でゲソが、侵略後の Scala の方が速いでゲソ。これはもう Scala(スカラ) を超えた・・・「スクルト」と呼んでもいいんじゃなイカ[*29]！ これで Scala も一人前の関数型言語の仲間入りでゲソ。

一方の Clojure。お前に足りない物、それは (ry そして何よりも速さが足りない！ でゲソ。今回のテストプログラムだと、Clojure は関数を全て clojure.lang.IFn という関数オブジェクトにして、関数呼び出しを invoke() というインターフェースメソッド呼び出しにしているようでゲソが、invoke() の引数が全部 Object 型になっているので、整数みたいなプリミティブ型だと Object 型へのキャストと元の型へのキャストが入ってしまって遅いようでゲソ[*30]。

ちなみに、一目見て分かる通りトランポリンの性能がかなり悪く出ているでゲソが、これは even/odd がかなり極端な例ということもあるでゲソ (ほとんど関数呼び出ししかないでゲソからね)。普通のプログラムではもっとオーバーヘッドは少ないはずでゲソ。

[*27] 作者注: 実際にはこのままだと verify が通らないので-Xverify:none も一緒につけています。これはパッチが Java6 以降の新しいクラスファイルフォーマット用の verify 関数にしか対応してないのに、scalac や clojurec が Java5 という 1 つ古いクラスファイルフォーマットで出力してくれるからです (これは file コマンドで確認できます)。scalac については、ProGuard というツールでクラスファイルをバージョンアップすれば-Xverify:none なしでも通ったのですが、clojurec の方は ProGuard でうまくバージョンアップできないみたいで、今のところ-Xverify:none なしだと上手く動いてません…。

[*28] 作者注: 測定環境は次の通りです。・ CPU : Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz、・ OS : Ubuntu 11.10 (x86-32) 。テスト自体は、上のプログラム例で書いた通り、1G 回の even/odd 相互再帰を 10 セット行いました。また、Java VM の JIT コンパイラは server コンパイラ (C2 コンパイラ) を使っています。

[*29] って、スクルトだと防御力の上昇量は逆に減っちゃたりするでゲソが…。ちなみに、これをスクル「ド」と読み間違えた 2 級神 1 種限定萌えな読者には猛省を促すでゲソ！

[*30] 作者注: 自分のテストプログラムの書き方が悪いだけのような気もしていたり…。一応 type hint はつけてみたんですが…。

9.3.5 今後はどうなるでゲソ？

さて、ここまで読んだ忍耐力あふれる読者諸君は、この TailCall パッチの今後が気になってきた頃じゃなイカ？ 残念ながら TailCall パッチは、最新のソースコード (jdk7-b147) には対応できていないのでゲソ (2011/12/10 現在)。が、2011 年 10 月に開かれた JavaOne 2011 という Java のお祭りでは、次々バージョンの Java に TailCall を取り込むという話が (その他たくさんの話に紛れて一瞬だけ) 出ていたようでゲソ。本当に正式採用になるかどうかはまだ分からないでゲソが、最近の関数型言語の盛り上がり型を見ると可能性は十分あるんじゃなイカ？。というわけで、関数型が Java VM を完全侵略する日はもう間近に迫ってきているのでゲソ。

参考：Scala で「明示的にトランポリンで書いた」even/odd の相互再帰テストプログラム

```
import scala.util.control.TailCalls._

object MutualTailCallTest2 {
  def isEven(n : Int) : TailRec[Boolean] = {
    if (n == 0) done(true)   else tailcall(isOdd(n-1))
  }
  def isOdd(n : Int)  : TailRec[Boolean] = {
    if (n == 0) done(false)  else tailcall(isEven(n-1))
  }

  def main(args : Array[String]) : Unit = {
    var start = System.currentTimeMillis
    for (i <- 1 to 10) {  isEven(1000 * 1000 * 1000).result  }
    var end = System.currentTimeMillis
    print((end - start)/1000.0); println("[sec]")
  }
}
```

参考：Clojure で「明示的にトランポリンで書いた」even/odd の相互再帰テストプログラム

```
(ns mutualtailcalltest2
  (:gen-class))

(declare my-even? my-odd?)
(defn my-even? [^long n]  (if (zero? n) true  #(my-odd?  (unchecked-dec n))))
(defn my-odd?  [^long n]  (if (zero? n) false #(my-even? (unchecked-dec n))))

(defn -main [& args]
  (let [start (System/currentTimeMillis)]
    (do
      (dotimes [_ 10]
        (trampoline my-even? (* 1000 1000 1000)))
      (. System/out print (/ (- (System/currentTimeMillis) start) 1000.0))
      (. System/out println "[sec]"))))
```

付録：諸君らが愛してくれた `OCamlopt` は負けた！ なぜだッ！？ でゲソ

セ^h オキャム　ル「バ、バカな、き、きさまはスカラだろ！？ ち、ちがうのか！？」
スカラ「ちがうな・・・、オレは、超（スーパー）スカラだ！！」（ドヤ顔で）
オキャム　ル「超スカラ・・・！？ な、なんだそれは！？」
スカラ「いちいち説明するのもめんどうだ、てめえでかってに想像しろ」

って、「かってに想像しろ」だとこのコーナーいきなり終わってしまうでゲソ (汗。結論から先に言うと、今回の速度差は単にインライン展開の差でゲソ。
OCamlopt は確かに x86 のネイティブコードを出力してくれる強力なコンパイラでゲソが、最適化は (少なくとも今のところは) あんまり頑張っていないでゲソ。今回も even/odd の相互再帰に対するインライン展開はなかったでゲソ。(つまりプログラムで書いた通りに毎回関数呼び出しが出ていたのでゲソね。一応 -inline オプションも付けてみたけどダメだったでゲソ...。)
それに対して、今回使った HotSpot という Java VM が Scala の even/odd に対して施したインライン展開は実に 10 段。単純計算すると末尾呼び出し時のジャンプのオーバーヘッドが 1/10 になっているわけで (いや、これは単純過ぎるでゲソが) さすがに勝てないでゲソ。
ちなみに読者の中には、(この作者と同じように) Java VM 上では関数呼び出しが全て virtual 呼びなので、ダイナミックディスパッチが必要だしインライン展開も出来なくて遅い、と思ってた人はいないでゲソか？ これは一般には正しいでゲソが、実際に JIT コンパイルする段階で isEven() や isOdd() という名前のメソッドが 1 つしか存在しなければ、飛び先がそこだというのは確定的に明らかなので、ダイナミックディスパッチも不要だしインライン展開もできてしまうでゲソ。Java VM 言語だから関数呼び出しは遅い、とは限らないのでゲソね。(もちろんこういうやり方をすると、JIT 生成した後で isEven() や isOdd() を実装したサブクラスがダイナミックロードされたら JIT をやり直すという手間も出てしまうでゲソが...。)
最後に余談でゲソが、インライン展開の差とだけ言われると GCC なんかが気になる人もいるんじゃなイカ？ 試しに C で even/odd 相互再帰を書いて gcc -O3 でコンパイルしてみると、、、、くぁ w せ drftgy ふじこ lp(結果は各自見てみるでゲソ。ちなみに今回私が使ってみたのは gcc4.6.1 でゲソ)。こういうえげつない行為をいともたやすく行ってしまう、それが D4C、、、ならぬ GCC でゲソね。とはいえ手元の環境では、-O2 だと単に TCO するだけで引数はスタック渡しなので OCamlopt にもボロ負け、さらに-O1 以下では TCO さえしないのでスタックオーバーフロー死と、安定感が全くないのも GCC でゲソ。所詮は C 言語、関数型言語のような過度な期待をしてはいけないのでゲソ。

参考文献

[1] ABELSON, H., DYBVIG, R., HAYNES, C., ROZAS, G., ADAMS, N., FRIEDMAN, D., KOHLBECKER, E., STEELE, G., BARTLEY, D., HALSTEAD, R., OXLEY, D., SUSSMAN, G., BROOKS, G., HANSON, C., PITMAN, K., AND WAND, M. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11* (1998), 7–105. 10.1023/A:1010051815785.

[2] CHIBA, S. Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (Oct. 1998).

[3] JONES, S. P., RAMSEY, N., AND REIG, F. C–: A portable assembly language that supports garbage collection. In *IN INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING* (1999), Springer Verlag, pp. 1–28.

[4] NECULA, G., MCPEAK, S., RAHUL, S., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 209–265.

[5] SCHINZ, M., AND ODERSKY, M. Tail call elimination on the Java virtual machine. In *In Proc. ACM SIGPLAN*

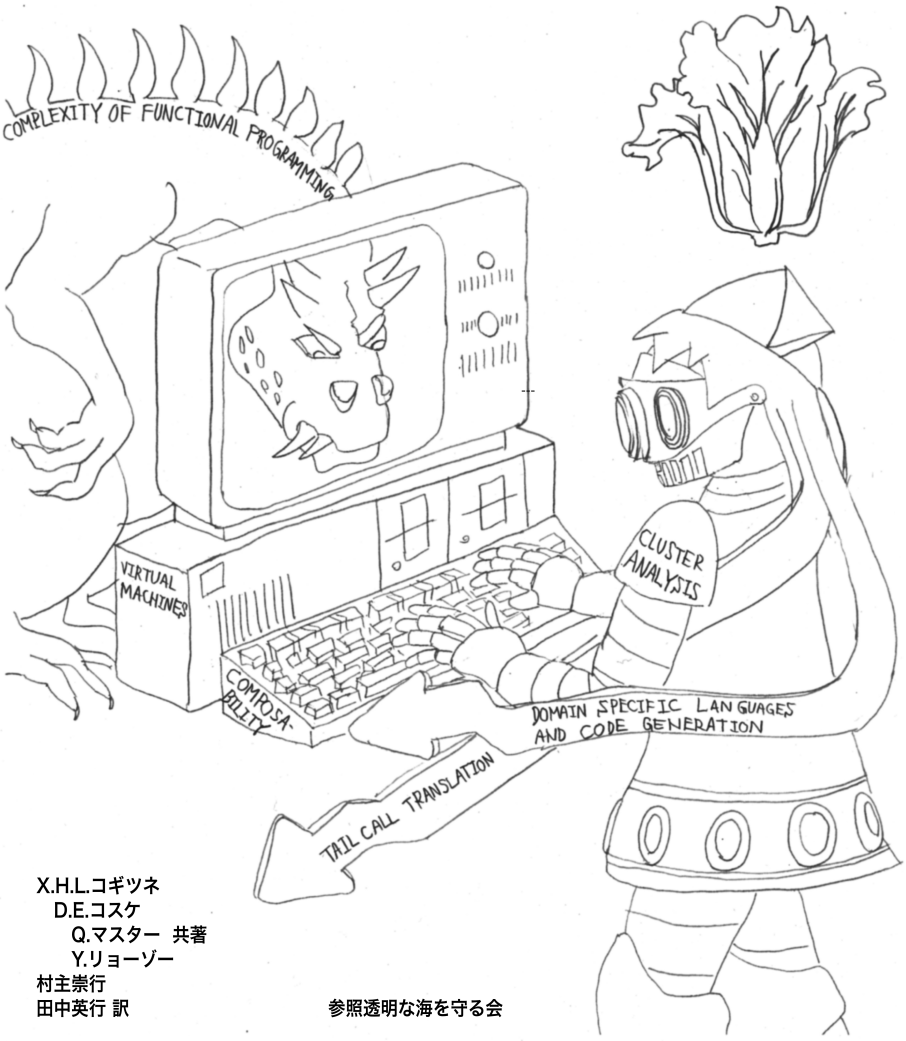
BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability (2001), Elsevier, pp. 155–168.

[6] SCHWAIGHOFER, A. Tail call optimization in the Java HotSpot VM. Master's thesis, Johannes Kepler Universität Linz, 2009.

[7] SERPETTE, B. P., AND SERRANO, M. Compiling Scheme to JVM bytecode: a performance study. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2002), ICFP '02, ACM, pp. 259–270.

イカムスメ II

簡約・簡約・簡約・簡約・簡約・簡約



X.H.L.コギツネ
D.E.コスケ
Q.マスター 共著
Y.リョーゾー
村主崇行
田中英行 訳

参照透明な海を守る会

第10章

関数型イカガール第3話

— @tanakh

この作品はフィクションであり、実在する人物、地名、団体とは一切関係ありません。

---

12 月。冬の海に僕はいた。夏の賑わいは見る影もなく、寂しい海にはもちろん他には誰もいない。海の家は夏の終わりと共に閉店し、あのイカの少女も姿を消した。

それでも僕は週末が来るたびここに足を運ぶ。あの時の胸の高鳴りを忘れることができなくてーー

最近の僕を悩ませているのは、圏論だ。正確には圏論そのものではなく、代数学とプログラミングの関わりについてだ。圏論のことを考えているときはとても楽しくて、それだけで時間を潰してしまうことも少なくないのだけども、プログラミングの時にどことなく圏論の影がちらつく Haskell は勉強を始めた当初は手こずったりしたものだった。それが Haskell、あるいは関数型パラダイムのの敷居を上げているのではないか、少なくとも心理的障壁になっているのではないかと思ってしまう。

「そんなの決まっているじゃないカ！」

突然の声に僕は顔を上げた。そこにはあのイカの少女のくりくりとした大きな目があった。彼女は夕日を背に、僕のノートを覗き込んでいた。いつからそこにいたのだろう。どうやら僕は考え事をすると周りが見えなくなるようだ。しかしそんなことよりも、イカ娘にまた会えたことが嬉しかった。

「それは Composability のためでゲソ」

そんな僕にはお構いなしにイカ娘は続ける。

「Composability はプログラムを自由に組み合わせられる性質でゲソ。プログラムの Composability は、安心してプログラムを書くためにとても重要なことでゲソ。基本的なことじゃないカ」

Composability という概念は聞いたことがないでもなかったが、それがそんなに重要なものだとは考えていなかったわけで。

「コンビネータ、って聞いたことないでゲソか？ 人間の世界では小学校で習うって聞いたでゲソが」

はて、どこの世界の小学校の話だろうか？

「たけるがドラゴンの表紙の分厚い本を教科書に使ってるでゲソ。確か、ドラゴンなんとか[*1]とか...」

どういう小学校[*2]なんだ。それはともかく、コンビネータっていうのは自由変数を含まないラムダ式のことだ。狭義には関数を引数に取って、それを組み合わせて新しい関数を返すような高階関数を指すことが多いように思う。パーザコンビネータ[*3]などは典型的な例で、組み合わせ可能な小さなパーザから目的の関数を組み立てるというものだ。

「それで、Composability というわけでゲソ」

[*1] 「コンパイラ—原理・技法・ツール」通称 “ドラゴンブック”

[*2] Functional Ikamusume Advent Calendar 2011 一日目「たけるの通う小学校に Dragon Book や SystemC 等の専門書籍がてんこもりでヤバイでゲソ!」`http://d.hatena.ne.jp/xiaohuli3/20111201#1322758513` 参照

[*3] Haskell では *Parsec* `http://hackage.haskell.org/package/parsec` 、atto-parsec* `http://hackage.haskell.org/package/attoparsec` などが有名

Composability が大事だという事は分かった。だけど、それがプログラムの代数的構造を考えることにどう関係があるのだろうか。それに、Composability という言葉を気軽に使ってはいるけど、どういう意味かあまりちゃんとは考えてはいなかった。

「じゃあ例として、パーザコンビネータを考えてみようじゃなイカ」

イカ娘はいつの日かそうしたように、頭から長く伸びた青い触手で鮮やかに僕の手からシャープペンを奪い取り、軽やかに紙の上に滑らせ始めた。

```
newtype Parser a
  = Parser { unParser :: String -> [(a, String)] }

empty :: Parser ()
empty = Parser $ \xs -> [((), xs)]

char :: Char -> Parser Char
char c = Parser f where
  f (x:xs) | c == x = [(x, xs)]
  f _ = []
```

「ここでは簡単のためにパーザの型を `String -> [(a, String)]` にしたでゲソ。まあここではどうでもいいことでゲソね。`empty` は空文字を受理するパーザでゲソが、これの詳しい話は後でするでゲソ。`char` が特定の文字を受理するパーザでゲソね。これも特にひねりのない実装でゲソが、実装の話はどっちにしてもここでは気にしない方がいいでゲソ」

ここまではいい。見慣れたパーザコンビネータの定義だ。

「じゃあ、文字列 “ab” を受理するパーザを書いてくれなイカ？」

ふむふむ。簡単な問題だ。`char 'a'` が “a” を受理するパーザで、`char 'b'` が “b” を受理するパーザだから、これらをシーケンシャルにつないだものになる。`char 'a' >> char 'b'` とでも書けるだろうか。

「その >> はどこから来たんでゲソ？」

>> を使うためにはパーザがモナドのインスタンスになっている必要がある。だけど、ここでの Parser 型はモナドのインスタンスにはされていない。

「Parser がモナドのインスタンスになっている必要はないんじゃなイカ？ やりたいことは、Parser をモナドのインスタンスにすることではなくて、2 つの Parser を直列につなぐことでゲソ。例えばーー」

そう言って、再度流れるような動きでペンを走らせる。緩やかにカーブした青い触手は目の前でくねりくねりとうごめき、その怪しい動きは僕を惑わす。

「ちゃんと聞いてるでゲソか？」

ノートには式が完成していた。

```
seq :: Parser a -> Parser b -> Parser (a, b)
seq p q = Parser $ \s -> do
  (a, t) <- unParser p s
  (b, u) <- unParser q t
  return ((a, b), u)
```

「これは 2 つの Parser をつなぐコンビネータでゲソね。実装にはリストモナドが使われているけど、それもやっぱりどうでもいいでゲソ。世の中どうでもいいことが多すぎじゃなイカ？」

それこそどうでもいいんじゃないか、と思いつつ、これを使って文字列 “ab” を受理するパーザを僕は次のように書きなおした。

```
char 'a' `seq` char 'b'
```

「上出来じゃなイカ！」

イカ娘は満足気にフンフンと鼻を鳴らした。あまり大したことをしたつもりはないのだけども。

「じゃあ、今度は文字列 “xyz” を受理するパーザを書いてみてくれなイカ？」

あれ、それはどこが難しいのだろう。2 つのものが 3 つになるだけじゃないのか？

「2 と 3 の間には超えられない壁があるのでゲソ。例えば、この現実の世界と、アニメの世界のように」

二次元の世界に行きたくてもいけないのと、この例題は関係ないのではないか？ 少なくともこのケースは、2 個を 3 個に変えるだけだ。

```
char 'x' `seq` char 'y' `seq` char 'z'
```

「いや、その理屈はおかしいでゲソ」

すっとぼけた顔でイカ娘は言った。

「この式は曖昧でゲソ。関数 seq を自然に演算子として使っているけど、結合性についてはどこでも触れてないじゃなイカ。その式は二通りの解釈があるでゲソ」

そう言って次の 2 つの式を書いた。

```
(char 'x' `seq` char 'y') `seq` char 'z'
char 'x' `seq` (char 'y' `seq` char 'z')
```

はて、この 2 つに違いはあるのだろうか。返り値の方はとりあえず置いておいて、その他に気にする必要があるものだろうか。

「当たり前じゃなイカ。この 2 つは全然違う式でゲソ。次の 2 つは全く違うものじゃなイカ！」

```
(1 - 2) - 3
1 - (2 - 3)
```

それはそうだ、整数の引き算が左結合と右結合で異なる結果になるのは誰でも知っている。だけど、今回の場合は関数だ。あるいは一般化してプログラムだとするべきか。いずれにせよ seq 関数は、プログラムに対する演算子に他ならない。

「seq 関数がプログラムに対する演算子なら、その結合性を気にするのは当然でゲソ」

そりゃそうだ。だけど、今の今まで僕はそのことを気にせず Haskell のプログラムを書いてきた。もちろんモナドを使わずに書いてきたわけではない。パーザコンビネータだってよく使う。でも結合性で問題を起こしたことはない。不可解な動作を起こしたことはない。なぜだろうか。

例えば 3 文字以上のケース、特定の文字列 s を受理するパーザを考えてみる。

```
string :: String -> Parser ()
string s = foldl seq empty $ map char s
```

あるいは次のように書けるだろうか。

```
string :: String -> Parser ()
string s = foldr seq empty $ map char s
```

右結合と左結合が異なるのであれば、これらも違うということになってしまう。どちらを使うべきなのか、fold を書く度に気にするのはめんどうだし、状況に応じて適切なものを使い分けるということがそもそも不可能になってしまう。

「少し意地悪な問題だったでゲソね。本当は、さっきの 2 つの式に違いはないでゲソ。時間がかかるからなんでそうなっているかは示さないでゲソ。気になるんだったら帰ってから式を展開してみるといいでゲソ」

そうなのだ。この 2 つが異なる意味を持ってもらっては困る。そんなことがあっては、おちおちコードを書いていられないじゃないか。

「そうでゲソ。我々は加算が結合律を満たすということを知っているし、減算がそれを満たさないということを知っているんでゲソ。じゃあ他の演算子についてはどうでゲソか？」

数に対する四則演算は息をするようにその性質を意識してコードを書くことができる。それは小さい頃、右も左もわからないまま算数教育としてそのあたりの法則を刷り込まれているからに過ぎない。我々が日々

用いる演算は、プログラミング時のそれと比較するとほんの一部分にすぎないのだ。

「そこで、圏論の登場というわけでゲソ」

深く首を立てに振りながら、イカ娘は勿体ぶってそう言った。

「プログラミングにおいて、特に関数プログラミングにおいてでゲソね。プログラムに対する演算というものを考え始めると、その代数的構造のもつ性質を考えることはとっても重要になるんでゲソ。例えば、さっきのようなーー」

ここまで言われたら、さすがの僕でも察しがつく。seq 関数とパーザコンビネータが構成する代数的構造がどのような性質を持っているのか。今回の場合だと seq 関数が結合律を満たすかどうかだ。

「私が書いた seq の定義は結合律を満たしているでゲソ。だけど、それを示すのは大変だし、大変じゃなかったとしても、演算子を見るたびにそんなことを確認するのは面倒じゃなイカ！」

そりゃあそうだ。でも、それをしなければ安心してコンビネータを使うことはできない。

「それで、そのへんにまつわる問題を一気に解決するためのツールとして、プログラミングに圏論、それから群論などの代数学を持ち込んだのでゲソ！」

そう言って、イカ娘は「ドヤ？」とでも言いたそうな顔でこちらを見つめてきた。

「例えば、モノイドというものがあるでゲソ。モノイドは単位元を持つ半群のことでゲソ。半群っていうのはーー」

半群は僕も知っている。次の 2 つの性質を満たす集合 S と二項演算 “・” の対のことだ。

- $a, b \in S \Rightarrow a \cdot b \in S$ (演算が閉じている)
- $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ (結合律)

「そのとおりでゲソ！ じゃあ、パーザとそれを組み合わせる二項演算が例えばモノイドになっていたとすると、どうなるでゲソ？」

あっ、モノイドだったら、何も考えずに結合律を満たすと見なせるではないか！

「モナドはモノイドに似たもので、クライスリ圏の射に相当するものでゲソが、これも結合律を満たすものでゲソ。これのお陰で細かいことは気にせずに自由に安心してモナドのプログラムを組み立てることができるんでゲソ！」

僕がモナドをあまり深く考えずに簡単に操ることができたのはそのおかげだったのか。しかし僕はまだピンと来ない。なぜ代数学なのか。

「それは、数学の世界の成果をまるごと取り込むことができるからじゃなイカ？」

イカ娘は腕を組み、目を閉じしばし考えに耽っている。頭から伸びた触手が中空を彷徨う。

「我々が対象としたいプログラムと演算子は、既に歴史ある群論や圏論で研究されている代数的構造である可能性が高いでゲソ。とすれば、それに関する数学的な考察がそっくりそのままプログラミングに適用できるでゲソ。それにーー」

なるほど。確かに敢えて独自の構造を再定義する必要はあまりないだろうし、既存のものに合わせたほうが知識も共有できて利便性も高い。

「モナド、モノイド、その他もろもろの借用物を演算子定義の際のガイドラインにする、つまりモナドはモナド則、モノイドはモノイド則を満たすように、プログラムやライブラリを書くみんなが気をつけることによって、私たちはモナドという型クラスの制約を自分のコードに課すだけで、何も心配せずに、安心して自由にコードを抽象化することができるようになるのでゲソ。これは計り知れないメリットでゲソ！」

ここまで聞いてようやく僕も納得がいった。自分の型を、既によく知られたプロパティを持つクラスのインスタンスにすることによって、細かいことを考えずとも自分の型が優れた代数的性質を持つようにすることができるのだ。その最たるものは、プログラムの Composability だったのだ。

「プログラミングで一番大事なことは Composability でゲソ。小さいプログラムから大きなプログラムが矛盾せずに作れるということは、プログラムの抽象化を自由に行えるということでゲソ。逆にそうじゃなければ、どこかに分割できないブロックが発生するということでゲソ。Composability さえしっかりしていれば、プログラムの妙な挙動で驚くことはなくなるでゲソ。これがホントの驚き最小の原則じゃなイカ！ アッ、と驚くタメゴロー」

フンフンフン、と鼻を鳴らしながら満足気にイカの少女は去っていった。一人残された僕は夕暮れの海を眺め、しばし呆然としていた。

帰りの電車で、僕はノート PC を開きながら、パーザについてもう少し考えていた。パーザコンビネータで本質的なものはもうひとつある。それは選択だ。Parsec の新しいバージョン、Parsec3 を見てみると、選択の演算子は Alternative クラス[*4]を実装した<|> になっていた。

Parsec3 には他にも、`many` や、`some` 、`manyTill` などのコンビネータが定義されているが、これらのコンビネータは実は<|> だけを用いて定義できることがわかった。これはパーザ固有のものではなかったのだ。Alternative のインターフェースに合わせれば、ただひとつの演算子を定義するだけで、様々な便利な関数がノーコストで使用可能になる。

そこで僕は気がついた。プログラムの抽象化、関数、型、オブジェクト指向におけるカプセル化、モジュール化、云々。これらはすべて Composability を高めるために行われることだったのだ。カプセル化は、モジュール間の依存関係を少なくすることによって、確率的に複数のモジュールが干渉する可能性を下げようとしていることに他ならない。圏論によるアプローチは、数学のフレームワークを利用することによって、依存関係を減らさずに Composability を確保するというある意味スマートな、でもある意味では強引なアプローチだったのだ。

「次は、＊＊＊、＊＊＊」

電車が駅に到着した。考え事をしていると時間が立つのが速いものだ。

「ダァシエリイェス！！」

どこか気怠い車掌のアナウンスを背に、僕は電車を降りた。今まで書いてきたプログラムを、書き直したくて手が疼く。あれもこれも、きれいに書き直せる。そんな気がして、それを早く確かめたかった。そんな想いを胸に、僕は足早に家路についた。

[*4] Control.Applicative に含まれる Functor クラスのサブクラス。標準ライブラリに含まれる。

第11章

映画けいさん！

— @tanakh

この作品は某映画を見たいけどなかなか見に行けない私の妄想の産物、つまりフィクションであり、実在する人物、地名、団体とは一切関係ありません。

「唯先輩、起きてください」
「むにゃむにゃ、あと 1 時間だけ……」
「ダメです。そんな時間ありませんよ」
「むにゃむにゃ、ハスにゃん。もう食べられないよ」
「何寝ぼけてるんですか。着きましたよ、ロンドン」
「……あ、おはよう、ハスにゃん」

私たちは今、ラムダ高計算機プログラミング競技研究部、通称「計算部」の課外活動としてイギリスに来ています。

飛行機の中でずっと私にもたれかかって寝ていたのが、平無駄唯先輩。いつもはとぼけているように見えて、今私たちがここに来れているのも唯先輩のおかげだったりします。

「ロンドンだー、ロンドンだー」
「ヒースローだー！」

あそこで唯先輩と一緒にはしゃいでいるのが、排中律先輩。2 年前に潰れかけだった計算部を復活させた人で、計算部の部長です。普段はいい加減な人ですが、いざというときは…やっぱり頼りないかも。

「おい、澪。何してんだ。早く荷物取ってこいよ」
「ううー…飛行機怖い……」
「飛行機はもう降りただろ。おい、しっかりしろー」

律先輩の隣で青い顔をしているのが、入文字山澪先輩。計算部で唯一の常識人で、実力もトップクラス。私のあこがれの先輩です。怖がりなのが玉に瑕かな。

「あらあら、ハスさちゃんは常識人じゃないのかしら？」
「あっ、モナ先輩。なんで私の考えてることを！？」

そしてこれが、型吹モナド先輩。おっとりしていて、すごいお嬢様なんだけど、ちょっと何を考えているのかわからない所があったりします。

「はあ、こんなので大丈夫なのかなあ」

海外に来ても、計算部は相変わらずです。

「ハスにゃん、すごいよ！ 白人がいっぱいいるよ！」
「もう、唯先輩。ここイギリスなんですから、当たり前じゃないですか。って、どこ行ってるんですか！？」
「えっ、こっちじゃないの？」
「乗り継ぎですよ！ ヒースローで降りちゃダメです」

はあ、こんなので大丈夫なのかなあ。

「うわあ、ここがグラスゴーかあ」
「すごい、すごいよハスにゃん！　ヨーロッパだよ！」
「イギリスは毎年来てますけど、今年が一番ワクワクします」
「ちくしょう、お嬢様は違うなあ」

グラスゴーはスコットランド最大の都市で、イギリスで 4 番目の人口を誇ります。イギリス国内では、ロンドンとエディンバラに次いで 3 番目に観客が多いそうです。ハリーポッターが書かれた街として有名なエディンバラからは列車で 1 時間程度。中世の時代から多数の文化が生み出され、産業革命以降は工業都市として栄えたそうですが、今では古風な街並みを残した落ち着いた街です。

私たち計算部は “International Computer Programming Contest”、通称 ICPC の世界大会に出場するためにグラスゴーに来ています。ICPC は毎年いろいろな国で開催されていますが、今年はイギリスのグラスゴーにあるグラスゴー大学が主催なのです。

グラスゴー大学は 15 世紀に設立された名門大学で、英語圏では最古の、とても歴史ある大学だそうです。折しもグラスゴー大学は敬愛するサイモン・ペイトン・ジョーンズ先生が教授として在籍していた所で、私は楽しみのあまり、数日前から満足に寝付けなくてーー。

「おーい、ハスさ、おいてくぞー」
「ハスにゃん、よだれ垂れてる。かわいい」
「むにゃむにゃ、あっ、先輩。待ってくださいよー」

でも、そんなことは先輩方にはお構いなしなようです。そんなこんなで、空港からのタクシーがホテルに到着です。いよいよ私たちの戦いが始まります。

ICPC は世界中から数万人が参加する、数あるプログラミングのコンテストの中でも最も歴史の長く、規模の大きい大会です。制限時間内に正しく動作するプログラムを、なるべくたくさん書いたら勝ちという、単純明快なルールです。国ごとの予選を勝ち抜き、さらに世界 5 つのエリアで行われる予選で上位に入った、わずか 100 チーム弱の強豪たちがここ世界大会の舞台に集うのです。ちなみに 1 チームは 3 人なので、じゃんけんに負けた律先輩とモナ先輩は補欠になりました。

「おお、すごい」
「あっ、看板があるよ。写真撮ろう！　写真撮ろう！」

”International Computer Programming Contest World Finals 20xx” と書かれた看板がホテルから入ってすぐのところにおいてあります。会場はとても大きなホテルで、不覚にもそわそわしてしまいそうです。顎足つきじゃなければ、とてもこんなところに泊まれないだろうなあって。

「ハスさちゃん、真ん中寄って」
「あ、お姉さん写真お願いします」
「おい唯、ここは日本じゃないぞ、ちゃんと英語で頼まないとだなあ。あ、えー、ぷりーず、えーと、ふぉと、ぷりーず」
「写真？　いいわよ。はい、みんなもっと真ん中寄って」

えっ、日本語？

「ありゃ、日本人の方ですか？　ああっ、よく見たらさわちゃん先生じゃないかー！」
「本当だ、さわちゃんだ。変な格好してるから気付かなかったよ」

さわ子先生、部室の押し入れの写真みたいな、本格的に変な格好してる……。ちょっと近寄りがたいような。でも、昔はさわ子先生もプログラミングに青春賭けてたって本当だったんだ。

「もう、失礼ね。本場なんだからそれなりの格好して行くわよ。それより排中さん。ちゃんと英語勉強しておくようにって言わなかったかしら？」
「あはははは。それよりさわちゃんなんでここに？　さては私たちに便乗してイギリス旅行とか？」
「何言ってるのよ、私、あなた達のコーチじゃない」
「あっ、そういえばそうだった」
「もう、あなたたちそんなので大丈夫なの？」

先生。私も同感です。

「うわあ、枕が。ハスにゃん、枕がいっぱいあるよ！」

唯先輩たっての希望で、私は唯先輩と同じ部屋になりました。部屋の鍵を開けた途端、唯先輩はベッドに向かってダイブしていきました。もう、子供だなあ。でも、すごい部屋です。広い部屋にキングサイズのベッドが 2 つ。そして、枕がひい、ふう、みい、よお...

「先輩、首痛くないんですか？」

「ふご、ふごご！ は、ハスにゃん助けて！」

先輩は枕を全部重ねてそこに寝ようとしたみたいです。首が変な方向に曲がっているような...。

「ふごごご！ ハスにゃん、はやく...」

「もう、なにしてるんですか」

唯先輩の首に手をかけて、起こしてあげます。あ、唯先輩、良い匂いがする...。

「ありがとうハスにゃん。死ぬかと思ったよお」

「こんなところで死なないでください。唯先輩、うちの頼みの綱なんですから」

唯先輩は、普段はこんな感じなのに、一度プログラミングを始めると別人になるんです。文字通り別人になるわけではなくて、見た目も、口調も、雰囲気もそのままなのに、淡々ととんでもないことをやり遂げてしまうんです。

去年の夏の合宿の時でした。

「gdb?」

「デバッガですよ。唯先輩、使ったことないんですか？」

「デバッガ？ デバッガって何？」

「デバッガっていうのは、プログラムのバグを取るツールですよ。唯先輩、printf デバッグしかしたことないんですか？」

「printf デバッグ？ それより、バグって何？」

「えっ、唯先輩。プログラムが思い通りに動かないときどうしてたんですか？」

「あれえ、私プログラム書いてて思ったとおりに動かなかったことなんてないよ？」

「そんなことあるわけ...。ちょっとソース書いてみてください！」

「じゃあちょっと書いてみるね。ほい」

カチャカチャカチャ...規則正しくタイピング音が響いて、淀みなくコードが出来上がっていきます。私はペアプログラミングするつもりで後ろから先輩がコーディングする様子を見ていました。でも、疑わしいコードを見つけることもなくて。

「できたよハスにゃん」

「え、もうですか？ じゃあサンプル通してみましょうか」

「ええ、めんどくさいよお。サブミットしちゃおうよ」

「そんな、コンパイルも通さずにサブミットだなんて無茶な...って、ええっ？」

躊躇うこともなく唯先輩はオンラインジャッジサイトに今書いたコードを投稿してしまいました。そして、それがそのままアクセプトされ...。

神様は不公平です。いくら練習しても私は唯先輩のようにコードを書くことはできません。そのことを考えて、私は何度計算部をやめようかと思ったかわかりません。でも、結局辞めることができなくて、それでここにいるわけでーー。

「よしっ、と。そろそろ出発するよ、ハスにゃんも早く」

「ああっ、待ってくださいよお。今準備します」

ちょっと考えこんでしまっていたようです。でもそんなことを考えるのは、今は必要ないよね。

「遅いぞ、唯」
「ごめん、澪ちゃん。みんな早いね」
「お前が遅いんだよ」
「律っちゃん。女の子には色いろあるのだよ。」
「なるほど、そうかー。って、私も女の子じゃい！」
先輩たち仲いいなあ。でも、来年にはみんな別々の大学に...。いつも通りにバカをやって、バカなことを言ってる先輩達を見ていると、これが一緒のチームとして参加する最後の大会なんだってことを忘れてしまいそうで。
「みなさーん。そろそろチーム登録行きませんか？」
「お、そうだな。ちょうど列も空いてきたみたいだし」

ICPC 最初のイベントが参加登録です。あ、参加登録ってバカにしちゃいけないのです。参加登録を正しく行わないと、出場権を失ってしまいますし、それに参加者が一堂に会する参加登録は、他の参加者との最初の顔合わせの場でもあるんです。
「うわあ、人がいっぱいいるよ。これみんなプログラマなんだね」
「しかも地区予選を突破してきた強者揃いだぞ」
「わ、私怖くなってきた……」
「お、おい澪、しっかりしろ―」
「そうだよ澪ちゃん、こういう時は堂々としてないと。ハロー、ウィーアー、ホカゴ tee タイム、フロムジャパン。イエァ、イエァ...」
唯先輩、登録始めちゃったけど、大丈夫かなあ。なんか喋ってるけど、あんまりよく聞き取れないや。私も英語苦手なんだった。
「みんなー。登録できたよ。はい、これ」
唯先輩の手には手提げ袋と、変な怪獣みたいなぬいぐるみ。これは参加賞みたいなものなのかな。それからもう一つ、チームの T シャツ。
「うおお、テンション上がってきた！」
「律っちゃんずるい、私も早く着たいのに」
T シャツは ICPC でのユニフォームのような存在です。私たちのチームの T シャツには、表面に”HTT” というチームのロゴが入っています。コーチのさわ子先生がデザインしたものです。
「無事登録は済んだみたいね。みんなお疲れ様。T シャツよく似合ってるわよ。さすが私の見立てね」
ちょうどさわ子先生がやって来ました。さわ子先生は T シャツを見ると満足気に帰っていきました。
「さわ子先生、何しに来たんだろう...」
「さあ...」
「先生もいそがしいんじゃないか？ 男探しとかで」
ひどい言われようです。

「……唯先輩、起きてください」
「あと 1 時間だけ...むにゃむにゃ」
「そんな時間ありませんよ。遅れちゃいます」
「むにゃむにゃ、もう食べられないよ」
「唯先輩、また寝ぼけてるんですか？」
「……あ、おはよう、ハスにゃん」

グラスゴー 2 日目です。時差ボケと緊張と興奮で、私はほとんど眠れませんでした。でも、そういう苦労も唯先輩には無縁のようです。
「はー、よく寝た。ハスにゃん、今日もがんばろうね！」

「本番は明後日ですよ、先輩。それまで体力残しといてくださいよ」
「えー、観光のほうが楽しみだよう。初めての海外なんだよ？」
「なんというか、とても先輩らしいです」
そうだ。この人はこういう人だ。むしろこうでなければいけないんだ。こういうある意味で異常な空間で、いつも通りの自分を保てるというのはとてもすごいことでもあるんです。そういうところも羨ましくて、すぐに舞い上がってしまう自分が何となく情けない感じもして。
「さ、ハスにゃん。そろそろ出発するよ」
「あ、待ってくださいよお」
唯先輩はもう準備を終えていたようです。私はカバンに水と名札を突っ込んで、急いで部屋の入口に行き、
「やってやるです！！」
気合を入れ直しました。私も負けていられません。

ICPC は競技自体は 5 時間程しかありませんが、全体で 5 日ほどの日程が組まれています。全世界から参加者が集うので、時差ボケの緩和の意味があるんだと思います。現に私は睡眠時間が不規則でふらふらです。その間にエクスカーションとして Tech Talk や観光が組み込まれています。この色々なイベントも ICPC の魅力だったりします。
Tech Talk は、主催の IBN による最新の技術紹介など、私たちプログラマにとってはとても刺激的な内容なのですが、当たり前だけど、全部英語で行われます。朦朧とした頭には、この淡々と流れる英語がとてもいい子守唄になるわけでーー。
「ハスにゃん、ハスにゃん。起きて」
「……うーん、ハッ！　ここはどこ！？」
気づくと、周りには誰もいなくなっていました。
「ハスにゃん気持ちよさそうに眠ってたから……。もう Tech Talk 終わっちゃったよ」
「そんなあ…。はあ……」
「あとで内容教えてあげるよ」
「め、面目ないです」
こういう時、唯先輩がいてくれてよかったなあと思います。

「ぷはー、食った食った」
「律、はしたないぞ」
「いやー、失敬失敬」
午後は、グラスゴー市内の観光がありました。それから、ウェルカムレセプションです。
唯先輩は終始はしゃいでいました。澪先輩は英語で話しかけられる度に青ざめていました。律先輩はひたすら食べていました。モナ先輩は完全に周りに溶け込んでいました。私はーー。
「ハスにゃん、こっちこっち」
「えっ、唯先輩」
唯先輩、外国のチームの人達と話してたみたいです。私を強引にそっちに連れていこうとしています。
「これがさっき話してたハスにゃん」
「えっ、えっ」
「とっても可愛いでしょ。私ハスにゃん大好きなんだ」
「Oh… you’re so cute!」
わわっ、いきなり大変なところに連れてこられちゃったみたいです。ええい、なんとでもなれです！
「さ、サンキュー」
「What’s your favorite programming language? … 」
「あっ、えーと…」

「ふう、疲れたです」
あれからしばらく唯先輩に連れられて、外国チームの人達と話していました。
「ハスにゃん意外と英語喋れたんだね。私びっくりしちゃった」
「もう、そう思うんだったら、無茶させないで下さいよ」
「だってハスにゃん、あんまり楽しそうじゃないみたいだったから。何事も経験だよ、ハスにゃん」
ほんと破天荒な人です。巻き込まれる身にもなってほしいです。でも、ちょっと楽しかったな……。

「唯先輩、起きて下さい」
「むにゃむにゃ、アイキャノットイートエニモワー」
「唯先輩、英語で寝ボケてないで、早く起きて下さい」
「……はぅっ、あ、ハスにゃん。おはよう」
唯先輩は通常運転です。私もホテルの枕にも少しずつ慣れてきました。

今日は、本番前日です。つまり、明日は本番なのです。と言うことは今日は本番前日で、明日の夜にはもう結果が出ているわけで、ここにこうして集まって和やかに談笑している人たちも明日の夜には明暗分かれて悲喜こもごもで、つまり今日は本番前日で……。
「わ、ハスにゃん、どうしたの？」
「あ、先輩。すみません」
緊張して頭の中がぐるぐるです。目眩で足元がぐらつきました。
「おい澪ー、しっかりしろ―」
「あ、明日…もう明日……」
「おいー、帰って来い―」
澪先輩も通常運転のようです。
今日は予行練習があります。明日実際に用いる機材のチェック、システムが正しく動作するか、ソフトウェア環境の確認。当日焦らないために、今日しっかりチェックしないといけません。
「あれえ、ログイン出来ないよ」
「唯、ちゃんとパスワード入れたのか？」
「ちゃんと入れたよお。何度も確認したし」
「ちょっとキーボード貸してみろよ。ええと、ユーザ名 team86、パスワード xxxxxxxx と。ありゃ？」
「ほら、入れないでしょ？ 澪ちゃん疑り深い」
「うーむ、なぜ入れない」
いきなりトラブルみたいです。前日でよかったです。
「係の人に聞いてみました」
「おお、さすがモナ！」
「パスワード、ダブルクオート込で入力しないといけないみたいです」
「ええっ、って、なんでやねん！」
たしかに、パスワードが印刷された紙を見てみると、ユーザ名はダブルクオートで囲われていないのにパスワードは囲われています。
「これはひどい」
「いやあ、こういうの有りがちみたいだぞ？ 実際に何年か前の某大会でも同じようなことがあったってどっかのブログの参加記で読んだし」
「へえ、そうなんだあ。こういう大会でも案外ミスがあるもんなんだね。あっ、ログインできたよぉ」
「よし。じゃあ適当に練習問題サブミットするか」
「アイアイサー！」
唯先輩はいつもどおり、流れるような手つきでかたかたコードを入力していきます。なめらかに動く指の動きに私は魅入られていました。規則的で不規則な怪しい動きは、休むことなくコードを生成し続け、私はそれを眺めるうちに、いつしか眠りの淵にーー

「おっしゃー、トップとったどー」
「とったどー」
「おい、リハーサルでトップとっても意味ないだろう」
いつのまにやら練習問題を解き終わっていたようです。
「写真とっとこー！」
「ハスにゃんもおいでよ」
唯先輩と律先輩がはしゃぎながらスコアボードの写真をとっています。はあ、まったく子供みたいなんだから。でも、せっかくだから、私もちょっとだけ。
「待ってくださいよお、先輩！」

夜、ホテルのボールルームで親睦を深めるためのイベントがありました。巨大なチェスや、様々なボードゲーム。技術ネタを話すスペース、それと PC。
「チェックメイト！！」
あ、唯先輩がチェスをやっています。しかも勝ってる。
「あ、ハスにゃん。来てたんだ」
「唯先輩、チェスもできたんですね」
「え、やったことなかったよ。ルールぐらいは知ってたけどね」
「えっ…」
「私一度やってみたかったんだ。あの大きなチェスボードで」
「そう…ですか……」
唯先輩は以前部で将棋のルーチン書いてたから、チェスができても不思議は…って、やっぱり納得できない！
「ん、どうしたのハスにゃん。眠いなら寝てきていいよ」
「はい、そうします。明日も早いですし。先輩も早く寝てくださいね」
「ハスにゃん、おやすみー」
明日は本番です。万全の調子で望まなければ。

「起きて、時間だよ。遅れるよ」
「うーん、むにゃむにゃ、あと 5 分だけ……」
「ダメだよ、もう時間ないよ。起きてハスにゃん」
「むにゃむにゃ、唯先輩、んっ、そこは…」
「寝ぼけてないで起きて、ハスにゃん！」
「……あっ、おはようございます唯先輩」
しまった、この日に限って寝坊しちゃった！

結局緊張して昨日はほとんど眠れませんでした。朝ごはんを食べる時間ももうなさそうです。
「ううん、すいません、先輩」
「仕方ないよ、ハスにゃん。それより急いで」
急いで名札と T シャツを身につけて、会場へと向かいました。
「遅い！ 何やってたんだ」
「ごめん、律っちゃん」
「すみません、私が寝坊しちゃって……」
「いいから、入場ゲートに向かって！ そっちよ。あと、電子機器は持ち込めないから、持ってたらここに入れて」
「おい、澪もしっかりしろ」
「あ……ここ、は……あ…こわい……コンパイラこわい…」

「おーい、帰ってこーい」
「しっかりして下さい、澪先輩」
澪先輩に肩を貸して何とか会場入りします。
「頑張ってこい！ 綺麗な色のメダルを期待してるぜ」
「がんばってね、私たちの分も」
そうだ、これは私だけの戦いじゃない。参加できなかった、律先輩とモナ先輩の分も、私達ががんばらなきゃ！

ふう、はあ、ふう、はあ。
「何やってるの、ハスにゃん」
「腹式呼吸です。集中力を高めているんです」
「ひ、ひ、ふうー。ひ、ひ、ふうー」
澪先輩、それ違います。
「ばりばり」
「うわあ、何やってるんですか唯先輩。やめてください」
「何って、問題の封筒を」
「封筒の面を見てください！」
「“Do Not Open Until Told To Do So.” … ？ あっ、しまったー」
「とっ、とりあえずしまって下さい！」
唯先輩、やっぱり通常運転です。
「あぁ…あと 5 分…あと 5 分……」
澪先輩も…。本番開始まで、あと 5 分…。

「10, 9, 8, 7, 6, 5, 4, 3, 2, 1, … スタート！」
いよいよ戦いの火蓋が切って落とされました。5 時間の長くて短い戦いが始まったのです。
「唯先輩！ 封筒、開けて下さい！」
「むにゃむにゃ…、はっ。あっ、開けるよっ！！」
封筒の中には綴じられた問題の紙が 3 部。1 人に 1 部というわけです。
「澪ちゃんとハスにゃん、問題読んで！ 一番簡単そうな問題を見つけといて！」
「わかった！」「わかりました！」
うちのチームの戦略はシンプルです。私と澪先輩が問題を読んで、簡単そうな問題から唯先輩に渡していって、唯先輩はひたすらコードを書いてサブミットしていきます。唯先輩は最初の内に、共有コードを打ち込んでおきます。ICPC ではコードの持ち込みが禁止されているからです。
「これ！ 素数を列挙すれば簡単に解ける問題だぞ」
「サンキュー、澪ちゃん」
グッジョブです、澪先輩！

「あれ、通らないよお。たすけてハスにゃん」
どうしたんだろう、唯先輩。今までこんなことはなかったのに。
「どういうことですか、唯先輩。って、TLE ですか？」
唯先輩はなぜだか Haskell を好んで使っています。コードをほとんど間違えないというのも、デバッガを使わないというのも、その恩恵によるところが大きいようです。初めてその様子を見たときはとても驚いたものですが、でも今回はそれが仇になってしまいました。
「ちょっとコード見せて下さい」
私も Haskell はちょっとは読めます。なんとかしないと。最終的には C++ に書き換えることも辞さない構えで。

「あ、これ、ブログで見た問題だ...」
唯先輩の書いた素数列挙のコードは次のようになっていました。

```
primes = 2 : sieve [3, 5 ..] where
  sieve (p:xs) = p : sieve [x | x <- xs, x `rem` p /= 0]
```

よく Haskell できれいに書ける例として登場するコードですが、これはダメです。エラトステネスの篩になってないんです[*1]。ターナーの篩というやつなんです[*2]。

```
primesTo m = 2 : eratos [3, 5 .. m] where
  eratos []     = []
  eratos (p:xs) = p : eratos (xs `minus` [p, p+2*p..m])
```

コードを少し変えて、こうするだけで大分速くなります。
「唯先輩！ これを！」
「あ、ありがとうハスにゃん！」
このコードは速くなるけど、C 等による実装に比べるとまだはるかに遅いです。これがダメだったら、ST モナド[*3]で普通に篩を実装するしかない...。
「通った！」「やりましたね、先輩！」
よかった。杞憂に終わった。私は問題読みに戻ります。

「先輩、これ簡単な DP です」
「ありがと、ハスにゃん！」
先輩に問題と解き方を伝え、託します。しかし、唯先輩の手が動きません。
「唯先輩、どうしたんですか？」
「ハスにゃん.....私 DP 書いたことない」
「えっ...」
練習しといてって言ったのに.....。
「Haskell で DP するには、メモ化ですよ、メモ化」
「ああっ、どうやってメモ化すればいいか、すぐに思いつかないよお」
「メモ化を Haskell でやるには幾つか方法があります。Purely なテーブルを持ちまわる方法、遅延無限テーブルを使う方法、そして、モナドを使う方法です。一番手軽で簡単で高速なのは、モナドを使う方法だと思います！」
「分かった！ ハスにゃん、ペアプロしてくれる？」
「任せて下さい。やってやるです！」
メモ化のモナドとしては MonadMemo[*4]を用いるのが楽です。

```
class Monad m => MonadMemo k v m | m -> k, m -> v where
  memo :: (k -> m v) -> k -> m v
```

MonadMemo の型パラメータは 3 つです。キーの型 k、値の型 v、そして、メモモナド型 m。具体的な型としていろいろなコンテナとモナドをバックエンドに使えるのです。
例えば、フィボナッチのメモ化ならこんな感じになります。

```
fib :: Integer -> Integer
fib 0 = 0
```

[*1] http://www.haskell.org/haskellwiki/Prime_numbers に詳しい
[*2] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes を参照
[*3] メモリを読み書きするためのモナド
[*4] http://hackage.haskell.org/package/monad-memo

```
fib 1 = 1
fib n = fib (n - 1) + fib (n - 1)
```

これが元のコードで、

```
fib :: Integer -> Memo Integer Integer Integer
fib 0 = return 0
fib 1 = return 1
fib n = do
  a <- memo fib (n - 1)
  b <- memo fib (n - 2)
  return (a + b)
```

こうなります。とてもシンプルです。Applicative を使えば、もっとすっきりとした記述になります。

```
fib :: Integer -> Memo Integer Integer Integer
fib 0 = return 0
fib 1 = return 1
fib n = (+) <$> memo fib (n - 1) <*> memo fib (n - 2)
```

MonadMemo の使い方を教えると、唯先輩の指は再び滑らかに動き始めました。
「できた、できたよハスにゃん！」
「やりましたね、先輩！」

「澪ちゃん、ハスにゃん。次の問題は？」
「うーん、ちょっと解き方がわかるのが無いな」
「私もです……」
膠着状態に陥ってしまいました。でも、澪先輩があんなに考えても分からない問題がそんなにたくさんあるものかなあ。
「あっ、この問題、探索でいいんじゃないんでしょうか？」
「うーん、でもちょっと入力値が大きくて自信が持てない」
「澪先輩が分からないということが何よりの状況証拠ですよ！」
「あ、ハスにゃん頭いい」
「おいおい、どうなっても知らないぞ」
「澪ちゃんは他の問題考えといて！ ハスにゃんは私とペアプロ！」
「了解です！」
探索と決まればあとは速いです。唯先輩の淀みないコーディングは瞬く間にコードを完成させました。
「いくよ、ハスにゃん！」
「どんとこいです！！」
勢い良くテストを走らせてみるものの、結果が帰って来ません。
「やっぱりダメだったかなあ……」
「あ、答え出ましたよ。でもちょっと遅いですね」
「うーん、困ったなあ」
「そうだ、並列化ですよ！」
ジャッジマシンは、Corei7 の 4 コアマシンです。4 倍早くなれば、通る可能性は無きにしもあらずです。
「は、ハスにゃん、私並列のコード書いたこと無い……」
「さあ、早く…って、またですか！？」
この人は準備とか練習とか言う物をしないのかなあ。
「もう、仕方がないですね。こんな事もあろうかと、私が並列化ライブラリ書いてきました。使って下さい」
「あ、ありがとう。ハスにゃん！」

私は普段は Haskell を使っていませんが、Haskell で並列コードを書くのは比較的簡単です。特に ICPC では複数テストケースがあるので、ワーカスレッドを立てて、あとはタスクキューを作ればそれらの制御は簡単です。

```
main :: IO ()
main = do
  q <- newChan
  r <- newChan
  n <- readLn

  forkIO $ forever $
    forM_ [1..] $ \ix -> do
      p <- getProblem
      writeChan q (ix, p)

  forM_ [1..4] $ \i -> forkIO $ do
    (ix, p) <- readChan q
    writeChan r (ix, solve p)

  results <- getChanContents r
  forM_ [1..n] $ \ix ->
    putStrLn $ fromJust $ lookup ix results
```

これを使って、唯先輩がごりごりとコードを書きなおしていきます。
「できた！」
「サブミットしましょう！」
しばしの沈黙、そして。
「通った！」
よかった、何とかなった。

その後、残った問題は解き方も分からず、膠着したまま、時間は過ぎて...。
「9, 8, 7, 6, 5, 4, 3, 2, 1, The Contest is Over!!」
私たちは苦戦しつつも、それなりに問題を解くことが出来ました。スコアボードはラスト 1 時間でフリーズされるので、最終的な順位はまだわかりません。問題数的には、メダル圏内に手が届かないとも...。
「ふわあ、疲れた」
唯先輩は机に突っ伏しています。
「.....」
澪先輩は白目を剥いてピクリとも動きません。お疲れ様です。ゆっくり休んで下さい。
「おーい、どうだったかー」
「3 人とも、お疲れ様」
「あ、律先輩、モナ先輩。ありがとうございます」
そういう私ももう疲労がピークです。
「お部屋に帰って、休憩したほうがいいんじゃないかしら」
「モナ先輩...。それじゃあお言葉に甘えて」
ふらふらと部屋に戻り、私は泥のように眠りました。

「おはよう、ハスにゃん」
「おはようございます、唯先輩」
今日は日本に帰る日です。結局私たちはメダルを取ることはできませんでした。時間差で勝てなかったのです。私は思わず泣いてしまいました。先輩たちに有終の美を飾らせてあげられなかったことが悲しくて、悔しくて。
「大丈夫だよ、ハスにゃん」
「……唯先輩？」
「みんなと一緒に頑張れて、私は楽しかったよ。それにーーー」
「それに？」
「計算部は、勝つことだけが目的じゃないよ。みんなで楽しくプログラミングするのが計算部なんだよ」
「そうよ、ハスさちゃん」
「私もハスさとプログラムするの、楽しかったぞ」
「私は補欠だったけど、旅行できて楽しかったかな」
「み、みなさん……」
みんな、いい人達だなあ。この人達と同じ部で活動できて良かった。
「そうだ、ハスにゃん、これ」
そう言って、唯先輩が SD カードを差し出しました。
「なんですか？ これ」
「計算部みんなから、ハスにゃんに。ほら私達、今年で引退だから」
「あとで開けてみて。大したものじゃないのだけど」
「あ、ありがとうございます…」
私はまた泣いてしまいました。

帰りの飛行機の中。SD カードが気になって、みんなが眠っている間に、こっそりノート PC で中身を見てしまいました。
SD カードの中には、「ToHsNyan.hs」というファイルがひとつだけありました。

```
main = "keisanbu" + "hsnyan" where
  "keisanbu" + "hsnyan" = forever $ do
    putStrLn "nakama da yo!"
```

これは……。みなさん、直接言ってくれればいいのに、恥ずかしがり屋さんなんだからーー。
でも、なんだかすごいコードだなあ。これほんとに動くのかな？

```
$ runhaskell ToHsNyan.hs
```

次の瞬間、端末が文字で埋め尽くされました。
「あっ、しまった。止まれ、止まって！」
カチャカチャカチャカチャ。私が Ctrl-C を連打する音が暗い飛行機の中を響き渡ってーー。
「……ふわあ、ハスにゃん。おはよう」

第III部

簡約!? 入力娘 (算) (2012年夏)

前回までのあらすじ

――『それ』は、人類がプログラミングを始めて以来、世界の各地にてたびたび観測されてきたものであったが、その年の国際学会にて正式に議題として取り上げられ、限りなく仕様の逸脱に近い仕様として、公式に処理されることで一致した。
プロジェクト維持のため、総員が協調し対処にあたるべき認定特異災害『バグ』。
バグはプログラムに潜み、プログラムに接触することで、ゴミへと変えて分解してしまう。対して、ヒトの行使する通常言語にのっとったデバッグ力は、いかに最新・先鋭を誇っていようと、バグには微々たる効果しか発揮できず、ヒトは往々にして、ただ納期が通り過ぎ、ソフトウェアが納品先へと姿を消すのを待つだけでしかなかった。
ある者は、そんなバグを「まさに災害だ」と評し、またある者は、プロジェクトを襲い、デスマ案件へと変え、やがてプログラマ自身も練炭の塊と崩れ落ちるその様に、「他人を巻き込む自殺願望そのものだな」と吐き捨てるのであった。
物語の舞台は、近未来の日本。日本政府は、公に出来ない暴力装置をいくつかかかえている。神保町にある国立情報学研究所機動部二課は、日露戦争時に英国図書館の指導のもと組織された特務機関を前身としており、一般に周知されている機動部一課と同様、特異災害バグに対する、被害拡大の阻止と事態収拾を担っているのだが、決定的に異なる点がひとつあった。
『型システム』――
天敵バグの駆逐のため、人類が備えうる、唯一絶対の兵器体系。その開発と、その行使である。
型システムは装着者ごとに非常に異なった姿を見せ、強さも様々だが、型システムを身に纏ったものは、対応するバグに対して効率的・有効な攻撃手段を備え、一網打尽にすることが可能となる。
だが、機動部二課が保有する異端技術の結晶でもある強く静的な型推論システムは、バグを殲滅せしめる強力な武装であると同時に、広く普及している既存言語の型システムとは比較にならぬ負担を装着者の脳に強いるもので、激しい拒絶反応を誘発することも多かった。このため、不用意な接触による国民への被害、米国との安全保障条約や、周辺諸外国に対する影響も鑑みられ、日本の型兵器の保有実態は、現在の政府与党判断によって、完全に秘匿されている状態でもある。
誰に知られることなくバグと戦い、ヒトの暮らしを守るニンジャたちが年端もいかぬ少女たちであることを――その正体が、当代トップの競技プログラミングユニットである『ツヴァイウナギ』の２人、[redacted]と[redacted]であることを知る者は、ごく僅かに限定されている。
[redacted]と[redacted]は、この春より、私立バイナリアン程序院高等科に通うこととなった。
憧憬の対象である、アーティスト・[redacted]が通うことでも知られるバイナリアン程序院に通うことは、[redacted]と[redacted]にとって、望外の喜びである。
しかし――そのことが大きな運命の転換になろうとは、まだ気付いてはいない。

ハスケるのを諦めるなッ！
オキャムるのを諦めないで!!

新たなるニンジャソウルの覚醒は、すぐそこにまで迫りつつあった。

第12章

蓮物語

— @dif_engine

物語の概要と目的

集合と写像について一応のことを知ってはいるけれど関数型言語に触れたことがない主人公がHaskellを学びます。同時に主人公は圏論の初歩を学び、圏論を応用するとHaskellの幾つかの重要な型クラスの挙動が理解しやすくなることを知ります。物語を通し、主人公は最小限の圏論の知識を手がかりにHaskellの型を理解しようと試行錯誤することになります。

このような話題をきちんと論じるならば、集合と写像のなす圏を圏論の立場で丁寧に抽象化してから論じることになるでしょう。とはいうものの、圏論をゼロから学びcartesian closed categoryのあたりまで進めるのはなかなか大変です。そこまで徹底した抽象化を前提としなくてもHaskellと圏論の関わりを説明することはある程度可能だと筆者は考えました。物語を通して、圏論についての初歩的な知識を紹介し、Haskellの幾つかの重要な型クラスが従うルールを圏論の立場から説明します。

12.1 プロローグ

怪異——。

そう呼ぶのが如何にもふさわしいように思えた。髪の毛の代わりに頭から青い触手が生えている少女を怪異と呼ばないのなら怪異とは一体何のための言葉だというのだろう。

晴れた夏の午後だった。他の観光客たちが思い思いにビーチライフを満喫する中、海の家で少し遅目の昼食を食べた僕は場違いにもテーブルに本とノートを広げて思案していた。

「——それはHaskellじゃなイカ？」

食べ終えたカレーライスの皿をテーブルから下げるついでに僕のノートを覗きこんだ少女はそんな風に声をかけてきた。少女はイカ娘と名乗った。

「いか——むすめ？」

「そうでゲソ」

風変わりな姿に似つかわしく、やはり名前も風変わりだと思った。他人の名前についてあれこれ言うのは——しかも初対面の相手に対してならなおさらだ——いささか不作法ではあったが僕は素直にそんな感想を述べた。もしかしたら思いがけず怪異に遭遇したことで僕はいくらか動揺していたのかもしれない。

「阿良々木暦（あららぎこよみ）という名前も十分風変わりじゃなイカ？」

これは僕の名前に対する彼女の感想。

とにかく、僕達の会話はそんな風に始まった。

ありふれた出会い。

そしてありふれた会話。

しかし今にして思えばあの瞬間から——謎めいた暗殺者との死闘とその奇妙な結末へ至る運命のレールが敷かれてしまったのではないか——そんな風にも思うのだ。あの死闘の結末を思い出さずにはいられない。あの暗殺者は——。

——本当に消滅してしまったのだろうか。
枯れて泥水の下に沈んだ蓮の花が残した種が誰にも気づかれる事なく泥中で発芽し、ある日思いがけず白い花として水面上に現れて驚かすように、あの暗殺者が突然この世界に蘇る——そんな事がいつか起こるのではないかと思えてならない。

この物語は——泥のように平凡な未来の日常の中、復讐のために忽然と——蓮の花のように——現れるかもしれない暗殺者の物語だ。また、これは僕が Haskell という言語に出会った物語でもある。

12.2　ラーン・ユー・ア・ハスケル・フォー・グレート・グッド

12.2.1　図書館にて

僕が Haskell に最初に出会ったのは、その数日前の土曜日のことだった。

高校三年生の五月に入ってから僕は遅まきながら大学受験に向けて本格的な勉強を始めていた。
事情があって予備校に行かせてもらえなかった僕は夏休みのあいだ同級生に勉強の面倒を見てもらうことにしていたのだった。面倒を見てくれるのは学年トップクラスの成績を有する戦場ヶ原ひたぎと学年トップの成績を有する羽川翼だ。

その日は羽川の担当だった。帰る間際になって彼女から本を手渡された。ポップな青い象の表紙の本と、黄色い洋書——こちらは少し古い本らしい——だった。

「あのさ、阿良々木くんは数学って得意だよね？　これはね、Haskell って言うプログラミング言語の本なんだけど——」
そう言って羽川は象の表紙の本を僕に差し出した。
象の本の表紙には『すごい Haskell たのしく学ぼう！』と書いてあった。

「……よくわからないけどこれを読めということか？」

「——あのね阿良々木くん、大学進学を真面目に考えるなら大学に入るまでだけの勉強だけじゃなくて、卒業したら何をするかも考えないとね——だから世の中に出て武器になりそうな技能を身に着けることも今から意識した方がいいと思うんだ。それで考えてみたんだけど阿良々木くんは数学を核にできるような技能を身につけるといいと思う」
羽川は非の打ち所がない人間だ。恩人でもある。そして彼女はとても面倒見が良い。ついでに言うと思い込みも激しい。

「……つまりこの Haskell という言語は数学が得意なら簡単に身につけられて、しかも世の中に出て武器になるということか？」

「私はそんな事言ってないよ」

「……でしたっけ……？」

「私はね、『武器になりそうな』って言っただけ。実際に武器にできるかどうかは阿良々木くん次第だよ——それに、『数学が得意なら簡単』とも言ってないよ——私は、阿良々木くんが得意の数学を生かして武器にできそうなものを探して提案しただけだよ」
文字にすると少しキツい事を言っている印象になるが、羽川はこんな時でも——なんというか穏やかな雰囲気なんだよな。

「つまり、武器にできるかどうかは僕の努力次第——という理解で OK?」

「うん。今度の月曜日はどうしても外せない用事があって『一回休み』だから日曜日と月曜日でその象の本を読んでおいてね。」

「羽川先生、なんかすごい無理ゲーな気がするんですが！　イージーモードでお願いします！」
そんな僕の抗議をあっさりスルーして羽川は——。

「それからね、その黄色い本は副読本のつもり。Haskell の本で分かりにくいところがあったらその本を読んでみるといいと思って」
黄色い本には『Categories for the Working Mathematician』と書いてあった。
洋書だ。
英語だ。

「……は、羽川先生、これ英語で書いてあるんですが！」

「数学の英語だから難しくないよ。それに大学に行くなら英語の専門書にも慣れておいたほうがいいと思って」

「……」

斜め上すぎる天才的正論に言挙げする気力を失った僕に追い打ちをかけるように羽川は——。

「んん、——あとね、水曜日に簡単な理解度のテストをするからね」

と宣告した。こうして僕は、本二冊分増し加わっただけのはずの荷物をひどく重く感じながら家路についたのだった。

12.2.2 羽川の本

夕食の後、羽川が貸してくれた本を読み始めた。そのときになって気付いたのだが両方とも図書館から借りてきたのではなく、羽川の私物だった。——図書館で貸してくれたからつい勘違いしていたのだが、よく考えれば羽川が本のまた貸しのようなマナー違反をするわけがないのだった。

洋書の方は古書店で買い求めたものらしく、見返しに鉛筆で数字が走り書きしてあった。いずれにせよ、羽川の複雑な家庭事情から推察する小遣い事情から考えて——安くない買い物だったはずだ。

身が引き締まる思いだった。情けない話だが、本当はこのときまで僕はあれこれ理由をつけて読まずに本を返却するというオプションを頭に入れていたのだった。だが——恩人である羽川がここまでしてくれたのだ。やれるところまで頑張ろう——柄にもなくそんな事を考えながら本を読み進めたのだった。

12.2.3 Haskell

もちろん羽川の心配はありがたかったが、僕もまったく将来の事を考えていなかったわけじゃない。自分なりに数学が得意であることを生かした技能について考え、IT 関係の技能を何か習得しようと考え、C 言語はそれなりに使えるようになっていた——初歩的なレベルではあったが。

一方 Haskell という言語については——まったくの初聞だった。しかし青い象の本を読み始めて少し経つと、数学を得意とするこの僕になぜ羽川がこの言語を薦めてくれたのか、その理由が徐々にわかってきた。この言語を使うと、数学的なアイデアに近い形で計算ができてしまうようだ。例えば、リストを扱ったこんな計算例を見てみよう——。

```
ghci> [x*x+1 | x <- [1..20]]
[2,5,10,17,26,37,50,65,82,101,122,145,170,197,226,257,290,325,362,401]
```

これは見かけも内容も

$$\{x^2+1 \,|\, x \in \{1,\ldots,20\}\}$$

という集合を作る様子と似ている。もし素数かどうかを判定する関数

$$\text{isPrime}\colon \mathbb{Z} \to \{\text{true}, \text{false}\}$$

が与えられていたとしたら、`filter` という関数と組み合わせて、先程のリストからこんな風にして素数だけを濾し取る (filter) こともできる[*1]——。

```
ghci> filter isPrime [x*x+1 | x <- [1..20]]
[2,5,17,37,101,197,257,401]
```

なるほど、こうやって関数を組み合わせて計算してゆくのが Haskell という言語のスタイルなのだろうか……？

12.2.4 数学：関数の復習

数学：関数の定義の復習

関数の話が出たついでに、「関数」という概念が集合論に基づいた数学ではどんな形で定義されていたか、一応復習しておこう。何を今更という感じがしないでもないが、記号や言葉遣いを確認したいという思惑も

[*1] `Data.Numbers.Primes` というパッケージを使えば実際にこのコードを実行できる。

あるので、少しお付き合い頂きたい。

集合論的な意味での関数 $\varphi\colon A \to B$ とは、積集合 $A \times B$ の部分集合であって次の性質 **(function)** を満たすものを指すのだった――。

$$\textbf{(function)} \quad \forall x \in A \ \exists! y \in B \ \langle x, y \rangle \in \varphi.$$

もちろん、上の条件に現れる「一意的存在」を表す記号 $\exists!$ を解いて

$$\textbf{(function')} \quad \forall x \in A \ \exists y \in B \ \langle x, y \rangle \in \varphi \ \wedge \left(\forall y' \in B \ \langle x, y' \rangle \in \varphi \Rightarrow y = y' \right).$$

と書きなおしてもよい。$\varphi \subseteq A \times B$ が関数条件 **(function)**（あるいは同じ事だが **(function')**）を満たすとき、$^{\forall} x \in A$ に対して、$\langle x, y \rangle \in \varphi$ となる $y \in B$ がただ一つ存在するが、このとき **φ の x における値は y であると**言い、

$$y = \varphi(x)$$

と書く。引数 x を括弧で囲まずに

$$y = \varphi x$$

と書く場合があるのもご存知の通り――要するに意味が通じれば良いわけだ。一般に、関数 $\varphi\colon A \to B$ が与えられたとき、A を φ の**定義域**、B を φ の**値域**と呼ぶ。

以上の定義は数学における標準的なものだが、分野によっては関数についてもう少し違った立場の定義を与える場合もある。すなわち、「関数 $\varphi\colon A \to B$」であっても、任意の $x \in A$ に対して、$\varphi(x)$ **が定義されているとは限らない**ようなものを許容するように関数という概念を定義する立場である。その立場で議論する場合には、$\varphi(x)$ が定義されていないような $x \in A$ が存在するとき φ を**部分関数**と呼び、任意の $x \in A$ に対して $\varphi(x)$ が定義されているとき F を**全域関数**と呼ぶ。以下では数学における標準的な関数の定義を採用する。したがって、特に断らない限り本稿における関数はすべて――後者の立場で謂うところの――全域関数だということになる。

数学：関数の合成

関数あるいは写像の合成というのは自然なアイデアだ。関数

$$f\colon X \to Y$$

と関数

$$g\colon Y \to Z$$

があったとき、$^{\forall} x \in X$ を関数 f で写して得られる $f(x)$ は――f の値域が Y であることがわかっていることから――集合 Y の元となり、従ってこれに対して関数 g を適用することができる。こんな風にして続けて関数を適用すると

$$g(f(x))$$

という元が得られるが、このようにして $^{\forall} x \in X$ に対して $g(f(x))$ を対応させる関数を**関数 f と g の合成関数**と呼び、

$$g \circ f$$

という記号で表すのだった。合成関数 $g \circ f$ の定義域と値域はどうなるだろうか。――これは簡単だ。今の簡単な考察から明らかに

$$g \circ f\colon X \to Z$$

となる。余談だが、こういう話をするとき、

$$\mathrm{dom}(g \circ f) = X, \ \mathrm{cod}(g \circ f) = Z$$

などと書くようにすると後々便利だ。つまり、$\varphi\colon A \to B$ のとき

$$\begin{aligned}\mathrm{dom}(\varphi) &:= A,\\ \mathrm{cod}(\varphi) &:= B\end{aligned}$$

としておこう。dom は英語の domain から来ている。cod は、ええと——なんだったっけ[*2] ……。

さて、集合 X, Y に対して

$$\mathrm{Map}(X,Y) := \left\{ \varphi \subseteq X \times Y \;\middle|\; \varphi \text{ は関数} \right\}$$

と定義しよう。つまり、$\mathrm{Map}(X,Y)$ は関数を要素とする集合である。
こうすると、関数合成の記号 $\circ$ を $\mathrm{Map}(Y,Z)\times\mathrm{Map}(X,Y)$ 上の演算とみなすこともできる——。

$$\begin{array}{ccc}\mathrm{Map}(Y,Z)\times\mathrm{Map}(X,Y) & \xrightarrow{\circ} & \mathrm{Map}(X,Z)\\ \langle g, f\rangle & \longmapsto & g\circ f\end{array}$$

関数合成の演算 $\circ$ が結合法則 (associative law)

$$h\circ(g\circ f) = (h\circ g)\circ f$$

を満たすことは簡単に確かめられる。実際、任意の $x \in \mathrm{dom}(f)$ に対して、関数合成の定義を繰り返し適用すれば

$$\begin{aligned}\big(h\circ(g\circ f)\big)(x) &= h\big(g\circ f(x)\big)\\ &= h\big(g\,(f(x))\big)\\ &= h\circ g\big(f(x)\big)\\ &= \big((h\circ g)\circ f\big)(x)\end{aligned}$$

となる。——当たり前の話だが、定義域と値域がうまく合わなければ関数の合成はできない。関数の合成を考えるときにはいつも定義域と値域を意識して扱う必要がある。

数学：関数記号の位置、特に二項演算子

先程、関数 φ の x における値を

$$\varphi(x) \quad \text{または} \quad \varphi x$$

と書くと説明したが、実際にはどんな関数でも必ずそう書くわけではない。例えば、二つの実数 x_1, x_2 の和 $x_1 + x_2$ を返す関数

$$\begin{array}{ccc}+\colon \mathbb{R}\times\mathbb{R} & \to & \mathbb{R}\\ \langle x_1, x_2\rangle & \longmapsto & x_1 + x_2\end{array}$$

の場合には関数の記号「+」は二つの引数の間に置かれている。こんな風に、二つの引数の間に関数記号を置くタイプの関数を、**中置関数**とか**二項演算子**などと呼ぶのだった。中置関数との比較の意味では、φx のように関数記号を先に、その後に引数を置くタイプの関数は**前置関数**と呼べるだろう。論理的な立場を徹底するならば、関数記号は常に先に書くことに統一したほうがスッキリするのかもしれないが、その立場では結合法則も

$$+\big(+(x_1, x_2), x_3\big) = +\big(x_1, +(x_2, x_3)\big)$$

と書かねばならないし、この書き方は

$$(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$$

と比べて随分と分かりにくいと思うのだ。

ついでの思いつきだが、前置関数と中置関数があるのだから**後置関数**というものがあってもおかしくないな——。

$$x\,\varphi$$

[*2] codomain 。普通の英語の辞書には載ってない。

——理屈としてはあってもおかしくないが実例はなさそうだ——いや、そうでもないか。階乗関数

$$n \longmapsto n!$$

の関数記号は『!』だから、これは引数を 1 つ持つ後置関数だ。

関数の種類	引数の数	形式
前置関数	≥ 1	$\varphi(x_1, \cdots)$
二項演算子 (中置関数)	$= 2$	$x \ \varphi \ y$
後置関数	≥ 1	$(\cdots, x_1)\varphi$

前置関数や中置関数や後置関数は、関数の本質についての分類ではなく、単に関数記号をどこに置くかという面から見た分類だ。

数学：二項演算子の結合性

これもまた本質というよりは書き方の習慣の話なのだが、二項演算子 (中置関数) には**結合性**がある。二項演算子 $\varphi\colon X \times X \to X$ があったとき、

$$A \ \varphi \ B \ \varphi \ C$$

と書いたとき、真ん中の B：

$$A \ \varphi \ \boxed{B} \ \varphi \ C$$

が「右にくっつく」つまり

$$A \ \varphi \ \left(\boxed{B \ \varphi \ C} \right)$$

と解釈するという約束になっている場合、φ を**右結合演算子**と呼ぶ。同様に、真ん中の B が「左にくっつく」つまり

$$\left(\boxed{A \ \varphi \ B} \right) \ \varphi \ C$$

と解釈するという約束になっている場合、φ を**左結合演算子**と呼ぶ。「結合性は左である」とか「結合性は右である」などという言葉遣いをする場合もある。

12.2.5 数学 ∩ プログラミング：型とその階層

プログラミング言語で扱うデータには色々な種類がある——。それは整数だったり浮動小数点数だったり文字列だったりするわけだが、普通の感覚に照らして考えれば数と文字列のデータは異なる**型 (type)** のデータだ。つまり、文字列データ `s` は文字列型 `String` と呼ばれるデータの集合に属し、整数データ `z` は整数型と呼ばれるデータの集合 `Int` に属し、更に

$$\texttt{String} \cap \texttt{Int} = \varnothing$$

となっている事が了解されているわけだ。型は集合の一種だが、集合ならどんなものでも良いのではなく——`String` や `Int` のように——同質な要素からなるような、特定の集合だけを型として扱う。

何らかの基礎的な集合があるとき、段階的に複雑な集合を作ってゆくというアイデアは自然だ。——例えば、実数列

$$(r_n)_{n \in \mathbb{N}}$$

を考えると、要するに自然数 n に対して実数 r_n が一つだけ決まれば良いのだからこれを関数

$$r\colon \mathbb{N} \to \mathbb{R}$$

と同一視できることがわかる。だから、実数列全体の集合は関数の集合

$$\mathrm{Map}(\mathbb{N},\mathbb{R})$$

と同一視できる。こうして——少し大げさな物言いかもしれないが——集合 $\mathbb{N}$ と $\mathbb{R}$ というビルディングブロックを Map というボンドで貼りあわせることによって、より複雑な集合——$\mathrm{Map}(\mathbb{N},\mathbb{R})$——を作った事になる。

全く同じようにして、**基本型 (basic types)** と呼ばれる基本的な型——データの集合——たち $S_1,...,S_N$ が与えられたとき、これらを Map で貼り合わせることによって、より複雑な**型**を考えることができる。——こうして得られた**型**は、Map で貼り合わせるための新しいビルディングブロックとみなせる——こうして更に複雑な**型**を考えることができる。このような操作を続けた結果得られる**型**全体の集まり $\mathscr{T}$ は、次のようにして——帰納的に——構成することができる。

$$\begin{aligned}
&\mathscr{T}_0 := \{S_1,...,S_N\}.\\
&\mathscr{T}_{n+1} := \mathscr{T}_n \cup \left\{ \mathrm{Map}(T_1,T_2) \,\middle|\, T_1,T_2 \in \mathscr{T}_n \right\} \quad (n \geq 0).\\
&\mathscr{T} := \bigcup_{n\in\mathbb{N}} \mathscr{T}_n.
\end{aligned}$$

結果として得られる型全体の集まり $\mathscr{T}$ は次の性質 **(maptypes1)(maptypes2)** を満たす事になる——。

(maptypes1) $S_1,...,S_N \in \mathscr{T}$,

(maptypes2) $^\forall T_1,T_2 \in \mathscr{T} \quad \mathrm{Map}(T_1,T_2) \in \mathscr{T}$.

与えられた型から別の型を作る方法は Map だけとは限らない。プログラミング言語において、型 X のデータを並べたもの——ここでは $\mathrm{List}(X)$ と呼ぼう——を考えることが多い。集合論的に模倣するならばこんな感じだろうか——。

$$\mathrm{List}(X) := \bigcup_{n\in\mathbb{N}} X^n,$$

ただし $X^0 := \{\varnothing\}, X^1 := X$ とする。この場合、List は一つの型 X に対して別の型 $\mathrm{List}(X)$ を返してくれる対応だと考えられる。いま見てきた Map や List のように、いくつかの「型パラメータ」から新しい型を作る写像を**型コンストラクタ (type constructor)** と[*3]呼ぶ。

Map は 2 つの型パラメータを取る型コンストラクタであり、List は 1 つ型パラメータを取る型コンストラクタであった。一般に、任意の $n>0$ に対して、n 個の型パラメータを持つ型コンストラクタが考えられる。このような型コンストラクタが存在する場合に**型**全体の集まり $\mathscr{T}$ がどうなるか考えてみよう。

$M(>0)$ 個の型コンストラクタからなる集合

$$K = \{k_1,\ldots,k_M\}$$

に対して、それぞれの型コンストラクタが持つ型パラメータの個数——この個数のことを英語では arity と呼ぶ——を対応させる関数

$$\mathrm{ar}\colon K \to \{1,2,3,\ldots\}$$

が与えられているものとする。つまり任意の型コンストラクタ $k \in K$ に対して $\mathrm{ar}(k)$ は k の型パラメータの個数なのだから、型コンストラクタ k は

$$k\colon \overbrace{\mathscr{T}\times\cdots\times\mathscr{T}}^{\mathrm{ar}(k)\text{ 個}} \longrightarrow \mathscr{T}$$

という写像になっている。基本型 $S_1,...,S_N$ が与えられたとき、型コンストラクタ $k_1,\ldots,k_M$ に基づいた型全体の集まり $\mathscr{T}$ の帰納的な構成は、次のようになる——。

[*3] **型構成子**という訳語を充てる場合もある。

$$\mathscr{T}_0 := \{S_1, ..., S_N\}.$$
$$\mathscr{T}_{n+1} := \mathscr{T}_n \cup \bigcup_{i=1}^{M} \left\{ k_i(T_1, ..., T_{\mathrm{ar}(k_i)}) \;\middle|\; T_1, ..., T_{\mathrm{ar}(k_i)} \in \mathscr{T}_n \right\} \quad (n \geq 0).$$
$$\mathscr{T} := \bigcup_{n \in \mathbb{N}} \mathscr{T}_n.$$

この場合、結果として得られる型全体の集まり $\mathscr{T}$ は次の性質 **(gentypes1)(gentypes2)** を満たす事になる——。

(gentypes1) $S_1, ..., S_N \in \mathscr{T}$,
(gentypes2) $\forall k \in K \ \forall T_1, \ldots, T_{\mathrm{ar}(k)} \in \mathscr{T} \quad k(T_1, \ldots, T_{\mathrm{ar}(k)}) \in \mathscr{T}$.

12.2.6 Haskell と型

さっきから型についての取り留めのない話を書いていて、気がついたら 2 ページぐらい使ってしまった気がするが——これは青い象の本を途中まで読んだ時点での感想を述べるための前フリというやつである。

Haskell：関数の型

例えば f が集合 X から Y への関数であるとき、数学では——先ほど導入した型コンストラクタの記号 $\mathrm{Map}(X,Y)$ を用いれば

$$f \in \mathrm{Map}(X,Y)$$

と書ける。Haskell で同等の関数宣言を書くならば次のようになる——。

```
f :: (X -> Y)
```

一般的な書式は

```
変数 :: 変数が属する型
```

——となる。

もし C 言語でこれに相当する関数を書くならばそのプロトタイプ宣言は

```
Y  f(X);
```

となるはずだ——これぐらいなら `f` の型——つまり何を受け取って何を返す関数なのか——を読み取るのは簡単だ。

だが、「関数を受け取って値を返す関数」——例えば

$$g\colon \mathrm{Map}(X,Y) \to Z$$

を考えたりするとき、これに相当する C 言語の関数プロトタイプ宣言[*4] はこんなふうになる。

```
Z  g(Y(*)(X));
```

——もちろん `typedef` を使えば今のプロトタイプ宣言と等価なものをもう少し読みやすく書けるかも知れない——こんな風に——。

```
typedef Y (*MapXY)(X);
Z  g(MapXY);
```

[*4] 関数ポインタを使っている。説明は省略する。サツバツ！

もちろん「わかりやすさ」というのは人それぞれだ。しかし、どちらの書き方をされたとしても僕はこの g が Map(Map(X,Y),Z) に属する関数を表している事を簡単には読み取れない。一方、g: Map(X,Y) → Z に相当する Haskell の関数宣言はこうなる。

```
g :: (X -> Y) -> Z
```

さらに複雑なこんな例を考えると――。

```
g :: ((X -> Y) -> Z) -> W    -- Haskell
W  g(Z (*)(Y(*)(X)));  /* C */
```

すげえ。
高階の関数――関数を返す関数――を簡単に書けるんだな。

たった今出てきた `(X -> Y)` は、数学的には Map(X,Y) に相当し、12.2.5 の用語で言えば 2 つの型パラメータを持つ**型コンストラクタ**である。ただしこの型コンストラクタの記号`->`は**右結合の二項演算子**のように振る舞う。つまり、

```
X -> Y -> Z
```

――と書いたとき、真ん中の `Y` は「右にくっつく」ので――

```
X -> (Y -> Z)
```

――と書いたのと同じ意味になる。

Haskell：複数の引数

複数の引数を扱うやりかたは Haskell と C 言語では少し――いや――かなり違う。特別な理由がない限り、n 引数関数に対応する Haskell の関数宣言はこんな風に書くのが良さそうだ――。

```
f :: X1 -> X2 -> X3-> ... -> Xn  -> Y
```

これは

$$f \in \mathrm{Map}(X_1 \times X_2 \times X_3 \times \cdots \times X_n, Y)$$

とは全然違う型の宣言になっている。実際、`->`の結合性を利用せずに書きなおせば

```
f :: X1 -> (X2 -> (X3 -> (... -> (Xn -> Y) ...)))
```

となり、これに該当するものを Map を使って書くならば

$$f \in \mathrm{Map}(X_1, \mathrm{Map}(X_2, \mathrm{Map}(X_3, \ldots \mathrm{Map}(X_n, Y)\ldots)))$$

となる。――記号が込み入ってきたみたいだ。Map という記号で書くと入れ子が複雑になるので、必要に応じて

$$\mathrm{Map}(X,Y)$$

という書き方の代わりに

$$X \triangleright Y$$

と書くことにしてみよう。Haskell の真似をして ($X \to Y$) にしなかったのは

$$X \to Y \to Z$$

と書いたときデータ型 X,Y,Z が

$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

のように関数で結ばれているという状況を――数学での習慣通りに――意味して欲しいからだ。
念のために書けば

$$X \triangleright Y = \{\varphi : X \to Y\}$$

——となる。
演算子 ▷ の結合性は Haskell に合わせて右にしておく。つまり、

$$X \triangleright Y \triangleright Z$$

と書いたらそれは

$$X \triangleright (Y \triangleright Z) \quad \left(= \ \mathrm{Map}\big(X, \mathrm{Map}(Y,Z)\big) \right)$$

を意味するということだ。少し見慣れない記号を入れたのでついでに練習しておくと、

$$x \in X,\ \ f \in X \triangleright Y \ \ \Rightarrow \ \ f(x) \in Y$$

となる。
さて、新しい記法のウォーミングアップも終わったので

```
g :: A -> B -> C -> D
```

という型を持った Haskell の関数宣言を考えることにする。g を数学の関数 g と同一視して書いてみれば

$$g \in A \triangleright B \triangleright C \triangleright D = A \triangleright (B \triangleright C \triangleright D) = \mathrm{Map}(A, B \triangleright C \triangleright D)$$

——となる。すると、$^\forall a \in A$ に対して

$$g(a) \in B \triangleright C \triangleright D = \mathrm{Map}(B, C \triangleright D)$$

となる。つまり $g(a)$ は関数だ。だから、$^\forall b \in B$ を持ってくれば

$$g(a)(b) \in C \triangleright D = \mathrm{Map}(C, D)$$

となる。全く同じようにして $^\forall c \in C$ を持ってくれば

$$g(a)(b)(c) \in D$$

となる。Haskell で複数の引数を持つ関数を書くとき、こんな風にして「関数を返す一引数関数」として実現することをカリー化[*5] というらしい。

Haskell：多相型

型についての話を書いたとき (12.2.5)、**型コンストラクタ**についても触れたのだった。Haskell では、例えばこんな風に型コンストラクタを宣言できる——。

```
data Maybe a = Nothing | Just a
```

今の例では、Maybe は一つの型パラメータを持つ型コンストラクタになっている。一つ以上の型パラメータを持つ型を**多相型**と呼ぶ。先程も触れたことだが、->は二つの型パラメータを持つ型コンストラクタだ。型コンストラクタは**型を受け取って型を返す**——例えば Maybe に型パラメータ Int を与えると Maybe Int という具体型を返す——写像なのであって、関数より上位の概念だが、やはり関数に似た型を考える事ができる。このような型コンストラクタの『型』に相当するものを型 (type) と区別して kind [*6] と呼ぶ。これは GHCi 上では:kind コマンドで確認できる——。

```
ghci> :kind Maybe
Maybe :: * -> *
```

```
ghci> :kind (->)
(->) :: * -> * -> *
```

[*5] 大雑把で不正確な説明かも知れないが、以下の議論のためにはこの程度の理解で十分である。
[*6] 『すごい Haskell 楽しく学ぼう！』では「種類」という訳語があてられている。

Haskell：型クラス

Haskell の型クラスは、型が実装するべき関数の宣言を記述する。自分で定義した型 MyType の要素に対して等号 (==) を使った等値性判定を行いたいならば、MyType を Eq 型クラスのインスタンスとして定義しておけばよい——。

```
instance Eq MyType where
   ......
```

Eq 型クラスの宣言はこんな形をしている——。

```
class  Eq a  where
   (==):: a -> a -> Bool
   (/=):: a -> a -> Bool
```

上に出てくる a は具体的な型が収まるべきプレースホルダーになっている。

Haskell：ファンクター

青い象の本を 7 章まで読み進めると、Functor 型クラスというのが出てきた。それはこんな風に宣言されている——。

```
class  Functor f where
   fmap :: (a -> b) -> f a -> f b
```

f a とか f b という塊が見えるが、これは型でないと意味をなさない。とすると f は型コンストラクタが収まるべきプレースホルダーなのだろうか？ ——どうやらそうらしい。実際にこんな例が掲げられていた：

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f  Nothing = Nothing
```

型クラス Eq の場合、そのインスタンスは具体的な型だが、型クラス Functor の場合、そのインスタンスは多相型でなければならない——ということになる。

更に読み進めようとしたとき、本にメモ——羽川の字だ——が挟んであるのに気付いた。内容はこうだった——。

[羽川のメモ]
functor : 関手、category theory、functor の object-part と arrow-part

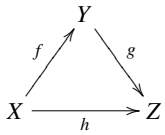
羽川が貸してくれた黄色い本『Categories for the Working Mathematician』と関係していそうだ。——そう言えば、Haskell の本で分かりにくいところがあったらその本を読めと言われたのだった。object-part とか arrow-part はなんだろう？ 黄色い本を読んだらわかるのだろうか——。しかし夜も随分更けてしまった。現役の吸血鬼ではない僕にとって、夜は睡眠のための時間だ。

12.2.7 圏論（1）

日曜日は『Categories for the Working Mathematician』を読んだ。Category theory ——日本語では**圏論（けんろん）**と呼ばれる数学の一分野の本だった。英語が得意でない自分が英語で書かれた数学の本を読むなど無理だと思い込んでいたが、使われている文法はむしろ高校の教科書よりもレベルが低いことに気付いた。確かに単語は見慣れないものが多かったが、ネットのお陰で大体の単語は調べがついた。

数学：圏論の意味

今まで紹介する機会がなかったが僕には少し変わった習慣がある。それは、本の序文をちゃんと読むということである。——こう書くと別に普通の事のようだが、序文や目次に丁寧に目を通す人は実は少数派なのだ。そんなわけでいつもどおり僕は黄色い本の序文 (Introduction) を読み始めた。序文[*7] の内容は実際には難しく、最初の段落以外はよくわからなかった。それでも理解できたと思える範囲を序文を参考に説明するとこういうことになる——。

> 圏論は、数学的な諸理論の多くの性質が矢印の図によって統合でき、本質の理解を容易にするという観察から誕生した。例えば集合 X,Y,Z とそれらの間の関数からなる図
>
> 
>
>
> を考えよう。——ここでは、意図的に集合の元 x とその関数値 $f(x)$ の関係を図から省いている。このとき、X から Z には h という関数で到達することができるが、「遠回り」して Y を経由して到達することもできる。——この「遠回り」の場合実際には f と g を使っているから結果として合成関数 $g\circ f$ を使って Z に到達したことになる。そんなわけだから、もし $h=g\circ f$ であるときには、X から Z に到達するためには、どの道を通っても同じという事になる。こんなふうに「どの道を通っても同じものが得られる」とき、この図式 (diagram) は**可換 (かかん、commutative)** であると言う。今の例は集合の数がたった三つだったが、もっと集合の数やそれらを結ぶ関数の数が増えてくると、こういった図式でもっと複雑な状況を絵に書いて把握することが出来るというのが利点だ。
>
> ところで、こんな図にまで抽象化してしまうと、この図式を異なる文脈で正しく解釈できることに気づく。例えば、集合 X,Y,Z が位相空間[*8]であり、f,g,h がそれらを結ぶ連続写像であると思っても今の議論は成立する。

序文で理解できたのはこの辺までで、正直に言うが僕は圏論の何が偉大なのか理解出来なかった。僕は序文をきちんと読む少数派の人間だと大見得を切った直後のこの体たらく、面目次第もない。
少し面白いと思ったのは、X が集合であることを意識しないように $x\in X$ や $x\mapsto f(x)$ という元と元の関係を意図的に図式から省いてあったことだ。もしかしたら圏論というのは写像で結ばれた集合の系を代数的に抽象化して捉える理論なのかもしれない。

数学：圏の定義

難しくなってきた序文を放り出して読み進めてみるとどうやら僕の予想は大体当たっていたようだった。本に従って定義[*9]を書いてみることにする。

圏の定義

集合族 O,A と写像

$$\begin{array}{lll} A \xrightarrow{\mathrm{dom}} O, & A \xrightarrow{\mathrm{cod}} O, & O \xrightarrow{\mathrm{id}} A \\ f \longmapsto \mathrm{dom}(f), & f \longmapsto \mathrm{cod}(f), & X \longmapsto \mathrm{id}_X \end{array}$$

が与えられているとする。更に、

$$A \times_O A := \left\{ \langle g,f\rangle \in A\times A \,\middle|\, \mathrm{dom}(g)=\mathrm{cod}(f) \right\}$$

[*7] Categories for the Working Mathematician(以下では CWM と略記）, p.1
[*8] 「連続性」にまつわる事柄を研究するための基礎として作られた空間概念。意外にもプログラミング言語の基礎づけにも役立つ事が後に知られるようになった。
[*9] CWM, p.10, ただし説明の都合や好みなどから記号は一部変えてある。

上定義された二項演算

$$\begin{array}{ccc} A \times_{\mathrm{O}} A & \xrightarrow{\circ} & A \\ \langle g, f \rangle & \longmapsto & g \circ f \end{array}$$

が与えられたとする。このとき、

$\mathbf{C} := \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ \rangle$ が**圏** (**category**) であるとは (cat1) – (cat5) が成立することである：

(cat1)	$^{\forall} X \in O$	$\mathrm{cod}(\mathrm{id}_X) = \mathrm{dom}(\mathrm{id}_X) = X.$
(cat2)	$^{\forall} \langle g, f \rangle \in A \times_{\mathrm{O}} A$	$\mathrm{dom}(g \circ f) = \mathrm{dom}(f).$
(cat3)	$^{\forall} \langle g, f \rangle \in A \times_{\mathrm{O}} A$	$\mathrm{cod}(g \circ f) = \mathrm{cod}(g).$
(cat4)	$^{\forall} f, g, h \in A$	$\langle h, g \rangle, \langle g, f \rangle \in A \times_{\mathrm{O}} A \implies (h \circ g) \circ f = h \circ (g \circ f).$
(cat5)	$^{\forall} f \in A$	$f \circ \mathrm{id}_X = \mathrm{id}_Y \circ f = f,$ ただし $X := \mathrm{dom}(f), Y := \mathrm{cod}(f).$

圏 $\mathbf{C}$ が与えられたとき、

$$\begin{aligned} \mathrm{Ob}(\mathbf{C}) &:= O, \\ \mathrm{Arr}(\mathbf{C}) &:= A \end{aligned}$$

とそれぞれ呼び、$\mathrm{Ob}(\mathbf{C})$ の要素を**対象 (object)** と呼び、$\mathrm{Arr}(\mathbf{C})$ の要素を**射 (arrow)** と呼ぶ。

何やら随分抽象的だが、具体例——集合と関数からなる系——を念頭に置きながら読んでみることにしよう。例えば、集合 $S_1, \ldots, S_N$ に対して

$$\begin{aligned} O &:= \{S_1, \ldots, S_N\}, \\ A &:= \bigcup_{1 \leq i, j \leq N} \mathrm{Map}(S_i, S_j) \end{aligned}$$

としてやり、$^{\forall} X \in O$ に対して

$$\begin{array}{ccc} \mathrm{id}_X : X & \longrightarrow & X \\ x & \longmapsto & x \end{array}$$

として定義してやれば、$\mathbf{C} := \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ \rangle$ は確かに圏になる事がわかる——。

集合と関数を扱っていたときには

$$\mathrm{Map}(X, Y)$$

というレベルで見ると関数合成などの理解がしやすくなったのだった。圏論でそれに該当するものは Hom と書かれることになっている——。

$$\mathrm{Hom}(X, Y) := \left\{ f \in \mathrm{Arr}(\mathbf{C}) \,\middle|\, \mathrm{dom}(f) = X, \mathrm{cod}(f) = Y \right\}.$$

複数の圏を扱っているときは**どの圏で考えているのか**を明記するために添字を付けて

$$\mathrm{Hom}_{\mathbf{C}}(X, Y)$$

と書いたりもする。

圏の定義から、任意の対象 $X, Y, Z \in \mathrm{Ob}(\mathbf{C})$ に対して

$$\mathrm{Hom}(Y, Z) \times \mathrm{Hom}(X, Y) \subseteq \mathrm{Arr}(\mathbf{C}) \times_{\mathrm{O}} \mathrm{Arr}(\mathbf{C})$$

となることがわかるし、更には

$$^{\forall} \langle g, f \rangle \in \mathrm{Hom}(Y, Z) \times \mathrm{Hom}(X, Y) \quad g \circ f \in \mathrm{Hom}(X, Z)$$

となることは簡単に確かめられる。

数学：図式の定義

前に進もうとしたが、図式の可換性の定義がサラリと文章で[*10]述べられていて、なんだかすこし曖昧な気がした。それでも基本的なアイデアは述べられているので、これを元に厳密な定義を考えてみることにした。

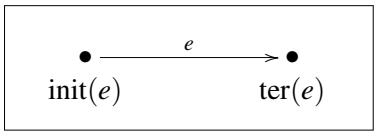
まずは図式を定義してみよう——。これは**有向グラフ (digraph)**——大雑把に言うと、**頂点**と呼ばれる点の集合 V の点を矢印で結んで得られる図形のこと——として定式化できそうだ：

有向グラフの定義

$G = \langle V, E, \mathrm{init}, \mathrm{ter} \rangle$ が**有向グラフ (digraph)** であるとは、次の条件 **(digraph)** が満たされることである：

(digraph)　　$\mathrm{init}, \mathrm{ter} \colon E \to V.$

有向グラフの定義における関数 init は辺——というか絵に書くときは矢印だが——$e \in E$ の**始点 (initial vertex)** を与え、関数 ter は辺 $e \in E$ の**終点 (terminal vertex)** を与える。そんなわけで、各 $e \in E$ に対して次のような矢印が対応していることになる：



これを真似して、圏における**図式**を次のように定義してみる：

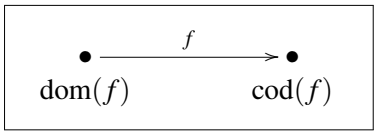
図式の定義

$\mathbf{C}$ を圏とし、$D := \langle V, E \rangle$ とする。ただし、$V \subseteq \mathrm{Ob}(\mathbf{C})$, $E \subseteq \mathrm{Arr}(\mathbf{C})$ とする。このとき、**D が $\mathbf{C}$ における図式である**とは $\langle V, E, \mathrm{dom}, \mathrm{cod} \rangle$ が有向グラフであることである。

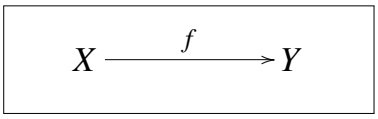
圏 $\mathbf{C}$ における図式 $D = \langle V, E \rangle$ に対して、常に次が成立する事に注意しておこう：

$$^{\forall} f \in E \quad \mathrm{cod}(f), \mathrm{dom}(f) \in V.$$

始点を返す関数として dom を、終点を返す関数として cod を持つような有向グラフを図式と呼ぶように定義したのだから、圏における射 $f \in \mathrm{Arr}(\mathbf{C})$ と $\mathrm{dom}(f), \mathrm{cod}(f) \in \mathrm{Ob}(\mathbf{C})$ の関係をこんな風に



図示しても良いが、これだと頂点の名前が見づらい。こんな場合、$X := \mathrm{dom}(f), Y := \mathrm{cod}(f)$ としてやり、さらに • をやめて



と描くようにしたほうがよほど分かりやすい。そんなわけで、集合と写像を抽象化し、さらに一巡りしてようやく見慣れた図に戻ってきたが、圏の図式という立場からはこれはもう「集合」と「関数」を意味しているとは限らない——。これらは「対象」と「射」からなる図式なのだ。

数学：図式の可換性

図式の可換性を定義するためにはどうしたらよいだろうか。どうやら可換性を定義するためには、「ひとつながりの矢印」を表す言葉があると便利そうだ。グラフ理論の本を探してみると**歩道 (walk)** という概念が見

[*10] CWM, p.8

つかった[*11]。

図式の歩道の定義

C を圏とし、$D := \langle V, E \rangle$ を **C** における図式とする。このとき、$f_1, \ldots, f_n \in E$ からなる n 重対 $\langle f_1, \ldots, f_n \rangle$ が D **の歩道である**とは、次の条件 **(walk)** が満たされることである：

$$\textbf{(walk)} \qquad {}^{\forall} i \in \{1, \ldots, n-1\} \quad \mathrm{cod}(f_i) = \mathrm{dom}(f_{i+1}).$$

歩道に含まれる射の個数 n を $\langle f_1, \ldots, f_n \rangle$ の**長さ (length)** と呼ぶ。

$$X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} X_2 \xrightarrow{f_3} \cdots \xrightarrow{f_n} X_n$$

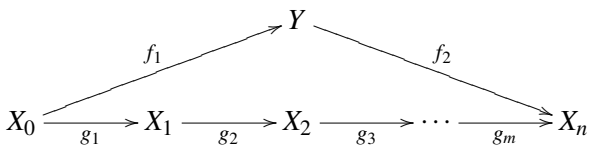
図式 D の歩道全体の集合を $\mathrm{Walk}(D)$ で表す。

三角形の図式の可換性は、長さ 1 の歩道 $\langle h \rangle$ と長さ 2 の歩道 $\langle f, g \rangle$ を合成して得られる $g \circ f$ が一致する事として述べられる。この例をよく検討して一般化すれば図式の可換性は次のように定式化できる——。

図式の可換性の定義

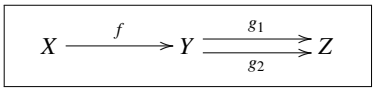
C を圏とし、$D := \langle V, E \rangle$ を **C** における図式とする。このとき、D **が可換である**とは、次の条件 **(com)** が満たされることである：

$$\begin{aligned} \textbf{(com)} \quad {}^{\forall} n \geq 2 \ {}^{\forall} \langle f_1, \ldots, f_n \rangle, \langle g_1, \ldots, g_m \rangle \in \mathrm{Walk}(D) \\ \mathrm{dom}(f_1) = \mathrm{dom}(g_1) \quad \text{かつ} \quad \mathrm{cod}(f_n) = \mathrm{cod}(g_m) \\ \Longrightarrow \quad f_n \circ f_{n-1} \circ \cdots \circ f_1 = g_m \circ g_{m-1} \circ \cdots \circ g_1. \end{aligned}$$



ただし、歩道 $\langle f_1, \ldots, f_n \rangle$ の長さ n が 1 のときは $f_n \circ f_{n-1} \circ \cdots \circ f_1$ は f_1 自身を意味するものと約束しておく。

地味だが条件 **(com)** において $n \geq 2$ という制限を置いていることは重要だ。この制限をおかないと $m = n = 1$ の場合も含まれてしまい、図式 $D = \langle V, E \rangle$ に含まれる任意の対象 $X, Y \in V$ に対して、$\mathrm{dom}(f) = X, \mathrm{cod}(f) = Y$ となる射 $f \in E$ はただ 1 つしか存在しないことになってしまい、可換な図式として許されるものが減ってしまうことになる。



例えば、今の可換性の定義の下では上の図式が可換であることは $g_1 \circ f = g_2 \circ f$ であることを意味する。しかし、**もし**可換性の定義において $n \geq 2$ という制限が**なかったとしたら**、上の図式が可換であることは $g_1 = g_2$ であることを意味するようになってしまう。

数学：関手 (Functor)

一応圏については理解したことにして、関手 (functor) の定義[*12] に進むことにした。名前から判断するに、きっと Haskell のファンクター型クラス `Functor` と関係しているのだろう。

[*11] 良く似た概念に**道 (path)** というものがある。グラフ理論の標準的な言葉遣いでは、**道**は辺や頂点の順序を含まない概念である。合成を考慮する都合上、ここでは**歩道**を利用することにした。

[*12] CWM, p.13, ただし説明の都合や好みなどから (ry

関手の定義

圏 **C**, **D** に対して、対象を対象に移す関数 F

$$F : \mathrm{Ob}(\mathbf{C}) \longrightarrow \mathrm{Ob}(\mathbf{D})$$

と射を射に移す関数（やはり F と書く）

$$F : \mathrm{Arr}(\mathbf{C}) \longrightarrow \mathrm{Arr}(\mathbf{D})$$

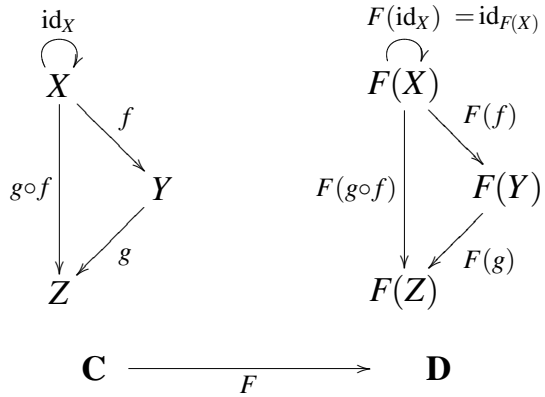
が与えられているとする。このとき、F が **C** から **D** への**関手 (functor)** であるとは次の **(func1)**-**(func3)** が成立することである：

(func1) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$.

(func2) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y) \qquad F(f) \in \mathrm{Hom}\big(F(X),F(Y)\big)$.

(func3) $\forall X,Y,Z \in \mathrm{Ob}(\mathbf{C})\ \forall \langle g,f\rangle \in \mathrm{Hom}(Y,Z)\times\mathrm{Hom}(X,Y) \qquad F(g\circ f) = F(g)\circ F(f)$.

関手というものがどう役立つかは解らなかったが、とりあえず可換図式を可換図式に移してくれるものだと考えておけばよいのだろうか：



Haskell：関手?

次に、この関手というものが Haskell の `Functor` 型クラスとどんな関係があるのか考えて見ることにした。

```
class  Functor f where
   fmap :: (a -> b) -> f a -> f b
```

という定義をみると、型コンストラクタ`->`は右結合性なのだからこんな風に書いても意味は同じのはずだ——。

```
class  Functor f where
   fmap :: (a -> b)  ->  (f a -> f b)
```

こう書いてみるとだんだん関手との関係がわかってきたような気がする。そう思って青い象の本を調べると少し先の方、11.2 に『ファンクター則』なんてものが載っていた。引用すると

(ファンクターの第一法則) `fmap id = id`
(ファンクターの第二法則) `fmap (f . g) = (fmap f) . (fmap g)`

つまり、`Functor` 型クラスのインスタンスが `fmap` を実装する際には上記の二法則を**守るようにしなければならない**ということだ。「ファンクターの第一法則」はちょうど **(func1)** に相当し、「ファンクターの第二法則」は **(func3)** に相当しているようだ。そして **(func2)** は `fmap` の型宣言と似ている——だから

Haskell の `Functor` 型クラスは、型を**対象**とし、関数を**射**とする圏 **C** から **C** への関手と関係している。

ということになりそうだ。……うーん、……「対象を対象に移す関数」と「射を射に移す関数」を同じ名前 *F* で呼んでいるせいか、対応が見えづらいような気がする……。
それにしても——何かしっくり来ない。新しく学んだ概念だから落ち着きが悪い——それだけのことなのかも知れない——。
でも——。何か見落としているような気がする。

少し雨が降り出していた。明日は晴れるだろうか。

12.3　海の家「れもん」にて

月曜日。いつもなら今日は羽川に勉強を見てもらう日なのだが、事前の連絡通り今日は一回休み。そんなわけで僕は海にいた。
もしかしたら読者の皆さんは「そんなわけで」が全然説明になってないと感じたかもしれない。
——確かに今日は本当は勉強をしないといけない日だし、本当は海に遊びに来るつもりはなかったのだが。もちろん家を出たときはちゃんと図書館で勉強するつもりでノートも羽川に借りた本も持ってきた。しかし、図書館に向かって自転車を漕いでいるときふと、こんなにも天気が良いし海に行ったらさぞかし爽快だろうなどと水着の用意もしていないくせに思いつき、そんな思いつきのままに駅に自転車を止めて電車に乗ってしまったのだ。
とは言うものの、泳ぐ用意もなかったため仕方なく僕は海水浴に興じる観光客を後目に海の家でノートと本を広げてぼんやりと考えていたのだった——。
こうして冒頭に書いたように僕はイカ娘に出遭った——という次第だ。

12.3.1　圏論（2）

Haskell：アプリカティブファンクター

昼飯時を過ぎているせいか、この時間になると客もまばらだった。最初に話しかけて少し言葉を交わしたあとも、イカ娘は店の手伝いをしながら僕のテーブルの側で立ち止まっては興味津々といった様子で僕のノートを覗きこんだりしていた。少し——いや、かなり気が散るのだが。それにしても、イカ娘は Haskell がそんなにも気になるのだろうか——。

ビーチではしゃぐ観光客の楽しげな声が耳に入るせいなのか——それとも店内を歩き回るイカ娘の華麗な触手さばきに時折目を奪われるせいなのか——あまり集中できないながら、僕は青い象の本の 11.3 を読んでいた。アプリカティブファンクター（applicative functor）の箇所だ。型クラス `Applicative` の主要な部分はこうだ——。

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

『`Functor f =>`』という箇所は『`f` が `Functor` 型クラスのインスタンスであるときだけ `f` は `Applicative` 型クラスのインスタンスになれる』という制約条件を表している。——というわけで、`Applicative` のインスタンスは `Functor` のインスタンスでもあるということになる。
そこまではわかる。
だが——なぜこんなものが必要なのだろうか？　それがよく分からなかった——。

分からないという状態にも色々なレベルがある——分からないポイントが自分で解っているレベルの「分からない」から、分からないのポイントが一体どれなのかがすでに判らないというレベルの「分からない」まで——。
自分が今直面してる「分からない」は多分その真ん中あたりだ。青い象の本にはアプリカティブファンクターはファンクターの強化版だと説明してある——まずここがわからない。——なぜ強化する必要があるの

だろう？　もし Functor 型クラスのインスタンスが型を対象とし、関数を射とする圏 **C** から自分自身への関手だというのなら、可換図式はそのまま可換図式に写るのだから、改めて強化する必要はどこにもないはずだ。

考えていてもよくわからず、集中力が——ますます——低下しているのが自分でもよくわかった。
——気分転換しよう。僕以外の客がいなくなってしまったタイミングで、あいかわらず僕のノートが気になる様子のイカ娘に話しかけてみた——。
「なぁ、Haskell に興味があるのか？」
まるで猫缶を見せられた人馴れした野良猫のように目を輝かせてイカ娘がいそいそとやってきた。
「Haskell になってからよく知らないから興味あるでゲソ！」
『Haskell になる』ってどういう意味だろう——。
まあいいや。
少し話をしてみると、イカ娘が卓越した数学の才に恵まれている事に否応なく気付かされた。彼女はどうやらあまりきちんとした教育は受けていない様子だったが、にも関わらず僕のノートや羽川から借りた本をパラパラと数分間めくっただけで僕の知識と理解に追いつき——それどころか僕が教えを請わねばならない立場になっていた。自分には数学の才能があるなどという自負心を——少しばかり——持ち合わせていたのだが、そんなプライドはこうして完膚なきまでに叩きのめされたのだった。——だが独りで勉強するのも少し飽きてきたところだったということもあり——格好の教師役を得た思いで色々教わることにした。
「どこがわからないか説明してみるでゲソ」
「いやさ、正直言ってどこがわからないかうまく説明できないというか……頭がぼんやりして理解できなくなっちゃうというか……」
「——じゃあその理解できなくなるのがどこか教えてみるでゲソ」
「うーん、それが難しいんだが……そうだな——じゃあ例えばだけど、なんで Functor 型クラスに強化版が必要なんだ？」
イカ娘がその質問を聞いて——意味がわからないとばかりに——首を傾げた。そうして僕のノートをもう一度パラパラとめくって何事かを確認してから——こう言った。
「多分——暦は、『型を対象として、関数を射とする Haskell の 圏』をちゃんと理解していないでゲソ」
僕は当惑した——その圏のことは十分に理解しているつもりだったからだ——。
「ええと——それはどういう事？」
「じゃあ——例えば Haskell の基本型が仮に $\mathbb{R}$ だけだったらどうなるか考えてみるでゲソ」
「その圏を構成してみせればいいのか？」
「そうでゲソ」
そんなの簡単だ——というより簡単すぎやしないか？　少し馬鹿にされてる気がした。
「——そうだな、じゃあ『型を対象として、関数を射とする Haskell の 圏』を **H** と呼ぶことにすると、$\mathrm{Ob}(\mathbf{H}) = \{\mathbb{R}\}$ となって——」
「——それはおかしいんじゃなイカ？」
「えっ……？」
「たとえばこんな関数宣言を考えてみるでゲソ——」

```
f :: (Real -> Real) -> Real
```

「——この関数 f の宣言は暦のノートにある数学記法で書けばこんな風になるはずでゲソ」

$$f\colon (\mathbb{R} \triangleright \mathbb{R}) \to \mathbb{R}$$

「……あれ？」
「——どこが変かわかったんじゃなイカ？」
「ええと——$\mathbb{R} \triangleright \mathbb{R}$ も $\mathrm{Ob}(\mathbf{H})$ に属してないとおかしい——？」
「その通りでゲソ。関数の矢印 $\to$ の両側は**型**のはずでゲソ。そして、すべての**型**は圏 **H** の対象——つまり $\mathrm{Ob}(\mathbf{H})$ に属する——から $\mathbb{R} \triangleright \mathbb{R} \in \mathrm{Ob}(\mathbf{H})$ でないとおかしいでゲソ」
言われてみれば——いや言ったのは僕か——その通りだった。なんでこんなことに気づかなかったんだろ

う。……ちょっとノートに書きながら考えてみよう。昨日まとめた箇所 12.2.5 を真似すれば基本型が $\mathbb{R}$ しかない場合の $\mathrm{Ob}(\mathbf{H})$ は——こんな風に構成することができるな……。

$$
\begin{aligned}
&O_0 := \{\mathbb{R}\}.\\
&O_{n+1} := O_n \cup \left\{ T_1 \triangleright T_2 \,\middle|\, T_1, T_2 \in O_n \right\} \quad (n \geq 0).\\
&\mathrm{Ob}(\mathbf{H}) := \bigcup_{n\in\mathbb{N}} O_n.
\end{aligned}
$$

「……まあ大体いいでゲソ」

——あれ——この言い方引っかかるぞ……

「もしかしてなんかまだ見落としがあったかな……？」

「$\mathrm{Ob}(\mathbf{H})$ の構成に使われてる型コンストラクタが $\triangleright$ だけなのは寂しいでゲソ」

「……ああ、なるほど——きちんと書くならば型コンストラクタの集合 K——ただし $\triangleright \in K$ を仮定しておく——を使ってこんな風に構成するべきだったか——」

$$
\begin{aligned}
&O_0 := \{\mathbb{R}\}.\\
&O_{n+1} := O_n \cup \bigcup_{k\in K} \left\{ k(T_1, ..., T_{\mathrm{ar}(k)}) \,\middle|\, T_1, ..., T_{\mathrm{ar}(k)} \in O_n \right\} \quad (n \geq 0).\\
&\mathrm{Ob}(\mathbf{H}) := \bigcup_{n\in\mathbb{N}} O_n.
\end{aligned}
$$

「それでいいでゲソ！」

「——あれ、でも待てよ……今の構成方法だと $\mathrm{Ob}(\mathbf{H})$ は型コンストラクタの集合 K のとり方に依存してしまわないか？」

「いいところに気づいたでゲソ。Haskell の型を対象とし、関数を射とする圏 $\mathbf{H}$ というのは実は確定したものではないでゲソ」

「じゃあ『僕が扱ってる $\mathbf{H}$』と『他の誰かが扱ってる $\mathbf{H}$』が違ってるかもしれないということか……」

「そういうことになるでゲソ」

「でもそれは少し気持ち悪いな……」

「暦は数学を気にし過ぎなんじゃなイカ？　確かに圏論は Haskell の挙動を合理的に設計するため使われているでゲソ。でもそれは——Haskell がすべての面で数学の習慣に従わなければいけないという事を意味しているわけではないでゲソ。それに、物事は厳密に論じれば理解しやすくなる——とも言い切れないでゲソ。」

「なるほど、物事を論じるには適切なレベルがあるということか……数学的に完璧に定式化しようと拘る事が却って物事の理解が妨げになっているようでは本末転倒だな。なんとなく痛いところを突かれた気がするよ」

「そんな凡庸な感想は無視して、先へ進むでゲソ」

「堂々とスルーされた！」

「まずは関手の定義をもう一度丁寧に見るでゲソ。暦も少し気になってるようでゲソが、Haskell への応用を考える場合、関手 F の『対象を対象に写す関数』と『射を射に写す関数』は名前を分けた方が結局は分かりやすいと思うでゲソ」

そう言ってイカ娘は僕のノートの新しいページに次のように書いた：

関手の定義 (object-part と arrow-part を分けて記述したものでゲソ)

圏 $\mathbf{C}, \mathbf{D}$ と関数

$$F_O : \mathrm{Ob}(\mathbf{C}) \longrightarrow \mathrm{Ob}(\mathbf{D}),\ \ F_A : \mathrm{Arr}(\mathbf{C}) \longrightarrow \mathrm{Arr}(\mathbf{D})$$

が与えられているとするでゲソ。このとき、$F := \langle F_O, F_A \rangle$ が $\mathbf{C}$ から $\mathbf{D}$ への関手であるとは次の **(func1')**-**(func3')** が成立することでゲソ：

(func1') $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad F_A(\mathrm{id}_X) = \mathrm{id}_{F_O(X)}$.

(func2') $\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad F_A(f) \in \mathrm{Hom}\big(F_O(X), F_O(Y)\big)$.

(func3') $\forall X, Y, Z \in \mathrm{Ob}(\mathbf{C})\ \ \forall \langle g, f \rangle \in \mathrm{Hom}(Y,Z) \times \mathrm{Hom}(X,Y) \qquad F_A(g \circ f) = F_A(g) \circ F_A(f)$.

「……少し気になったんだけどさ、お前っていつもこのレベルで言文一致なの？」

「何かおかしいでゲソ？」

「……いやさ、この語尾まで含めて言文一致って相当変わってると思うけど——特に手書きのときは。ごめん、気にしないで続けてくれ……」

「——とにかく、こんな風に分けて書いてみると Haskell の `Functor` 型クラスとの関係がはっきりするんじゃなイカ？」

少し考えてみた。**(func2')** を書きなおすと

$$F_A\colon \mathrm{Hom}(X,Y) \to \mathrm{Hom}\big(F_O(X),F_O(Y)\big)$$

となる。そして、今考えている圏 **H** では

$$\mathrm{Hom}(X,Y) = \mathrm{Map}(X,Y) = X \triangleright Y$$
$$\mathrm{Hom}\big(F_O(X),F_O(Y)\big) = \mathrm{Map}\big(F_O(X),F_O(Y)\big) = F_O(X) \triangleright F_O(Y)$$

なのだから、結局

$$F_A\colon X \triangleright Y \to F_O(X) \triangleright F_O(Y)$$

となる。ここまで来れば

```
fmap :: (a -> b)  ->  (F a -> F b)
```

との関係は明らかだった——。

「——なるほど、`fmap` は F_A に対応しているのか」

「そして関手 F の object-part F_O は型パラメータを 1 つ取る型コンストラクタになっているでゲソ」

圏論の関手と Haskell の `Functor`

F_O：関手の object-part	`Functor` のインスタンスの型コンストラクタ
F_A：関手の arrow-part	`Functor` のインスタンスが実装する `fmap`

「この辺のことは薄ぼんやりは理解していたけど今はっきりと理解できたみたいだ」

「『わからない』にも色々な段階があるように——『わかる』にも色々な段階があるでゲソ」

「なるほど……いや勉強になります……」

「本題のアプリカティブに進む前に、なるべく記号を簡単にしたいから、関手 F の object-part は同じ記号 F で、arrow-part は——できるだけ——区別して F_A と書くようにしなイカ？」

「うん、そうしよう。F と書いたときにそれが関手自身なのか関手の object-part なのかは文脈で判断できそうだからな。——混乱しそうなときは必要に応じて F_O という記号を使うようにすればいい」

「——早速でゲソが——こんな関数があったとするでゲソ——」

```
f :: X -> Y
```

「`X` と `Y` は何らかの具体型ってことだよな——」

「そうでゲソ。そして、`F` が Haskell のファンクターだとして、この `f` を元にこんな関数 `f'` が欲しいとするでゲソ——」

```
f' :: F X -> F Y
```

「——それだったら `fmap` で作れる——」

「その通りでゲソ。じゃあ今度は `f` と `f'` がこんな形だったらどうなるか考えてみるでゲソ」

```
f :: X -> Y -> Z
f' :: F X -> F Y -> F Z
```

それも `fmap` で——と即答しそうになったが、この質問は罠の臭いがする。よく考えてみよう。

$$f \in X \triangleright Y \triangleright Z = X \triangleright (Y \triangleright Z)$$

なので

$$F_A(f) \in F(X) \triangleright F(Y \triangleright Z)$$

となるが、$F(Y \triangleright Z) = F(Y) \triangleright F(Z)$ **であるとは一般には言えない**。
このことは具体例を考えれば分かる。`Maybe(X -> Y)` は、集合としては

$$\Big\{\texttt{Nothing}\Big\} \cup \Big\{\ \texttt{Just f} \ \Big|\ \texttt{f :: (X -> Y)}\ \Big\}$$

だが、`(Maybe X) -> (Maybe Y)` には、ある `y0 :: Y` に対して

```
g :: (Maybe X) -> (Maybe Y)
g _ = Just y0
```

と定義できる関数、つまりどんな引数に対しても `Just y0` を返すような関数も含まれている。明らかに、これは `Nothing` とも違い、しかもどの `f :: X -> Y` に対する `Just f` とも違う。
——こう考えてみると——。
「`fmap` だけじゃダメみたいだな」
「その通りでゲソ。——まずは、アプリカティブの`<*>`の型を見るでゲソ——。

```
(<*>) :: f(a -> b) -> f a -> f b
```

「これを今の状況に合わせて数式で書きなおせばこうなるでゲソ。

$$\text{（二項演算子）}\quad \langle * \rangle \quad : F(Y \triangleright Z) \times F(Y) \to F(Z).$$

「だから、$^\forall x \in F(Y \triangleright Z)$ と $^\forall y \in F(Y)$ に対して——

$$x \langle * \rangle y \in F(Z)$$

となるでゲソ。でも——これをこんな風に解釈することもできるんじゃなイカ？」

$$x \langle * \rangle \in \mathrm{Map}(F(Y), F(Z)) = F(Y) \triangleright F(Z).$$

「ええと——その $x\langle * \rangle$ というのはラムダ記法で書けば

$$x \langle * \rangle = \lambda y \mapsto (x \langle * \rangle y)$$

——ということ？」
「そういうことだけど——要するにただの部分適用じゃなイカ？」
「いや——言われてみればそうなんだけど、何故か二項演算子は部分適用できないように思い込んでたみたいだ」
「確かに、初めて見るとぎょっとするかもしれないでゲソ。でも、理屈は普通の関数と同じだし、要するに慣れの問題なんじゃなイカ？」
うーん……どうだろう。実際に慣れてみないとその感覚はわからないな。——あれ……？
「——今気づいたんだけど、これって x の後に関数記号 $\langle * \rangle$ が来ているから、つまり**後置関数**になってる！

$$\text{（後置関数）}\quad \langle * \rangle \quad : F(Y \triangleright Z) \to F(Y) \triangleright F(Z).$$

「その通りでゲソ」
「いやしかし、土曜の夜に思いつきで書いた後置関数の話題がここでつながるのか。こういうのを予定調和っていうんだろうな」
「……伏線を回収できたのがそんなに嬉しかったでゲソか……」

続きを考えてみよう。$x' \in F(X), y' \in F(Y)$ とすると——先程確認したように——$F_A(f) \in F(X) \triangleright F(Y \triangleright Z)$ だから

$$F_A(f) x' \in F(Y \triangleright Z)$$

なので、$\langle * \rangle$ を使えば

$$F_A(f)\,x'\,\langle * \rangle \ \in F(Y) \triangleright F(Z)$$

となる。従って、

$$F_A(f)\,x'\,\langle * \rangle\,y' \in F(Z)$$

であることがわかる。最後の式の左辺を Haskell 記法で書けば

```
(fmap f) x' <*> y'
```

となる。更に、`Applicative` 型クラスにおける F_A つまり `fmap` の後置関数バージョン`<$>`を使えば、

```
f <$> x' <*> y'
```

とも書ける。そんなわけで――。

「さっきの問題の答えは

```
f' :: F X -> F Y -> F Z
f' t u = f <$> t <*> u
```

だな」

「もっと引数が多い場合はどうなるでゲソ？」

「ここまでくればもう簡単だ。

```
f :: X1 -> X2 -> X3 -> ⋯ Xn -> Y
```

の引数の型を全部『F 付き』にしたものはこんな風に定義できる：

```
f' :: F X1 -> F X2 -> F X3 -> ⋯ F Xn -> F Y
f' x1 x2 x3 ⋯ xn = f <$> x1 <*> x2 <*> x3 <*> ⋯ <*> xn
```

」

「どうやらちゃんとわかったみたいじゃなイカ」

「いや、本当に助かったよ」

この後も話は弾み、モナドに関連して自然変換の話まで教わったのだが、機会があればこの話も書いてみたい[*13]。

12.4　死闘

12.4.1　商店街

夏の長い午後も終わりかけていた。少し涼しくなり、いかにも海水浴といった風情の客は減り始めた。気の早い一部の客が花火と消火用のバケツを持ちだしていたが外はまだ明るかった。もう少し待てば素敵な夕日を眺められそうだった。

夕食を作って待っているであろう妹たちの事を考えると――それでも少し普段より遅くなりそうだったが――そろそろ帰らねばならない時刻だった。駅まで送ってくれると申し出てくれたイカ娘と僕は小ぶりな商業用ビルやシャッターの降りた店が並ぶ静かな商店街をゆっくりと歩いていた。
少ないながらも人の出入りが絶えなかった海の家から離れ、僕らは二人きりになっていた。何故かそんな事を急に意識してしまい、横で歩いているイカ娘は僕のことをどう思っているのだろう――などと埒もない事に思いを巡らせ始めたとき――イカ娘が突然足を止めた。

[*13] 残念。時間切れである。

イカ娘の表情が殺気立っていた。待て、イカ娘はこんなにも——剣呑な雰囲気をまとった奴だったろうか？ 僕は身の危険を感じた。怪異が人の姿をしており人と同じような感情を持っていたとしてもそれが怪異である以上何がしかの人知を超えた力を持つ存在なのであり、下手に感情を刺激することは命取りになりかねない。怪異に遭遇する人間は稀だとはいえ、怪異との接し方で生死が左右される事は珍しいことではないのだ。幾つかの怪異に接しているうちに、僕はいつしか怪異慣れ——悪い意味で——してしまっていたのかも知れない。イカ娘がどんな力を持っているのかはわからないが、しばらく忍に血を吸わせていない僕は吸血鬼としての最大の力を発揮することはできない。
もっとも、吸血鬼としての力が弱っているからこそ真夏の日差しの中を歩いても平気なのだが。
僕の影の中から忍が様子を伺っている気配が感じられたがこの危機的な状況に応じて何かしてくれる様子はなかった。もちろん彼女には僕を助ける義理もないのだ。必死で打開策について思いを巡らしている探す僕を無視して、イカ娘は静かな声でこう言った——。
「すごいニンジャ・アトモスフィアを感じる……でゲソ」
その頃になって僕も周囲を囲繞するただならぬ殺意に気付いた。まとわりつくような殺意。その恐るべき必殺の悪意はイカ娘が睨みつける方向から放たれていた。
「何者でゲソ？」
イカ娘がまだ姿を見せていない相手に呼びかけた。数十メートル先——路上にわだかまっていた不定形の黒っぽい霧のようなものが凝集して黒装束の男の形を取った。男が出現するのに合わせたかのように周囲が薄暗くなった。同時に夏の暑気は急速に薄れてゆき——それどころか肌寒さすら感じるようになっていた。
忍者——だろうか。男の姿は時代劇などで見かける忍者のような衣装をまとっていたが、コスプレ会場や撮影現場から抜けだしてきたという印象は全く受けなかった。——男がまとう漆黒の衣装から漂う間違え様のない血臭は、男が正真正銘本物の殺人者であることを物語っていた。

男の素顔の半分を黒い面頬で隠しており、その両頬には「蓮」「殺」という赤い文字が妖しく光っていた。男はこちらに進み出ると軽く一礼し
「ドーモ、ハスケルスレイヤーです」
と挨拶した。男は軽い一瞥を僕にくれたがどうやら取るに足らない雑魚だとみなしたのか、僕に対しては名乗らなかった。次に男はこう叫んだ——。
「Haskeller 死すべし！ 貴様をケジメする！」
これに応じてイカ娘は
「いきなり現れて——」
その台詞を最後まで聞き終える事はできなかった——なんの前触れもなく、唐突に全ての感覚を失ったからだ。

12.4.2 商店街（闇）

無聲の闇に放り出された僕は自分でも意外なほど落ち着いており、状況についてゆっくりと考えていた。五感はまだ回復していなかったが——なぜか忍の気配を感じた。——ふと、ここは忍のいる影の世界なのではないかと思った。やがて、ゆっくりと五感が戻ってきた。場所は今さっきまでいた商店街のはずだったが依然として非常に暗く、誰の姿も見えなかった。物の見え方が少しおかしかった。近くの建物に目を凝らすと急速に建物が遠ざかるような感じがした。周囲の物体は月光を受けてか仄かな銀光を帯びており、その微細な銀の光の粒たちによって物体の輪郭を認識できるのだった。
——ふと気になって月を見上げると、それは見慣れた月ではなく、夜空に高く輝く黄金の立方体だった。その立方体からは蜘蛛の糸のような銀の細かい繊維が数限りなく伸びており、それらは立方体から遠ざかるにつれて急速に夜空に溶け込んでいた。
「起きたか」
突然話しかけられて僕は振り向いた——が誰もいなかった。
「無駄じゃ。この空間では儂は形を纏っていない」
久しぶりに聞く忍の声だった。
「ここは影の世界じゃないのか？」

僕は先程から思っていたことを口に出した。
「うぬとは口を聞かん」
「話しかけてきたのはそっちじゃないか！　ひょっとしてツンデレキャラに路線変更でもしたのか？」
「バーバヤガからの伝言じゃ。『観察し、書きとめよ』とのことじゃ」
「バーバヤガって誰だそれ？」
「知らん」
「『観察』って何をだ？」
「知らん」
「一体どうなってるんだ？」
——最後のは質問じゃなく独り言だ。
「始まるぞ」
との声に促されて商店街に再び目をやると、商店街には二つの人影があった。——最初にこの闇の商店街を見たときには誰もいなかったはずだが。

僕の傍らにはいつの間にかイカ娘が立っていた。やや距離をおいた向こうには忍者装束の——先程「ハスケルスレイヤー」と名乗った——男が立っていた。二人とも彫像のように静止していた。二人の姿は基本的には先程普通の世界で見た姿と似ていた——だが対比をなすような違いもあった。この闇の世界の中では男は二回りほど体が大きく、禍々しく陰惨な闘気をその身に滾らせていた。イカ娘は——美しかった。現実世界より頭身が高く、体全体を循環する微細な銀色の粒子によって体全体が輝いていた。
——突然、まるで一時停止が解除されたかのように二人の体に動きが戻った。イカ娘が叫んだ
「——なに理不尽な事を言ってるでゲソ！」
対峙する二人と僕はどうやら少しだけずれた時間軸に置かれていたらしい。そしてたった今僕は戦いの舞台を観察し記録すべく同期する時間軸に置かれた——そういうことらしかった。

12.4.3 ◆◆◆◆イカ娘 vs ハスケルスレイヤー◆◆◆◆

漆黒の闇を纏った禍々しいニンジャ、その名はハスケルスレイヤー。そのメンポの左右に刻まれた、鮮血よりなお赤い朱文字は「蓮」「殺」。溢れんばかりの殺意を宿すその視線の先にはイカ娘。このコトダマ空間におけるイカ娘は長身で女神めいた白銀の輝きを帯びており、その胸も豊満であった。
「狙いはなんでゲソ！」
「貴様をケジメする……。Haskeller は……死なねばならん！」
「それが理不尽だと言ってるでゲソ！」
ハスケルスレイヤーがメンポの奥で顔を嘲弄に歪めた。そして驚愕の真意を語り出した。
「……理不尽ではない。Haskell の高度な表現能力を獲得したコトダマ空間の情報寄生体が際限なく自己増殖を繰り返した結果、このコトダマ空間は情報密度臨界に近づいている。臨界点を超えればこのコトダマ空間が相転移を起こし一気に爆発する。そうなれば、お前たちが現実と呼んでいる世界も破滅するのだ」
「何言ってるかわからないし、この変な世界が終わると私の世界も破滅するなんておかしいじゃなイカ？」
「……だから貴様達は何もわかっていないと言うのだ。貴様たちが現実と呼んでいる世界はこのコトダマ空間の中に実装されている箱庭空間に過ぎない。したがってこのコトダマ空間の終焉とともに貴様達の宇宙も終わる」
ゴウランガ！　ハスケルスレイヤーの口から語られる壮大な宇宙観！
「お前の言ってる事は妄言でゲソ！　なにも証拠がないじゃなイカ！」
「証拠ならある……私が証拠だ。未来世界においてすでにコトダマ空間の相転移は起こったのだ。その影響で貴様達にとっての現実である箱庭宇宙の計算力学系の不動点が移動した。その結果、全ての物理定数が変動し、宇宙はビッグ・クランチによる終焉を迎えた。私は保存されていた継続を利用してスタックフレームを巻き戻し、歴史を修正するために過去の宇宙にに戻ってきたのだ」
「さっきから何を言っているかわからないでゲソ！　それに、もしそうだとしてもそれはこの宇宙とは違う並行世界での出来事でゲソ！　こっちの宇宙も破滅するとは限らないじゃなイカ！」
「貴様が納得するかどうかは問題ではない。この宇宙を破滅から守るためにすべての Haskeller をケジメす

る。貴様も例外ではない。名乗れ。そして名誉ある死を迎えるがよい！」
これ以上話しても無駄だとようやく悟ったのか、イカ娘がこの求めに応えて決断的にアイサツした。
「ドーモ、イカ娘です――。お前をぶちのめす！」
先に仕掛けたのはハスケルスレイヤーであった。イカ娘がオジギを終えた瞬間に二本のクナイ・ダートを投擲！　だがどうしたことだろう？　クナイはイカ娘に向かうどころか垂直に近い仰角で投げられている！　ハスケルスレイヤーとて男、コトダマ空間におけるイカ娘のわがままなボディに心ならずも気をそらされ手元が狂ったのだろうか？　続けざまに地を這うように前転しつつクナイを投擲する。だがまたしてもクナイは大きめの仰角で投げられている！　そして再び前転しつつクナイを投擲！　そしてようやくコントロールを取り戻したのか、最後に投げられた二本のクナイはまっすぐにイカ娘を目指して飛翔する！　だが見よ、最後に投げられた二本のクナイがイカ娘に近づくにつれて大仰角で投擲されたクナイたちが描く放物線が次々と最後のクナイに合流しつつある！　前転しながらの不安定な姿勢での連続投擲にも関わらずのこの驚くべき精密制御である――なんというワザマエ！
合計八本のクナイがイカ娘に命中するかと思われた――その刹那！　すべてのクナイはイカ娘の把握力を持つ触手に捉えられていた！　タツジン！　ハスケルスレイヤーのワザマエも見事だがイカ娘の神速かつ精密な触手さばきもまたスゴイ級である。息を付く間も与えずイカ娘がその触手に捉えられたクナイを全て同時に猛スピードで投げ放った。自らの凶器により絶体絶命の境地に落ちるハスケルスレイヤー。インガオホー。

速やかに迫る死の脅威に対してハスケルスレイヤーは瞬時に身を構え――体を振り絞るように放ったのはただ一発のツキ・パンチ――。
全身の筋肉が引き絞られる――加速度ベクトルが合成されるその極みには恐るべき豪速の拳！
――いまやハスケルスレイヤーの拳は超音速に達している！　右の拳を頂点に形成された衝撃波コーンと衝突した八本のクナイ・ダートは甲高い金属音を立てて崩壊した！　クナイ・ダートが持っていた運動量自体はさほど減殺されなかったとはいえ、分散して飛来する欠片にはもはや当初の殺傷力はない。ハスケルスレイヤーが軽く顔をガードしただけで飛来するクナイ・ダートの欠片はたちまち無力化されてしまう。そしてハスケルスレイヤーのツキ・パンチが生み出した衝撃波はそのままイカ娘に襲いかかった！　まさに攻防一体、ミヤモト・マサシの思想を具現化したタイチー・カラテの極意である！　コワイ！

イカ娘は咄嗟に体の正面に触手を束ねて壁を作る。続いて轟音とともに襲来する衝撃波――だが衝撃波はテンタクル・ウォールに吸収されてイカ娘には効かない！　この無力化は彼女が軟体動物だからこそ出来るジツである。もし彼女の体に一箇所でも硬い部分があったなら先ほどのクナイ・ダートのように衝撃波の破壊力に巻き込まれていたことだろう。軽業師めいた精密動作をメインとしたジツをしかけてくると思わせつつ豪快な超音速ツキ・パンチを放ったハスケルスレイヤーのコン・フーはもはやヤバイ級の域に達している――だが、瞬時に相手の攻撃を見極めて最小限の防御で防ぎ切ったイカ娘の冷静な判断もまた、彼女が見かけによらない熟練の戦士であることを物語っていた。

思いがけず先に油断したのはハスケルスレイヤーであった。イカ娘がテンタクル・ウォールを解除する際の動作が必要以上に鋭く、触手が微かに空気を切り裂いていたのを見逃したのである。高速運動する触手先端で生み出された真空の渦が音もなくハスケルスレイヤーに忍び寄り、次の瞬間全身を切り裂いた！　ナムサン！
「グアーッ！」
悲鳴を上げるハスケルスレイヤー！　全身の切創はいずれも深く、噴水めいた勢いで血が噴き出している。――だが、ハスケルスレイヤーは予想外にも手負いのスモトリめいた推進力を発揮し、目にも留まらぬ速さで連続ツキ・パンチを繰り出しながら襲いかかる！　これを己の拳と触手で迎え撃つイカ娘！
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
二本の腕しか持たぬハスケルスレイヤーと、両腕に加えて十本の触手を持つイカ娘。これまでの戦闘から伺

える実力を考慮すると、手数で上回るイカ娘が優位なのは明らかである——。だが奇妙にも両者の応酬は拮抗していた。ナンデ？ 注意深く観察するとその理由は明らかであった——イカ娘は全ての触手を戦闘に回していない！ 十本の触手のうち四本は背中側にまわっており、戦闘とは無関係の奇妙で複雑な動作をしていた。勝利を確信したイカ娘が手を抜いて応戦することで相手を侮辱しているのだろうか——いやそうではない。見よ！ 背中側の触手が動きまわるのにあわせてこぼれ出す銀の粒は複雑な軌跡を空間に残し、今や複雑な魔法陣を形成しつつあった。ハスケルスレイヤーはイカ娘がなにか仕掛けようとするのを察知したのかトドメを刺そうと一層鋭い攻撃を仕掛ける！
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
だがついにイカ娘が編んでいた魔法陣は完成し、周囲のコトダマ・エネルギーを吸収して起動！
突如虚空に緑色の光点の群れが湧き出し、中空の一平面上に整列して文字を形成した——。

```
GHCi, special version for KOTODAMA-IRC space :? for help
Loading package haskell-karate ... linking ... done.
ghci>
```

イカ娘が出現した光点スクリーンの上で触手を滑らせ文字を入力する——。

```
ghci> haskell_slayer <- return $ Just haskell_slayer
\noindent
```

入力された文字が爆発的な光輝を帯びた。それに合わせるようにスクリーンが巨大化し、周囲の空間を巻き込みながら湾曲してゆく。湾曲の勢いは止まらず、いまや円軌道の残像を残す光点のパイルからなるトルネードに成長している。トルネードの鋭い先端が象の鼻のように持ち上がり——ハスケルスレイヤーの体に突き刺さった！

物理的な衝撃を予期して身構えるハスケルスレイヤーの目に当惑の表情が浮かんだ。トルネードが突き刺さると同時に緑色の光点は肉体の厚みなどまったく無視してハスケルスレイヤーの体に素早く浸透し、微生物めいた動きで泳ぎまわっていた。トルネードの光点がすべてハスケルスレイヤーの体に注ぎ込まれると、光点の群れはハスケルスレイヤーの体の中でますます輝き、やがて急速に消滅していった。同時にハスケルスレイヤーの全身の輪郭が徐々に曖昧になり、急速に不透明度を失ってゆく。ここまで読んできた読者には何が起こったのかよく理解できることであろう。——そう、ハスケルスレイヤーは今や `Maybe` モナドに閉じ込められて $\mathrm{Ob}(\mathbf{H}^{\mathrm{Maybe}})$ に属する存在と化しており、$\mathrm{Ob}(\mathbf{H})$ に属する我々とは文脈がずれてしまったのである。もはや向こうからは何も手が出せない。
これが、あっけない死闘の幕切れであった。

12.4.4 別れ

一瞬めまいを感じた。周囲が明るくなった。微かな潮香を感じる——元の空間に戻ってきたのだ。濃厚だったニンジャ・アトモスフィアも薄れていた。元の空間に戻っての僕の最初の感想は——
「あっ、一人称に戻ってる！」
——だった。
イカ娘は——見たところ無傷だった。やや戦闘の興奮が残っているのか微かに頬を上気させている。ハスケルスレイヤーが先程立っていた場所には誰もいなくなっていた。だが、目をついと左右にずらすと微かな陽炎のようなゆらめきがあり、男の姿らしき輪郭をなしているのがわかった。僕は不安を感じてイカ娘に訊いた——。
「あれはもう危険じゃないのか？」
「大丈夫、あそこから出てくることは決して出来ないでゲソ」
「本当に——？」
「`Maybe` モナドによって文脈が変化して『いるかもしれないし、いないかもしれない』存在になってしまっ

たから、もうこっちに干渉することはできないでゲソ」
「なるほど――」
とは言ったものの、僕はまだ不安を感じていた。――だがそれはそれとして。
「それにしてもさっきの技は凄かったな」
「あれくらいの相手、私にとってはベイビー・サブミッションでゲソ！」
「アイエエエエ！　ニンジャ・アトモスフィア消えてない！」
――僕は悲鳴を上げてしまった。

帰りの電車に揺られながら、僕はつらつらとハスケルスレイヤーの事を考えていた。イカ娘はハスケルスレイヤーが出てこれないと自信を持っていたようだが、Maybe はモナドだから――いや、やめておこう。考えても仕方がない。世の中の問題は、大抵は完全には解決できない。Maybe に閉じ込められたハスケルスレイヤーは、いつの日かこちらに『干渉してくるかもしれない』と考えるべきだ。

12.5 エピローグ

後日談というか、今回のオチ。
水曜日。僕は羽川の出したテストに合格し、面目を保つことができた。――イカ娘のおかげだ。羽川は興味深げに僕のノートをチェックして
「んん？ ――へぇ、Cartesian Closed Category [*14] まで行かなくてもこういう話が出来るんだね。――ちょっと意外だったかな」
――という感想をくれた。意味は解らなかったがとりあえず書いておく。

戦場ヶ原にも今回の顛末を話した。彼女の感想は――
「阿良々木くん。――どうもイカ臭いと思ってたらそんな子と会ってたのね」
――だった。
なんというか。
相変わらずだよなあ。

参考文献

この記事の執筆にあたり、以下の諸作品からキャラクターや設定を拝借いたしました。素晴らしい作品を発表されている原作者様各位に敬意を表明いたします。勝手なキャラクターや設定の拝借および改変については何卒ご寛恕下さいますよう願い申し上げます。

12.5.1 ネタ元

- 安部真弘「侵略！ イカ娘」

- 西尾維新〈物語〉シリーズ
  特に「化物語」（上・下）、「傷物語」、「偽物語」（上・下）

- ブラッドレー・ボンド、フィリップ・ニンジャ・モーゼズ「ニンジャスレイヤー」
  http://d.hatena.ne.jp/NinjaHeads/

また、執筆にあたって以下の文献を参考にいたしました。

12.5.2 参考にした教科書および専門書

[1] 結城 浩「数学ガール」シリーズ　ソフトバンク・クリエイティブ
本記事の直接のネタ元というわけではありませんが、主人公「僕」が周囲の人間を巻き込みあるいは巻

[*14] CWM, p.95

き込まれながら、対話を交えた試行錯誤のなかで数学を学んでいくというスタイルには多大な影響を受けました。

[2] 松村 英之 「集合論入門」 朝倉書店 (1966)
本記事が前提としている数学独特の言い回しや記号遣いに不慣れの場合は、「日常語としての集合論」についての教科書に目を通しておくことをおすすめします。そのような教科書は数限りなくありますが、個人的な好みからこの松村先生の本を推薦いたします。数学の諸分野で日常語として使われる集合論が、圏論への誘導を意識して説明されています。

[3] Miran Lipovača 「すごい Haskell 楽しく学ぼう！」, 田中英行・村主崇行 共訳, オーム社 (2012/5)
プログラミング経験者向けの Haskell 入門書として評判の高い
“Learn You a Haskell for Great Good!: A Beginner’s Guide”, No Starch Press (2011/4)
の日本語訳です。とても読みやすく訳されていると思います。なお、英語原文は Web で全文を読むこともできます：`http://learnyouahaskell.com/`

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman “Compilers: Principles, Techniques, and Tools”, Addison-Wesley (1986/1)
表紙の絵から「ドラゴンブック」と呼ばれて親しまれたコンパイラの教科書です。型や型コンストラクタについての一般論を書く際にこの本の 6.1 を参考にしました。

[5] S. Mac Lane, “Categories for the Working Mathematician”, Springer-Verlag (1971)
よく読まれている圏論の教科書です。何の説明もなく「CWM」と言う略称で呼ばれることもあります。すでに何らかの分野における数学の専門知識を持っている読者を想定読者としているため難しいところも多いのですが、圏論について真剣な興味があるなら持っていて損はありません。日本語訳も『圏論の基礎』というタイトルで出版されています。

[6] J. L. Bell, “Toposes and Local Set Theories”, Oxford University Press (1988)
図式の定義や可換性の定式化はこの本を参考にしました。現在は Dover から安価なレプリントが入手可能です。

[7] R. Diestel, 「グラフ理論」 根上生也・太田克弘 共訳, シュプリンガー・フェアラーク東京 (2000)
“Graph Theory” 2nd ed., Springer-Verlag (2000)
の日本語訳です。グラフ理論に関係する事項を書く際に参考にしました。

12.5.3 追記

- 初版原稿の作成時に、`@xhl_kogitsune` 氏にはとりわけ丁寧な査読をして頂き、幾つもの有用なコメントを賜りました。おかげで、説明の方法や話の流れをかなり改善できました。この場を借りて御礼申し上げます。
- 第二版の印刷に際し、幾つかの誤植を訂正し、数式の微調整を行いました。また、気のついた箇所の文言を再度整えました。大幅な内容の追加はしていませんが、p.117 において、初版における明らかな説明の欠落と思われた id_X についての簡単な記述を補いました。

第13章

RTS海溝二万マイル

— @master_q

みんな毎日 GHC で元気にはすはすしてるでゲソ？ Hackage には色々なパッケージが登録されていて、それらの使い方を調べると新しい知見が得られ、読んでるだけで楽しいでゲソ。

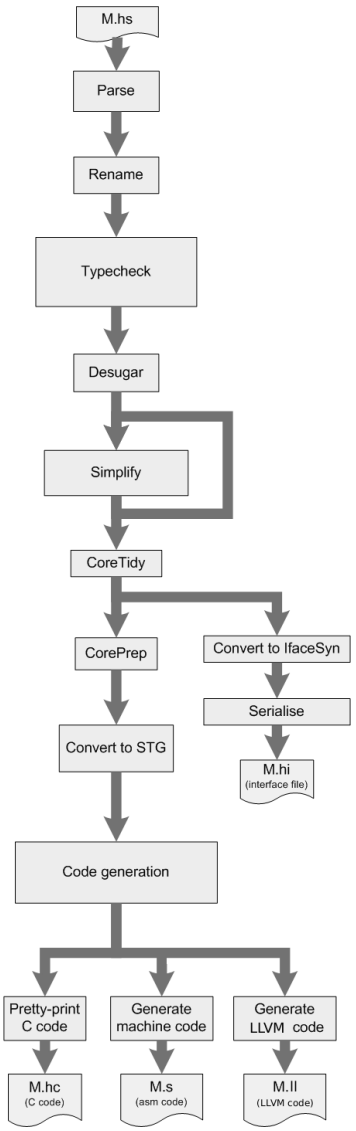
ところで、こんな楽しい世界 (GHC) がどんなしくみで動いているのか知りたくなイカ？

13.1 RTS は Haskell コードを実行する VM でゲソ!

一言に GHC と言っても中は広いんでゲソ。今日はその中でも RTS(Runtime System) というモジュールの中を探検してみようと思うでゲソ。

ところで RTS ってなんでゲソ？ ちょっと Haskell から離れて他の言語の実行モデルを思い出してみるでゲソ。C 言語の場合、ソースコードをコンパイラにかけるとアセンブリ言語になり、アセンブラを通すことで機械語になるでゲソ。この機械語同士をリンクすれば実行バイナリができるでゲソ。Java 言語の場合、ソースコードをコンパイラにかけると機械語ではなくバイトコードになるでゲソ。このバイトコードは Java VM という仮想マシンの上で実行できるでゲソ。その Java VM 自体は C 言語や C++ 言語など Java 以外の低レベルな言語で設計されることが多いでゲソ。Perl 言語の場合、ソースコードはコンパイラにかけず、Perl インタープリタ上で実行するでゲソ。そのインタープリタはやはり C 言語のような低レベル言語で設計されているでゲソ。

さて、Haskell に戻ってみるでゲソ。GHC は Haskell のソースコードを右の図のようなコンパイルパイプライン[*1] を通して機械語に変換するでゲソ。他の言語と比較すると C 言語のケースと似ているでゲソね。この機械語化された Haskell の関数をつなげれば、Haskell で作ったプログラムの意図通りに動作をする実行バイナリができるはずでゲソ。じゃあ、Haskell で書かれたソースコードはこのパイプラインを通してオブジェクトファイルを作り、それらだけをリンクすれば実行バイナリになるのカ、というとそんなことはないのでゲソ。以下のようなプリミティブは (少なくとも現時点では)Haskell で書くことができていないでゲソ。



- コンパイル済み Haskell 関数の実行
- forkIO などで作った Haskell スレッドのスジューリング
- 並列実行

[*1] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscMain

- メモリ管理
- GC
- STM

既存の C 言語ライブラリでも Haskell ライブラリでも実現できないこのような機能を提供するのが RTS なんでゲソ。別の言い方をすると、GHC にとって RTS は Java にとっての VM のようなものだ、ということもできるでゲソ。この RTS の全てを解説することは残念ながら大変荷が重いでゲソ。今回は実際に Haskell ソースコードをコンパイルしてみた結果を調査することで、「機械語化された Haskell 関数を RTS がどのように実行するのか」を解明することに焦点をしぼってみようと思うでゲッソ。

今回対象とする実行環境は Debian GNU/Linux amd64 sid 2012/07/01 時点、GHC のバージョンは 7.4.1 で、-threaded オプションは使わないでゲソ。[*2]

13.2 [表層] Haskell の世界

まず簡単な Haskell コードを書いてみるでゲソ。

```
main :: IO ()
main = putChar 'H'
```

うん、とっても簡単でゲソ。このコードはどうやって動くかわかるでゲソ？

1. main 関数から実行開始
2. putChar に'H' を適用
3. hPutChar に stdout と'H' を適用
4. (その他いろいろな Haskell 関数が呼び出される)
5. なんだかんだあって stdout のバッファに'H' を書き込む

Haskell 言語の上ではこれで動作を理解できたことになるでゲソ。しかし実行バイナリとしての動作を考えてみると色々な疑問がわくじゃなイカ。

```
### わからないことリスト ###
* [疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？
* [疑問 2] Haskell の関数はどうやって他の関数を呼び出すのか？
* [疑問 3] stdout バッファの flush は誰がやるのか？
```

13.3 潜水艦に乗り込もうじゃなイカ!

まず “[疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？ ” を解明したいでゲソ。ひょっとしてさっきの main.hs ソースコードをコンパイルしてできたオブジェクトファイルの中に C 言語の main 関数が紛れているんじゃなイカ？ 調べてみるでゲッソ!

```
$ ghc -c Main.hs
$ nm Main.o
0000000000000020 D Main_main_closure
0000000000000018 T Main_main_info
0000000000000010 d Main_main_srt
0000000000000050 D ZCMain_main_closure
00000000000000b8 T ZCMain_main_info
```

[*2] RTS にはスレッド版と非スレッド版があるでゲソ。-threaded オプションを付けない場合には、コンパイラは非スレッド版 RTS を使うようになるでゲソ。

```
0000000000000040 d ZCMain_main_srt
0000000000000000 D __stginit_Main
0000000000000000 D __stginit_ZCMain
                 U base_GHCziTopHandler_runMainIO_closure
                 U base_SystemziIO_putChar_closure
                 U ghczmprim_GHCziTypes_Czh_static_info
                 U newCAF
0000000000000000 d sfC_closure
                 U stg_CAF_BLACKHOLE_info
                 U stg_ap_p_fast
                 U stg_bh_upd_frame_info
```

Main.o の中には C 言語の main 関数はなかったでゲソ。どうやって実行されるのかさっぱりじゃなイカ……。それはそうといくつか見えるシンボルは何を表わすのでゲソ？ Haskell 関連のシンボルは z-encoding [*3] という規則でエンコーディングされているでゲソ。それを翻訳するとイカのような意味になるでゲソ。

- Main_main_closure := Main.main closure
- ZCMain_main_closure := :Main.main closure
- base_GHCziTopHandler_runMainIO_closure := base GHC.TopHandler.runMainIO closure
- base_SystemziIO_putChar_closure := base System.IO.putChar closure
- ghczmprim_GHCziTypes_Czh_static_info := ghc-prim GHC.Types.C# static info

GHC.TopHandler, System.IO, GHC.Types, Main はモジュール名じゃなイカ。でも:Main なんてモジュールは定義した覚えがないでゲソ。また closure と info というのはなんでゲソ？ さらに以下のシンボルは RTS 組み込みのもののようでゲソ。

- newCAF
- stg_CAF_BLACKHOLE_info
- stg_ap_p_fast
- stg_bh_upd_frame_info

わからないことが増えたじゃなイカ……。

```
### わからないことリスト（更新）###
* [疑問 4] :Main モジュールとは何か？
* [疑問 5] closure, info とは何か？
* [疑問 6] newCAF とは何か？
* [疑問 7] stg_*とは何か？
```

13.4 [中深層] C 言語の世界

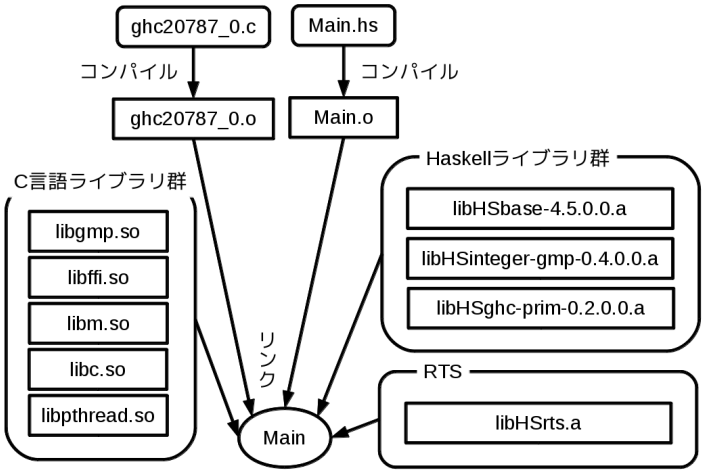
気分をかえて、ghc -v コマンドで Main.hs のコンパイル順序をよく見てみるでゲッソ。

```
$ ghc -v Main.hs
--snip--
*** C Compiler:
'/usr/bin/gcc' '-fno-stack-protector' '-Wl,--hash-size=31' '-Wl,--reduce-
memory-overheads' '-c' '/tmp/ghc20787_0/ghc20787_0.c' '-o' '/tmp/ghc20787
```

[*3] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/SymbolNames

```
_0/ghc20787_0.o' '-DTABLES_NEXT_TO_CODE' '-I/usr/lib/ghc/include'
*** Linker:
'/usr/bin/gcc' '-fno-stack-protector' '-Wl,--hash-size=31' '-Wl,--reduce-
memory-overheads' '-o' 'Main' 'Main.o' '-L/usr/lib/ghc/base-4.5.0.0' '-L/
usr/lib/ghc/integer-gmp-0.4.0.0' '-L/usr/lib/ghc/ghc-prim-0.2.0.0' '-L/us
r/lib/ghc' '/tmp/ghc20787_0/ghc20787_0.o' '-lHSbase-4.5.0.0' '-lHSinteger
```

ふむ。このログからコンパイルの流れ図を書いてみるとイカのようになるでゲソ。



今回作った Main.hs を動かすために、Haskell ライブラリ群と C 言語ライブラリ群の力を借りるんでゲソが、その寄せ集めでは実現できない部分を吸収してくれるのが RTS で、そのライブラリ名は libHSrts.a なんでゲソ! [*4]

ところで身に覚えのない/tmp/ghc20787_0/ghc20787_0.c などというソースコードがコンパイル対象になっているでゲソ。気になるので生成ファイルを残すように GHC にオプションを与えてみるでゲソ。

```
$ ghc -keep-tmp-files -tmpdir ./tmp Main.hs
```

すると tmp/ghc25150_0/ghc25150_0.c なんてファイルが残っていたでゲソ。

```
#include "Rts.h"
extern StgClosure ZCMain_main_closure;
int main(int argc, char *argv[])
{
    RtsConfig __conf = defaultRtsConfig;
    __conf.rts_opts_enabled = RtsOptsSafeOnly;
    return hs_main(argc, argv, &ZCMain_main_closure,__conf);
}
__asm__("\t.section .debug-ghc-link-info,\"\",@note\n\t.ascii ...
```

見つけたでゲソ! C 言語の main 関数じゃなイカ! ということは、GHC が作った実行バイナリは hs_main 関数から始まるということでゲソ。hs_main というのは何者なのでゲソ？ この hs_main から Haskell の世界を

[*4] スレッド版 RTS のライブラリ名は libHSrts_thr.a でゲソ

繋ぐのが RTS なのでゲソ。RTS の本体はこの hs_main 関数から呼び出される real_main 関数で、この関数を見れば Haskell プログラムの一生が一目瞭然でゲソ!

```
static void real_main(void)
{
    int exit_status;
    SchedulerStatus status;

    hs_init_ghc(&progargc, &progargv, rtsconfig);
    {
        Capability *cap = rts_lock();
        rts_evalLazyIO(&cap,progmain_closure, NULL);
        status = rts_getSchedStatus(cap);
        taskTimeStamp(myTask());
        rts_unlock(cap);
    }
    switch (status) {
/* --snip-- */
    }
    shutdownHaskellAndExit(exit_status);
}
```

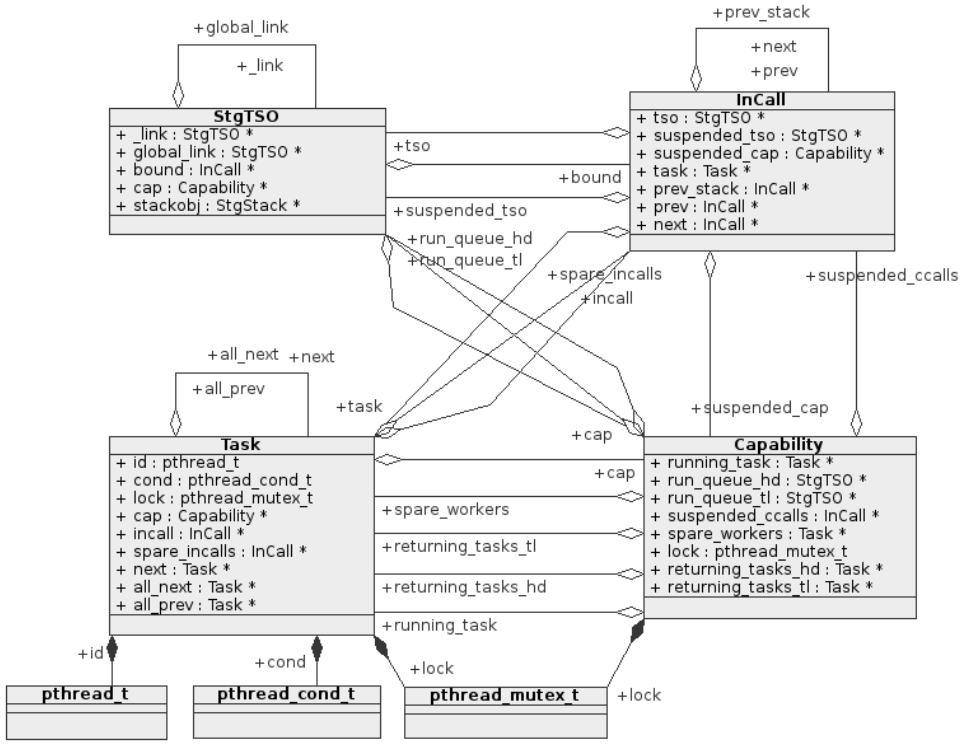
real_main 関数の中身を簡単に解説すると、

1. hs_init_ghc := RTS の初期化
2. rts_lock := Task 生成、Capability の取得
3. rts_evalLazyIO := StgTSO を生成して run queue に繋げ、スケジュール実行
4. rts_unlock := Task 解放
5. shutdownHaskellAndExit := 後処理

のようになっているでゲソ。キモは rts_evalLazyIO 関数あたりだということがわかると思うでゲソ!
ところで Capability とかなんのことでゲソ？

- Capability := RTS における仮想 CPU。非スレッド版 RTS では一つだけ作られるでゲソ
- Task := 実際の実行を司る何か。非スレッド版 RTS なので、最初は一つだけ作られるでゲソ
- StgTSO := Haskell のスレッドの実体。Haskell のスタックはこの構造体が保持しているでゲソ

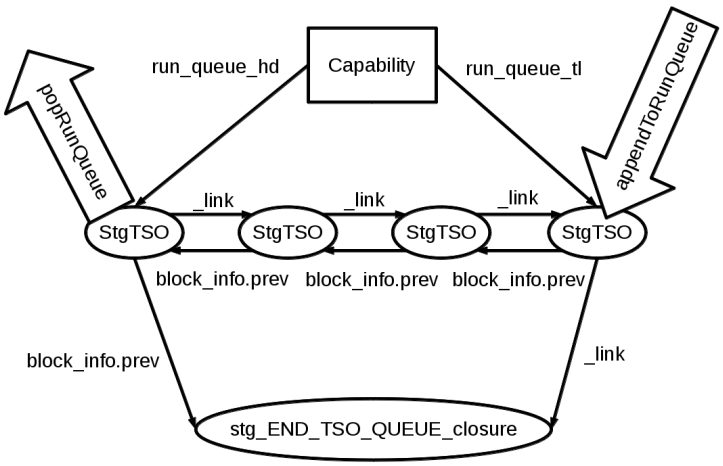
これだけだと StgTSO だけあれば十分じゃなイカと思うでゲソが、threaded な RTS の場合にはイカの図のように Task が pthread と紐付くようになるでゲソ。今回は pthread を使わない RTS なので、簡素な実装で済んでいるんでゲソ。



13.4.1 rts_evalLazyIO 関数でプログラム本体を実行

```
hs_main => real_main => rts_evalLazyIO => scheduleWaitThread => schedule
```

のような順序で呼び出される schedule という関数の中で Haskell のスレッドをスケジュール実行するでゲソ。当然、スケジュール実行するからには run queue があり、それは Capability に紐づいた StgTSO の queue でゲソ。



スケジュール実行される StgTSO は何をするかというと、あらかじめ StgTSO のスタックにやるべきことが積まれてから実行されるんでゲソ。

```
/* rts/RtsAPI.c */
INLINE_HEADER void pushClosure   (StgTSO *tso, StgWord c) {
  tso->stackobj->sp--;
  tso->stackobj->sp[0] = (W_) c;
}
/* --snip-- */
StgTSO *
createIOThread (Capability *cap, nat stack_size,  StgClosure *closure)
{
  StgTSO *t;
  t = createThread (cap, stack_size);
  pushClosure(t, (W_)&stg_ap_v_info);
  pushClosure(t, (W_)closure); /* = ZCMain_main_closure */
  pushClosure(t, (W_)&stg_enter_info);
  return t;
}
```

stg_*と ZCMain_main_closure というシンボルを StgTSO のスタックに積んでいるでゲソ。ZCMain_main_closure とはなんだったかというと Main.o の中に含まれていたことを覚えていると思うでゲソ。Main.o のシンボルが StgTSO のスケジュール実行によって呼び出されそうな気がしてきたでゲソ!

```
### わからないことリスト（更新）###
* [疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？
  => 手掛かり: StgTSO を schedule 実行する中で呼び出されそう
```

13.4.2 shutdownHaskellAndExit 関数で後始末

```
hs_main => real_main => shutdownHaskellAndExit => hs_exit_ => flushStdHandles
```

と呼び出される flushStdHandles 関数の中で再び Haskell コードがスケジュール実行されるんでゲソ。もう全部終わったかと思ったじゃなイカ。今度はどんなことを StgTSO にさせるかというと……

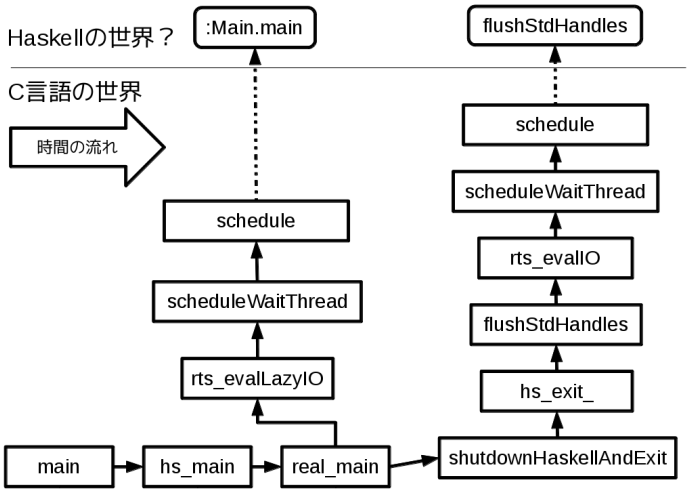
```
/* rts/RtsAPI.c */
StgTSO *
createStrictIOThread(Capability *cap, nat stack_size,  StgClosure *closure)
{
  StgTSO *t;
  t = createThread(cap, stack_size);
  pushClosure(t, (W_)&stg_forceIO_info);
  pushClosure(t, (W_)&stg_ap_v_info);
  pushClosure(t, (W_)closure); /* = flushStdHandles_closure */
  pushClosure(t, (W_)&stg_enter_info);
  return t;
}
```

closure については後の章で解説するでゲソが、Haskell の flushStdHandles 関数が stdout と stderr のバッ

ファを flush する関数でゲソ。Haskell の main 関数の中で stdout の flush をしなくても、RTS 終了直前に flush してくれるようでゲソ。気軽に putStr して怠惰に実行終了しても、ちゃんと RTS が flush してくれるので便利でゲッソ!

```
### わからないことリスト（更新）###
* [疑問 3] stdout バッファの flush は誰がやるのか？
  => 解決: RTS 終了直前に flushStdHandles 関数が呼び出される
```

C 言語で書かれた部分を調べることで判明したことを図にまとめてみたでゲッソ。



13.5 [漸深層] 幽霊船

……とちょっと待ってほしいでゲソ。さっきから ZCMain_main_closure なるものが出てきているんだが何者なんでゲソ？ z-encoding からひもとくと、:Main.main closure になるようでゲソ。でも:Main.main なんて関数を作った覚えがないでゲソ。その謎を解くために、GHC の-ddump-ds オプションで Main.hs をコンパイラパイプラインの Desugar まで通してみるでゲソ。

```
$ ghc -c Main.hs -ddump-ds
==================== Desugar (after optimization) ====================
Result size = 8

Main.main :: GHC.Types.IO ()
[LclIdX]
Main.main = System.IO.putChar (GHC.Types.C# 'H')

:Main.main :: GHC.Types.IO ()
[LclIdX]
:Main.main = GHC.TopHandler.runMainIO @ () Main.main
```

:Main.main 関数がコンパイル時に自動生成されているでゲソ!

```
### わからないことリスト（更新）###
* [疑問 4] :Main モジュールとは何か？
```

=> **解決**: **コンパイル時に** `Main.main` **関数を** `runMainIO` **でラップするために自動生成される**

13.5.1 main 実行前にシグナルハンドラ設定 (GHC.TopHandler.runMainIO)

じゃあ GHC.TopHandler.runMainIO というのは何者なんでゲソ？

```
runMainIO :: IO a -> IO a
runMainIO main =
    do
      main_thread_id <- myThreadId
      weak_tid <- mkWeakThreadId main_thread_id
      install_interrupt_handler $ do
           m <- deRefWeak weak_tid
           case m of
               Nothing  -> return ()
               Just tid -> throwTo tid (toException UserInterrupt)
      main -- hs_exit() will flush
    `catch`
      topHandler
--snip--
install_interrupt_handler :: IO () -> IO ()
install_interrupt_handler handler = do
   let sig = CONST_SIGINT :: CInt
   _ <- setHandler sig (Just (const handler, toDyn handler))
   _ <- stg_sig_install sig STG_SIG_RST nullPtr
   return ()
--snip--
topHandler :: SomeException -> IO a
topHandler err = catch (real_handler safeExit err) topHandler
```

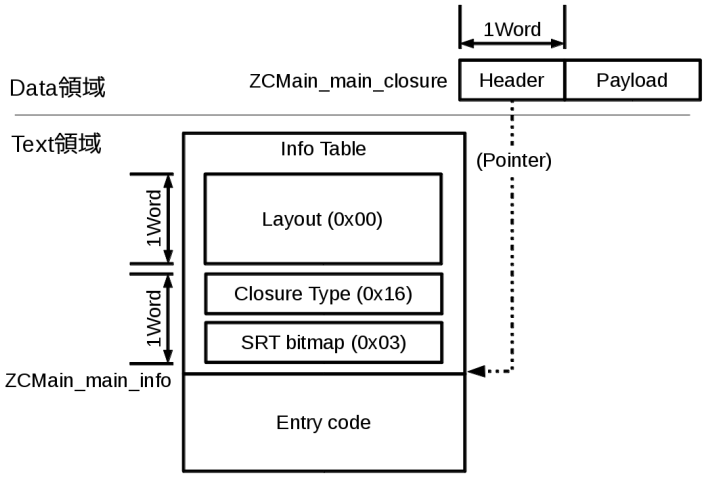
main 関数の実行前にシグナルハンドラの設定をしているでゲソ。すなわち setHandler で SIGINT が入ったら Haskell の main 関数を実行しているスレッドに UserInterrupt 例外が入るようにし、sigaction を SA_RESETHAND に、つまり一回 SIGINT が入ったらシグナルハンドラの設定をデフォルトに戻すように設定するでゲソ。そういえば GHC が吐いたバイナリは C-c を二回入力しないと終了できなかった気がするでゲソ。こんなところで日頃の疑問が解けたじゃなイカ。また、main 関数が例外終了したら real_handler を呼び出して後始末をするでゲソ。

わからないことリスト（更新）###
* [**疑問 1**] **そもそも** `main` **関数がどこからどうやって呼び出されるのか？**
=> **解決**: `StgTSO` **が実行開始すると**:`Main.main` **から** `runMainIO` **経由で呼び出される**

13.6 Haskell の関数はどんなメモリ配置？

ちょっとコードの解析を休んで、これでまで説明を避けてきた末尾に closure や info が付いたシンボルについて解説するでゲソ。この図はコンパイル済み:Main.main 関数のメモリレイアウトでゲソ。[*5]

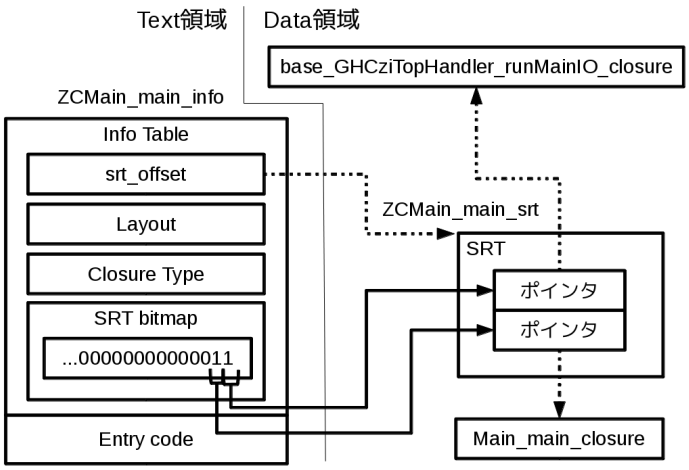
[*5] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects



Haskell の関数は HOGE_closure と HOGE_info という二つのシンボルにコンパイルされるでゲソ。HOGE_closure の方が親玉で、先頭 1 ワードには HOGE_info へのポインタが入っているでゲソ。HOGE_info の高位のアドレス方向には機械語のコードが格納されていて、そして低位のアドレス方向には Info Table という closure の形式を見わけるためのヒントが格納されているでゲソ。一つのポインタで二度おいしいとはすごいしくみじゃなイカ!

Closure Type には closure の形式が入っているでゲソ。:Main.main の場合 0x16 が入っているので、includes/rts/storage/ClosureTypes.h ヘッダの定義によると THUNK_STATIC ということになるでゲソ。THUNK というのはみんな知っている通り遅延されて未評価の式でゲソ。STATIC というのは動的に確保されない thunk という意味でゲソ。:Main.main はトップレベルで引数がないので thunk は 1 つしか作られないはずでゲソ。Layout は Payload がどのような形式になっているのか表わしているでゲソ。

今度は SRT bitmap に注目して図を描いてみたでゲソ。



SRT bitmap はその Info Table から辿れる SRT の中のどのワードまで有意なポインタが入っているのか表わしているでゲソ。GC する際に SRT 内のポインタを素早く辿るために使う典型的な手法でゲッソ。SRT(Static Reference Table) というのは Entry code 内で使用している closure へのポインタのリストのようでゲソ。

少し話がそれるでゲソが GHC のソースコードやドキュメントでは bitmap という単語は 2 つの意味で使われるでゲソ。 1 つはさっき説明した SRT bitmap でゲソ。もう 1 つは Payload が Bitmap layout という形式であった場合に、その Payload のサイズとどの部分がポインタかを表わす bitmap でゲソ。この Payload の

bitmap は SRT bitmap とは形式も異なり、上位のビット列に Payload のサイズが格納され、下位のビット列にポインタ/非ポインタかを表わすビットが格納されているでゲソ。混同しがちなので気をつけるでゲッソ!

```
### わからないことリスト（更新）###
* [疑問 5] closure, info とは何か？
  => 解決: Haskell の関数をコンパイルすると作られ、info に機械語コードが格納されている
```

13.7 [深海層] cmm 言語の世界

なんとなく解ったような気になったでゲソ。でも、C 言語の schedule 関数によるコンパイル済み Haskell 関数のスケジュール実行のイメージがさっぱりつかめないじゃなイカ？ 落ち着いて C 言語の schedule 関数から追い掛けてみるでゲッソ。

13.7.1 :Main.main 関数はどこでゲソ!?

```
/* schedule 関数 (rts/Schedule.c) */
static Capability *
schedule (Capability *initialCapability, Task *task)
{
/* --snip-- */
  while (1) {
/* --snip-- */
    switch (prev_what_next) {
/* --snip-- */
    case ThreadRunGHC:
    {
        StgRegTable *r;
        r = StgRun((StgFunPtr) stg_returnToStackTop, &cap->r);
        cap = regTableToCapability(r);
        ret = r->rRet;
        break;
    }
```

schedule 関数は StgTSO を実行可能であれば、StgRun 関数を呼び出すでゲソ。

```
/* StgRun 関数 (rts/StgCRun.c) */
#define STG_RUN "StgRun"
/* --snip-- */
static void GNUC3_ATTRIBUTE(used)
StgRunIsImplementedInAssembler(void)
{
    __asm__ volatile (
        /*
         * save callee-saves registers on behalf of the STG code.
         */
        ".globl " STG_RUN "\n"
        STG_RUN ":\n\t"
        "subq %0, %%rsp\n\t"
```

```
        "movq %%rsp, %%rax\n\t"
        "addq %0-48, %%rax\n\t"
        "movq %%rbx,0(%%rax)\n\t"
        "movq %%rbp,8(%%rax)\n\t"
        "movq %%r12,16(%%rax)\n\t"
        "movq %%r13,24(%%rax)\n\t"
        "movq %%r14,32(%%rax)\n\t"
        "movq %%r15,40(%%rax)\n\t"
        /*
         * Set BaseReg
         */
        "movq %%rsi,%%r13\n\t"
        /*
         * grab the function argument from the stack, and jump to it.
         */
        "movq %%rdi,%%rax\n\t"
        "jmp *%%rax\n\t"
```

StgRun 関数はアセンブラで書かれていて、しかも長いでゲソが、やってることは簡単でゲソ。 x86-64 の ABI として保存すべきレジスタ[*6] を C 言語のスタックに保存し、Capability の r メンバー (StgRegTable 型) へのポインタを r13 レジスタに入れてから stg_returnToStackTop にジャンプするだけでゲソ。

この StgRegTable には仮想 CPU で用いるレジスタが入っているでゲソ。ただし全ての仮想 CPU レジスタがメモリに置かれている訳ではなくて、割り当て可能なものはなるべく物理 CPU のレジスタを使って速度をかせぐでゲソ。[*7] includes/stg/MachRegs.h ファイルの x86_64_REGS を見れば x86-64 での割り当てがわかるでゲッソ。cmm 言語をコンパイルしたバイナリを逆アセンブルする際には注意でゲソ。

ジャンプ先の stg_returnToStackTop のソースコードを見てみると……

```
/* stg_returnToStackTop 関数 (rts/StgStartup.cmm) */
#define CHECK_SENSIBLE_REGS() \
    ASSERT(Hp != 0);            \
    ASSERT(Sp != 0);            \
    ASSERT(SpLim != 0);         \
    ASSERT(SpLim - WDS(RESERVED_STACK_WORDS) <= Sp);
/* --snip-- */
stg_returnToStackTop
{
  LOAD_THREAD_STATE();
  CHECK_SENSIBLE_REGS();
  jump %ENTRY_CODE(Sp(0));
}
```

これが cmm 言語 [*8] でゲソ。 cmm 言語とは C 言語モドキの見た目を持っている高級アセンブラで、限定用途の switch 文は使えるでゲソが、ループ構文はないでゲソ。cmm 言語の中では実際の CPU レジスタではなく、さっきでてきた仮想 CPU レジスタを使ってプログラミングするでゲソ。スタックポインタも仮想 CPU のものなので、cmm 言語で使われるスタック領域は C 言語とは別の領域になるでゲソ。

[*6] http://www.x86-64.org/documentation/abi.pdf
[*7] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution/Registers
[*8] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/CmmType

compiler/cmm/CmmParse.y が cmm 言語のパーサなんでゲソが、基本文法とは別に専用の命令も定義されていたりするでゲソ。さらに Haskell の関数は長いコンパイルパイプラインを通って一旦 cmm 言語に変換された後に機械語になるでゲソ。GHC をささえる縁の下の力持ちでゲソね!

ところで stg_returnToStackTop 関数の 3 行が全く読めないでゲソ……。LOAD_THREAD_STATE と ENTRY_CODE は compiler/cmm/CmmParse.y によって解釈される拡張文法らしいでゲソ。なんとか compiler/codeGen/CgForeignCall.hs を読みながらカンで読んでみるでゲソ。

0. BaseReg(x86-64 だと r13 レジスタ) から StgRegTable へのポインタを得る
1. StgRegTable の rCurrentTSO メンバーを tso 変数に
2. tso の先にある StgTSO の stackobj メンバーを stack 変数に
3. StgRegTable の Sp メンバーを stack->sp で初期化
4. StgRegTable の SpLim メンバーを stack->stack + RESERVED_STACK_WORDS で初期化
5. StgRegTable の rHpAlloc メンバーを 0 で初期化
6. StgRegTable の Hp メンバーを初期化
7. StgRegTable の HpLim メンバーを初期化
8. スタックとヒープが正常かチェック
9. StgRegTable の Sp メンバーの先頭にジャンプ

つまり BaseReg の中にあるスタックとヒープの情報を初期化して StgTSO のスタック先頭にジャンプするんでゲソ。スタック先頭はなんだったかというと stg_enter_info だったじゃなイカ？ createIOThread 関数で積んでいたハズでゲソ。

```
/* includes/Cmm.h */
#define Sp(n)  W_[Sp + WDS(n)]
#define Sp_adj(n) Sp = Sp + WDS(n)
#define TAG_BITS                3
#define TAG_MASK ((1 << TAG_BITS) - 1)
#define GETTAG(p) (p & TAG_MASK)
#define LOAD_INFO                                   \
  if (GETTAG(P1) != 0) {                            \
      jump %ENTRY_CODE(Sp(0));                      \
  }                                                 \
  info = %INFO_PTR(P1);
#define ENTER()                                       \
 again:                                               \
  W_ info;                                            \
  LOAD_INFO                                           \
  switch [INVALID_OBJECT .. N_CLOSURE_TYPES]          \
         (TO_W_( %INFO_TYPE(%STD_INFO(info)) )) { \
/* snip */                                            \
  default:                                            \
   {                                                  \
      UNTAG_R1                                        \
      jump %ENTRY_CODE(info);                         \
/* rts/HeapStackCheck.cmm */
INFO_TABLE_RET( stg_enter, RET_SMALL, P_ unused)
{
    R1 = Sp(1);
    Sp_adj(2);
```

```
    ENTER();
}
```

stg_enter_info でやっていることを気合いで読むでゲソ!

0. StgTSO のスタック 2 ワード目を R1 レジスタに取り出す (= closure)
1. StgTSO のスタックポインタを 2 ワード戻す
2. R1 レジスタのタグビットが立っていたらスタックポインタ先頭の差すアドレスにジャンプ
3. R1 レジスタから Info Table のアドレスを得る (= info)
4. Info Table の Closure Type によって処理を切り換え
5. Closure Type が特殊なものでなかった場合 Entry code へジャンプ

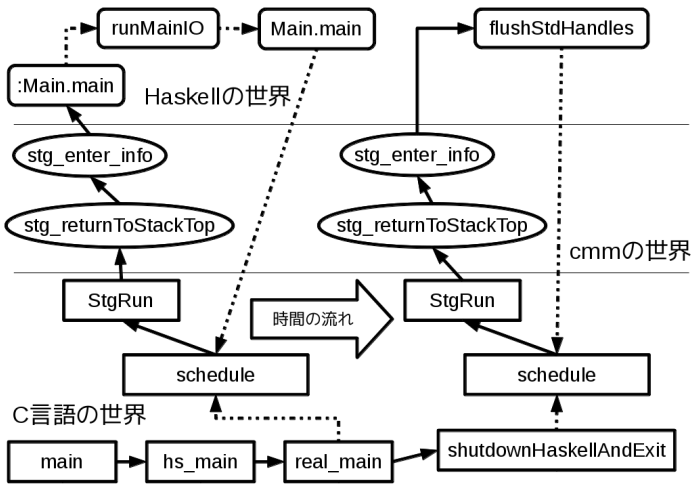
今回のケースではスタックの二番目に積まれていたのは ZCMain_main_closure なので、ZCMain_main_closure の先頭アドレス (Header) から ZCMain_main_info(Info Table) を辿り、 Closure Type が THUNK_STATIC なので ZCMain_main_info にそのままジャンプするんじゃなイカ？

結局 stg_enter_info は「StgTSO の次のスタックに積まれた Entry code をいい感じに実行する」という機能を持っているみたいでゲソ。

ということで、やったでゲソ! 遂に Haskell で書かれた:Main.main 関数に到達したでゲソ!

```
### わからないことリスト（更新）###
* [疑問 2] Haskell の関数はどうやって他の関数を呼び出すのか？
  => 手掛かり: 最初の関数は stg_enter_info から呼び出される
* [疑問 6] stg_*とは何か？
  => 手掛かり: stg_enter_info:=スタックに積まれた Entry code を実行する
```

ちょっと疲れたので、一旦休憩をかねてここまでの軌跡を図にまとめるでゲッソ。



こうして眺めてみると RTS というか GHC が吐き出す実行バイナリは大きく 2 つの領域に分断されていることがわかるでゲソ。それは C 言語と cmm 言語でゲソ。上の図には Haskell で書かれた領域も出てくるでゲソが、Haskell も結局は cmm 言語に変換されるので同じ領域と考えても良いでゲソ。GHC が吐き出すバイナリが実行開始すると、まず C 言語で記述されたコードが走り出すでゲソ。この領域では CPU が決めた ABI で動いているでゲソ。例えば x86/x64 の場合にはスタックも ABI の通りなので、スタックトレースが取れるでゲソ。ところが、StgRun 関数を経由して cmm 言語で記述されたコードが走りはじめると、そのコードが自発的に C 言語コードを呼び出したり schedule 関数に戻らない限り、CPU が決めた ABI ではなく GHC が

自分自身のために作り出した仮想 CPU によって決めた ABI によって関数呼び出しを行なうでゲソ。もっとも小さなレベルでは、この関数呼び出しは stg_enter_info のような cmm 言語で書かれたアドレスへのジャンプの繰り返しに見えたでゲソ。このような ABI は x86/x64 の ABI ではないので、gdb などのデバッガを通してみてもスタックトレースは取れないでゲソ。参照透明な海の力はハードウェアが決めた法則さえ再定義するんでゲソ!

13.7.2 関数から関数へジャンプ! (:Main.main => runMainIO)

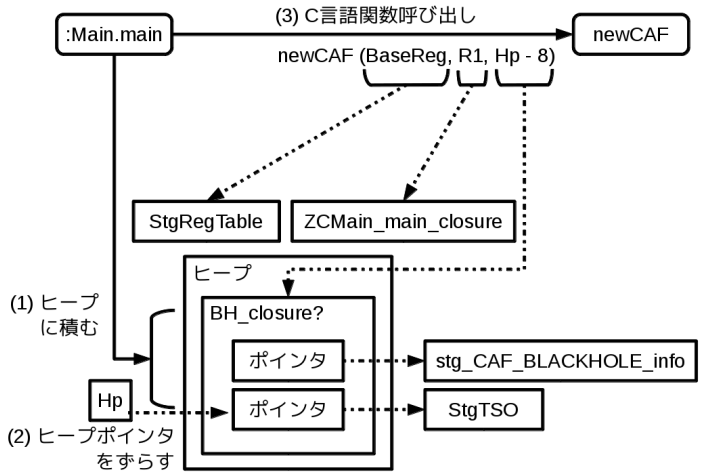
さっき辿り着いた:Main.main 関数の Entry code(ZCMain_main_info) からふたたび出発でゲソ! ZCMain_main_info の cmm ソースコードは ghc の-ddump-cmm オプションを使えば手に入るでゲソ。

```
ZCMain_main_info()
        { update_frame: <none>
          label: ZCMain_main_info
          rep:HeapRep static { Thunk }
        }
    cg2:
        if (Sp - 16 < SpLim) goto cg4;
        Hp = Hp + 16;
        if (Hp > HpLim) goto cg6;
        I64[Hp - 8] = stg_CAF_BLACKHOLE_info;
        I64[Hp + 0] = CurrentTSO;
        (_cg7::I64,) = foreign "ccall"
          newCAF((BaseReg, PtrHint), (R1, PtrHint), (Hp - 8, PtrHint));
        if (_cg7::I64 == 0) goto cg8;
        goto cg9;
    cg4: jump stg_gc_enter_1 ();
    cg6:
        HpAlloc = 16;
        goto cg4;
    cg8: jump I64[R1] ();
    cg9:
        I64[Sp - 16] = stg_bh_upd_frame_info;
        I64[Sp - 8] = Hp - 8;
        R1 = base_GHCziTopHandler_runMainIO_closure;
        R2 = Main_main_closure;
        Sp = Sp - 16;
        jump stg_ap_p_fast ();
}]
```

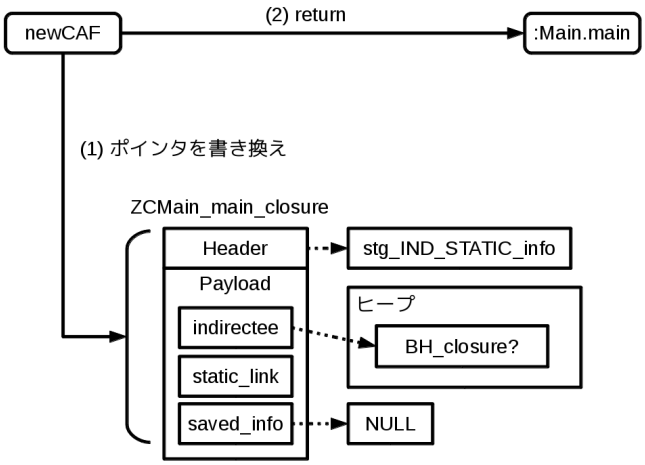
:Main.main 関数の Entry code 前半ではヒープにポインタを格納してから C 言語の newCAF 関数を呼び出すでゲソ。newCAF のコードを読むと判明するでゲソが、このヒープに置いたポインタは動的に作成した closure として使われるようでゲソ。ここではこの作成した closure は BH_closure と名前を付けて説明するでゲソ。

ヒープポインタ (Hp) はスタックポインタ (Sp) と違い、高位のアドレス方向に成長するんでゲソ。ここで、ヒープポインタ (Hp) とスタックポインタ (Sp) をずらす前にそれぞれ HpLim レジスタと SpLim レジスタと比較して、領域確保可能かどうか判別しているでゲソ。もしヒープやスタックあふれを引き起こすようであれば、stg_gc_enter_1 (本体は GC_GENERIC という define マクロ、rts/HeapStackCheck.cmm で定義) を経由して一旦 C 言語の schedule 関数に戻るでゲソ。schedule 関数は仮想 CPU の rRet レジスタから StgTSO 中断原因を調べて、領域を再割り当てしてくれたり GC を実行してくれたりするでゲソ。

もちろん今回の実行ではヒープもスタックもあふれないので、そのまま newCAF 関数を呼び出すでゲソ。



newCAF 関数 (rts/sm/Storage.c) に入ると、:Main.main 関数の closure を書き換え、:Main.main に return するでゲソ。



CAF(Constant Applicative Form) というのは「一度実行したら、その結果をメモ化して使いまわせるもの」のことでゲソ。例えば、無引数の関数はメモ化できるので、CAF になるでゲソ。CAF かどうかは単純に「無引数の関数としてコンパイルされる」かどうかで決まって、IO モナドかどうかには関係しないので、今回の:Main.main も CAF なんでゲソ。

「無引数の関数として定義する」ことではなく、「無引数の関数としてコンパイルされるかどうか」が重要になる点に注意するでゲソ。今回の話題の範囲外になるので深く説明はできないでゲソが、無引数の関数として定義してもコンパイル時にそうでなくなったり、:Main.main 以外にもコンパイル時に無引数の関数が生成されることなんかもあるでゲソ。[*9] なので、どれが CAF かを判断するには、コンパイラの生成するコード (Core 言語や STG 言語) などを見て確認する必要があるでゲソ。

newCAF 関数はこのメモ化に必要な準備をするでゲソ。newCAF 関数を読んだ段階ではまだ thunk を潰した結果が得られていないので、closure に BLACKHOLE という印を付けておくんでゲソ。そして closure に対応する thunk の評価が完了した段階になったら、今まで BLACKHOLE だった closure を評価済みの値で上書

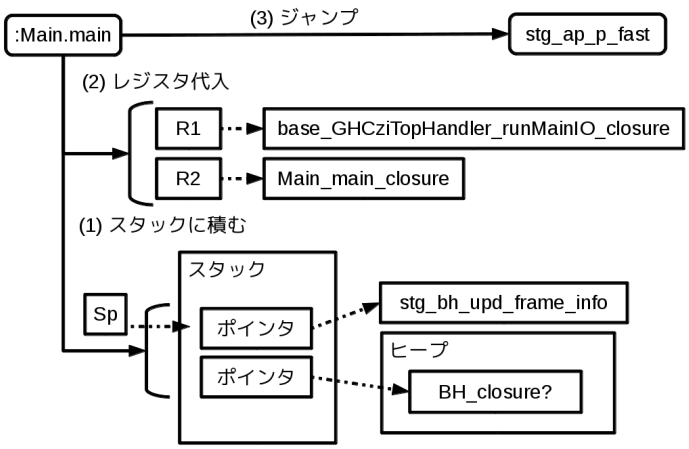
[*9] http://stackoverflow.com/questions/8330756/what-are-super-combinators-and-constant-applicative-forms

きするんでゲソ。[*10] threaded な RTS を使う場合には同時に別々の Task から thunk を触るケースが考えられるのでこの BLACKHOLE のしくみは大変重要でゲソ。

仮に thunk を潰している最中 (BLACKHOLE 状態) に別の Task から closure が呼び出された場合、stg_IND_STATIC_info(rts/StgMiscClosures.cmm) は stg_CAF_BLACKHOLE_info を呼び出し、messageBlackHole 関数 (rts/Messages.c) を使って BLACKHOLE 状態にある closure の評価待ち queue を作るようでゲソ。そしてブロックする準備ができたら ThreadBlocked を要因として schedule に戻るようでゲソ。すると schedule 関数は BLACKHOLE によってブロックされた StgTSO を再び run queue には繋げずに run queue から別の StgTSO を取り出してスケジュール実行を再開するでゲソ。ということはこの BLACKHOLE の closure に触ったことを要因とするブロックは誰か別の StgTSO が thunk を評価完了し終わったら、ブロック解除されるはずだと推測できるでゲソ。ここらへんはちょっと理解不足でゲソ。

```
### わからないことリスト（更新）###
* [疑問 6] newCAF とは何か？
  => 解決: newCAF 関数はメモ化に必要な器を closure に確保する
```

:Main.main 関数の Entry code 後半ではスタックに stg_*関数へのポインタを積み、R1 レジスタに呼び出し関数を R2 レジスタに引数へのポインタを代入して、stg_ap_p_fast へジャンプするでゲソ。



stg_ap_p_fast のソースコードは、GHC をソースコードからコンパイルするとできる inplace/bin/genapply というコマンドを実行することでソースを見れるでゲソ。さらにこの stg_ap_p_fast で使っている StgFunInfoExtraRev_arity マクロは inplace/bin/mkDerivedConstants というコマンドによって自動生成されるでゲソ。

この genapply コマンドは多量の cmm 関数を自動生成するでゲソ。さらっと見るかぎりでは Haskell 関数をコンパイルしてできる closure を動作させるのに必要な語集が stg_*関数のように見えるでゲソ。それらの語集の意味を知るにはもっと調査が必要そうでゲソね。

と、ここまで潜ってきて cmm コードを一つ一つ解釈するのに息切れしてきたでゲソ。参照透明で新鮮な空気を吸いた……いでゲソ。なんか楽をする方法はなイカ？ 手っ取り早く gdb で追ってみるでゲソ。[*11] gdb で実行する際には +RTS -V0 オプションを付けるといいでゲソ。Haskell スレッドのタイムスライスを実現するためのタイマーシグナルを無効にできるでゲソ。gdb で break している最中にタイマーシグナルが入ってしまうと、本来の動作を観測できなくなるでゲソ。

[*10] http://blog.ezyang.com/2011/05/reified-laziness/
[*11] http://hackage.haskell.org/trac/ghc/wiki/Debugging/CompiledCode

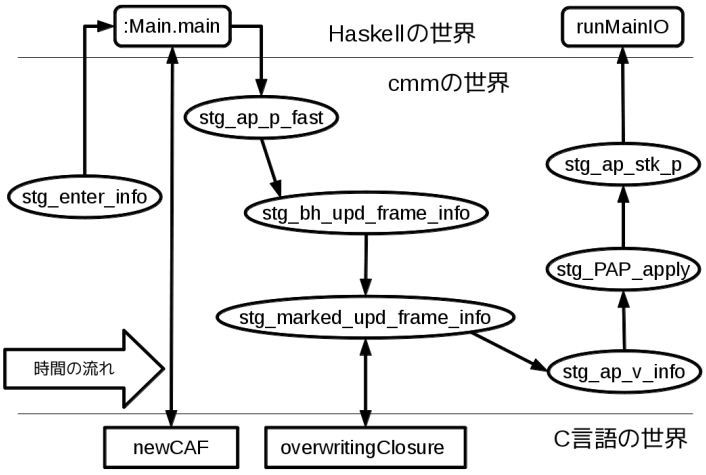
```
$ gdb Main
GNU gdb (GDB) 7.4.1-debian
--snip--
Reading symbols from /home/kiwamu/src/DiveIntoRTS/Main...done.
(gdb) b ZCMain_main_info
Breakpoint 1 at 0x404278
(gdb) run +RTS -V0
Starting program: /home/kiwamu/src/DiveIntoRTS/Main +RTS -V0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000000404278 in ZCMain_main_info ()
(gdb) si
0x000000000040427c in ZCMain_main_info ()
(gdb) si
0x000000000040427f in ZCMain_main_info ()
```

こんな感じにひたすらステップ実行していけばいつかは base_GHCziTopHandler_runMainIO_info に辿り着くはずでゲソ。頭使わなくて済んで楽勝じゃなイカ。

gdb で追いかけた関数の履歴

```
=> ZCMain_main_info
=> newCAF (C 言語関数)
=> stg_ap_p_fast
=> stg_bh_upd_frame_info
=> stg_marked_upd_frame_info
=> overwritingClosure (C 言語関数)
=> stg_ap_v_info
=> stg_PAP_apply
=> stg_ap_stk_p
=> base_GHCziTopHandler_runMainIO_info
```

やはり closure と closure の間は stg_*関数のジャンプによって繋いでくれるようでゲソ。この呼び出し関係を図にしてみるとイカのようなになるでゲソ。




```
### わからないことリスト（更新）###
* [疑問 7] stg_*とは何か？
  => 解決: Haskell 関数をコンパイルしてできる closure を動作させるのに必要な語集
```

この後どのように実行が進むかというと、コンパイルされた Haskell 関数 (closure) を stg_*などの RTS 側で準備された Entry code が間を繋げながら実行が進むんでゲソ。各 Entry code 間はジャンプ命令によって呼び出しが行なわれるのでゲソ。そのため、本来のスタックは使わず、StgTSO のスタックのみを使用することになるでゲソ。

13.8　浮上でゲッソ

cmm 言語の世界まで潜ったところで、今回の潜水を振り返ってみるでゲソ。RTS は C 言語と cmm 言語で実装されていたでゲソ。GHC の吐くコンパイル済みバイナリは C 言語の main から RTS の schedule 関数を呼び出すでゲソ。schedule 関数は Haskell のスレッドである StgTSO をスケジュール実行していたでゲソ。StgTSO の実行に入ると C 言語の ABI から cmm 言語の ABI に切り替わり、コンパイル済みの Haskell 関数である closure を実行していたでゲソ。

さて、疑問は全部解決したんでゲソ？

```
### わからないことリスト（最終結果）###
* [疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？
  => 解決: StgTSO が実行開始すると:Main.main から runMainIO 経由で呼び出される
* [疑問 2] Haskell の関数はどうやって他の関数を呼び出すのか？
  => 未解決!
* [疑問 3] stdout バッファの flush は誰がやるのか？
  => 解決: RTS 終了直前に flushStdHandles 関数が呼び出される
* [疑問 4] :Main モジュールとは何か？
  => 解決: コンパイル時に Main.main 関数を runMainIO でラップするために自動生成される
* [疑問 5] closure, info とは何か？
  => 解決: Haskell の関数をコンパイルすると作られ、info に機械語コードが格納されている
* [疑問 6] newCAF とは何か？
  => 解決: newCAF 関数はメモ化に必要な器を closure に確保する
* [疑問 7] stg_*とは何か？
```

=> **解決**: `Haskell` **関数をコンパイルしてできる** `closure` **を動作させるのに必要な語集**

ぬ？ 疑問 2 が解決してないじゃなイカ! この疑問 2 を解決するためには大量にある stg_*関数をどう組み合わせればどのような機能が実現できるのか、その大ざっぱな仕様について調査する必要があるでゲソ。かなり古い論文でゲソが、 “Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine” [*12] を読んで、設計思想のようなものを理解してから cmm 言語を読むと理解が速いかもしれないでゲソね。次回はもっと準備をしてから潜ろうと思うでゲッソ!

この夏、みんなも「わからないこと」を探しに RTS の海に潜ってみなイカ？

13.9 謝辞

今回の記事では筆者の調査結果が正しいのか自信が持てなかったので、サークル外の方々にレビューをお願いしたでゲソ。

@tyasunao さんには cmm 言語と STG 言語の基本的な読み方を教えてもらったでゲソ。@shelarcy さんには CAF について教えてもらったでゲソ。実は CAF の説明の文章は @shelarcy さんにそのまま書いてもらったでゲソ。

紙面を通じて感謝でゲソ! 本当にありがとうでゲソー!

13.10 参考文献

- GHC 7.4.1 のソースコード [*13]
- suztomo の日記「Haskell のコンパイル中に現れる STG と, GDB で見る C-backend な C–」 [*14]
- INSIDE 233「Tracing the compilation of Hello Factorial!」 [*15]
- GHC Developer Wiki「I know kung fu: learning STG by example」[*16]
- GHC Developer Wiki「Object Structure」 [*17]
- github: master-q/DiveIntoRTS (本記事での調査結果詳細) [*18]
- GHC ソースコードリーディング勉強会 #readghc のまとめ [*19]

[*12] `http://research.microsoft.com/apps/pubs/default.aspx?id=67083`
[*13] `https://github.com/ghc/ghc/tree/ghc-7.4.1-release`
[*14] `http://d.hatena.ne.jp/suztomo/20111224/1324718354`
[*15] `http://blog.ezyang.com/2011/04/tracing-the-compilation-of-hello-factorial/`
[*16] `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode`
[*17] `http://hackage.haskell.org/trac/ghc/wiki/StgObjectTypes`
[*18] `https://github.com/master-q/DiveIntoRTS`
[*19] `http://wiki.haskell.jp/Workshop/ReadGHC/`

第14章

評価

— @xhl_kogitsune

「お前たち! 今日も元気に簡約するでゲソ!!! タイトルを見て千反田とかいう奴が出てくることを期待していたのなら残念だったでゲソね! ここは私が侵略済みでゲソ!!

今日は式の**評価**の話をするでゲソ! 評価、あるいは**簡約**、というのはここでは計算の 1 ステップみたいなものでゲソ。たとえば式 `3 * (1 + 2)` を 1 回評価 (簡約) すると式 `3 * 3` になって、更に 1 回評価 (簡約) すると値 `9` が得られる、みたいな感じでゲソ。

一言に評価と言っても、やり方は色々あるでゲソ。式のどこから評価をしていくか、という**評価順序**も色々あるし、1 回の評価をどう実装するかも色々あるでゲソ。今日は、私オススメの評価順序である Normal Order 評価の話をしつつ、答えが出るまでにかかる評価回数を減らすにはどのように 1 回の評価を実装すればいいかという話をするでゲソ。

まず式 1 の評価を考えるでゲソ。ここでは Haskell 風に書いたでゲソが、`(\x -> x * x)` という「x を受け取って $x*x$ を返す関数」を $(1+2)$ に適用しているでゲソ。

式 1: Haskell

```
(\x -> x * x) (1 + 2)
```

OCaml だと

式 1: OCaml

```
(fun x -> x * x) (1 + 2)
```

C++ だと

式 1: C++

```
int f(int x){ return x * x; }
int main(){
...
int r = f(1 + 2);
...
}
```

こんな感じでゲソ (r の値を計算するでゲソ)。今日は一番最初の Haskell 風表記を使って説明するでゲソ。評価順序とかは実際の Haskell 実装とは違うことがあるので注意でゲソ! 式 1 に出てきた、関数、関数適用 、四則演算の書き方くらい把握しておけばだいたいサンプルコードは読めるんじゃなイカ?

さて、実は式の評価の順番 (評価順序) には色々あるでゲソ。この式を、どこから評価するでゲソか? C++ や OCaml なら、まず `1 + 2` を計算して `3` を出すところからはじめるんじゃなイカ? 然る後に関数 `(\x -> x * x)`[*1] を呼び出し、`x = 3` として `x * x` → `3 * 3` → `9` という風になるんじゃなイカ? これを式変形の過程として書くと図 14.1 のようになるでゲソ。

[*1] OCaml 版なら `(fun x -> x * x)`、C++ 版なら `f`

```
(\x -> x * x) (1 + 2)
→ (\x -> x * x) 3
→ 3 * 3
→ 9
```

図 14.1: 式 1: Applicative Order 評価、文字列ベース簡約

行頭の矢印 (→)1 個分が、評価 1 回分に対応するでゲソ。それぞれの評価ステップでは、下線を引いた部分を評価しているでゲソ:

1. まず最初に、(1 + 2) を評価して 3 に置き換えるでゲソ。
2. 次に、(\x -> x * x) の 3 への適用を評価するでゲソ。具体的には、関数の中身 x * x の中の仮引数 x を実引数 3 に置き換えて、3 * 3 を作るでゲソ。
3. 後は 3 * 3 を評価して完了じゃなイカ!

これはよく使われる評価順序の一つで、関数の引数を評価してから関数適用を評価する、**Applicative Order (作用的順序)** と呼ばれるものでゲソ。今回は 3 ステップで評価できたでゲソ。Call By Value の時に使われる評価順序はこんな感じでゲソ。手続き型言語の大半はこんな感じでゲソ。

しかし、世の中 Applicative Order ばかりじゃないでゲソ。これ以外に方法なんてあるでゲソ? と思う人もいるかもしれないでゲソ。でも、引数の評価を後回しにして関数適用を先に評価する、**Normal Order (正規順序)** という評価順序もあるでゲソ。Haskell なんかはそんな感じの順序でゲソ。式 1 を Normal Order で評価すると図 14.2 のようになるでゲソ。

```
(\x -> x * x) (1 + 2)
→ (1 + 2) * (1 + 2)
→ 3 * (1 + 2)
→ 3 * 3
→ 9
```

図 14.2: 式 1: Normal Order 評価、文字列ベース簡約

1. 最初に、(\x -> x * x) の (1 + 2) への適用を評価するでゲソ。(1 + 2) はそのままにして、関数の中身 x * x の中の仮引数 x を実引数 (1 + 2) に置き換えて、(1 + 2) * (1 + 2) を作るでゲソ。慣れていない人には、Applicative Order の時は値 (3) だったところに式 ((1 + 2)) が無理やり入っているように見えるかもしれないでゲソが、入れてしまえば何ともないでゲソ。
2. それから、(1 + 2) の評価をして (左側)、
3. それから、(1 + 2) の評価をして (右側)、
4. 最後に 3 * 3 の評価をして、最終的には同じ答え 9 を得るでゲソ。

これが私の好きな Normal Order でゲソ! **遅延評価 (Lazy Evaluation)** の時に使われる評価順序はこんな感じでゲソ。引数の評価を遅延させてとりあえず関数適用の評価をしてしまうでゲソ。Normal Order には、後で説明するでゲソが、「とりあえず Normal Order を使っておけば間違いは起こらない」という安心感があるでゲソ。今日の話の前半ではそんな Normal Order の宣伝をするでゲソ!

しかし、今回は 4 ステップかかったでゲソ。Applicative Order の時と比べて 1 ステップ増えてしまったでゲソ…。でも、大丈夫でゲソ! 後半では評価の回数を減らす方法を紹介するでゲソ!

14.1 準備しなイカ?

ここで、下準備として基本的な用語とかを説明しておくでゲソ。今日の話は、λ 計算における「λ 式から、正規形 に至るまでの 評価 (β 簡約) の回数」についての話に基づいているでゲソ。何のことか分からなくても心配ないでゲソ。おおざっぱには「上に出したような Haskell 風の式 から、結果の値 (整数) が出るまでの 評価 (簡約) の回数」くらいに考えておけばいいでゲソ。この二つには違う面もあるでゲソが、今日はそこには踏み込まないので気にしなくていいでゲソ[*2]。

14.1.1 式

まず、評価をしたい**式**とは何かをはっきりさせようじゃなイカ! 今日使う Haskell 風の式では「変数」「関数 (λ 抽象)」「関数適用」「四則演算」「整数定数」の 5 種類のことが書けるでゲソ[*3]。λ 計算では、このうち「変数」「関数 (λ 抽象)」「関数適用」の 3 種類だけから構成される λ **式**を扱うでゲソ。構文や意味とかをまとめるとこんな感じになるでゲソ。カッコは適宜省略するでゲソ。λ 式は Haskell 式と表記がちょっと違うだけで怖くないでゲソよ♪

表 14.1: 式

種類	λ 式	Haskell 風の式	意味
変数	x	`x`	変数 x
関数/λ 抽象	$(\lambda x.A)$	`(\x -> A)`	x を受け取って A を返す関数
関数適用	(AB)	`(A B)`	関数 A を引数 B に適用する
四則演算	なし	`+` など	
数値定数	なし	`1` など	

この A とか B の所には実際には λ 式や Haskell 風の式が入るでゲソ。たとえば、Haskell 風の式での `(\x -> x * x)` は、「引数 `x` を受け取ってその 2 乗 `(x * x)` を返す関数」でゲソ。この場合、`A` は `(x * x)` に相当するでゲソ。これを式 `(1 + 2)` に適用すると、`((\x -> x * x) (1 + 2))` になるでゲソ。この式の値は $(1+2)*(1+2)=9$ になるはずでゲソ。これで簡単な整数演算は書き放題じゃなイカ?[*4] λ 式でも同様でゲソ[*5]。

Haskell「風」と書いたのは、Haskell だと型エラーになる式が出てきたり、実際の評価順が Haskell と違う場合があるからでゲソ。注意するでゲソ。

14.1.2 評価の 1 ステップと redex

今日は後半で「評価の回数」を減らす話をするでゲソ。で、「評価 1 回」って何でゲソ? 「計算の 1 ステップ」みたいなもの、と言ったでゲソが、ここでもう少しきちんと定義しておこうじゃなイカ!

λ 計算の場合は、評価は β **簡約**という、関数適用の計算 1 回分に相当する操作でゲソ。これは、

$$((\lambda x.A)B)$$

という形をした λ 式 (これを簡約可能式 (reducible expression)、略して **redex** と呼ぶでゲソ) を、「A の中に

[*2] たとえば、λ 計算では全て先行評価でも遅延評価でもできるでゲソが、Haskell とかの式の四則演算とかは普通、被演算数の値を先に評価する必要があるので先行評価が必要でゲソ。こういう話は [SPJ 1987] あたりに書いてあるでゲソ。

[*3] 純粋な λ 式だと 1+2 とかすら書けなくて読みにくくなるので、四則演算とかを足して読みやすい例を作るでゲソ

[*4] ループとか再帰は今回は扱わないでゲソ。Y コンビネータを λ 式で表現して再帰を書く分には今回の話同様に評価すればいいでゲソが、それだと評価回数が多くなるでゲソ。明示的に Y コンビネータをプリミティブとして定義する場合とかは特別に考慮が必要でゲソ。これらの話については詳しくは [SPJ 1987] あたりを読めばいいんじゃなイカ?

[*5] λ 計算では四則演算や数値定数が無いのにどうやって計算するかって? チャーチ数とかを使えば変数・λ 抽象・関数適用の 3 種類だけで整数 (のようなもの) と整数演算は表現できるでゲソ (しかしこのことと今日の話はあまり関係ないでゲソ)。詳しくはチャーチ数で検索するか「簡約! λ カ娘」(1 冊目) を参照でゲソ。

ある x を B で置き換えたもの」(これを $A[B/x]$ と表記するでゲソ) に置き換える操作でゲソ。関数の中の仮引数を実引数に置換する感じじゃなイカ! たとえば $((\lambda x.x)y)$ の場合、$A=x, B=y$ で、これを β 簡約すると「x の中にある x を y で置き換えたもの」つまり「y」になるでゲソ。

Haskell 風の式の場合には、評価は β 簡約の他に「四則演算の計算」も含むでゲソ。

```
n + m
```

のような式 (n, m は整数) を「四則演算を計算した結果」に置き換えるステップを 1 回の評価とみなすでゲソ。今日はこのような式も redex と呼ぶことにするでゲソ。式 1 で、

```
(\x -> x * x) (1 + 2)
```

の (1 + 2) という redex を選んで評価すると、

```
(\x -> x * x) 3
```

になるでゲソ。一方、(\x -> x * x) (1 + 2) という redex の評価を先にすると、x * x の中の x を (1 + 2) で置き換えて

```
(1 + 2) * (1 + 2)
```

になるでゲソ。このように、一つの式の中に redex が複数ある場合があるでゲソ。そのどれを選んで評価するかで評価順序が変わってくるでゲソ。

14.1.3 計算の終了 - 正規形 (Normal Form)

評価ステップを繰り返していって、redex がなくなるまで (それ以上できない状態の式になるまで) 到達したら、そこで計算終了 (停止) でゲソ。この redex がない式のことを **正規形 (Normal Form)** というでゲソ。

ただし、正規形に辿り着かず、無限に評価が終わらない場合もあるでゲソ。

14.2 評価順序を変えなイカ?

さて、一通り下準備が終わったところで最初に話した評価順序の話をもう一度しておくでゲソ。

すいません、遅れました!

なっ……こんなに早くここにたどり着くとは…あぶらあげの罠はどうしたのでゲソ?

? 何のお話ですか?

…なんでもないでゲソ。しかし今更来てももはや侵略済みでゲソ! まぁいいでゲソ、話を続けるでゲソよ! お前も Normal Order で侵略してやるでゲソ!

14.2.1 Applicative Order (作用的順序)

最初にちょっと出てきた **Applicative Order (作用的順序)** は、関数の引数をまず評価してから、関数適用を評価するでゲソ。**最右最内戦略**、つまり「一番右側の、一番内側の redex を選んで評価する」という言い方もするでゲソ[*6]。最初の方で出した図 14.1 では、式 1: (\x -> x * x) (1 + 2) を Applicative Order で評価しているでゲソが、(1 + 2) を 3 に評価してから関数適用を評価しているでゲソ。下線を引いた部分が各ステップで評価した redex を表しているでゲソ。

[*6] 構文木上での preorder 順で一番最後の redex という言い方もできるんじゃなイカ?

関数型、手続き型問わず広く見られる評価順序ですね。普及度からすればこちらの評価順序に「正規」という名前を与えても良かったのではないでしょうか?

そうはいかないでゲソ。他の評価順序では停止するのに Applicative Order で評価すると停止しないことがある、という欠点があるでゲソ。そんな評価順序には「正規」という名前はやれないでゲソ。たとえば式 2 のような場合でゲソ[*7]。

式 2: 評価順序によって停止したり停止しなかったりする例

```
(\y -> 1) ((\x -> x x) (\x -> x x))
```

これを Applicative Order で評価すると、図 14.3 のように無限ループするでゲソ。

```
(\y -> 1) ((\x -> x x) (\x -> x x))
→ (\y -> 1) ((\x -> x x) (\x -> x x))
→ (\y -> 1) ((\x -> x x) (\x -> x x))
→ ...
```

図 14.3: 式 2: Applicative Order では停止しない

この式の中の `(\x -> x x) (\x -> x x)` (λ 式だと $(\lambda x.xx)(\lambda x.xx)$) は、計算が停止しない有名な式のひとつでゲソ。Applicative Order で評価するとこの部分の評価を繰り返して無限ループになるでゲソ。

14.2.2 Normal Order (正規順序)

確かに、停止しないのは困りますね。何かいい評価順序はないものでしょうか…

そこで **Normal Order (正規順序)** でゲソ。これは、**最左最外 (leftmost outermost) 戦略** とも呼ばれるもので、その名の通り「最も外側にある redex のうち最も左側にあるものを評価する」評価戦略でゲソ[*8]。式 1 の Normal Order での評価は最初の方の図 14.2 で出した通りでゲソ。`(\x -> x * x) (1 + 2)` の方が `(1 + 2)` よりも外側にある (前者が後者を包含している) ので、前者を先に評価するでゲソ。

この Normal Order は、「正規」の名にふさわしく、「ある式 e を正規形まで評価して停止する評価順序が存在するなら、Normal Order で評価すれば必ず停止する」という性質があるでゲソ。つまり、どうやっても計算が停止しないような式でない限りは、Normal Order をとりあえず使っておけば必ず計算が停止するのでめでたいでゲソ!

さっきの式 2 も、Normal Order なら図 14.4 のように停止するでゲソ:

```
(\y -> 1) ((\x -> x x) (\x -> x x))
→ 1
```

図 14.4: 式 2: Normal Order なら停止する

[*7] これは Haskell だと型が付かないでゲソが、Haskell でも再帰を使えば Applicative Order で評価すると停止しない式が書けるでゲソ。今回は再帰は扱わないのでこういう例にしたでゲソ

[*8] 構文木上での preorder 順で一番最初の redex という言い方もできるんじゃなイカ?

14.2.3 チャーチ・ロッサー

これで停止するので安心ですね。でも、評価順序によって停止する場合と停止しない場合があるのなら、評価順序によって別の答えが出たりしないんでしょうか?

評価順序によって停止するか停止しないかが違ってくることはあるでゲソが、停止する場合は全ての停止する評価順序は必ず同じ答えを出す (同じ正規形になって停止する) でゲソ! これはチャーチ・ロッサー (Church-Rosser) の定理というものでゲソ。停止しない場合は答えが出ないんだから気にする必要はないんじゃなイカ?

14.2.4 どの評価順序が評価回数が少ないでゲソ?

よかった、答えが変に変わる心配もしなくていいですね! でも、式 1: `(\x -> x * x) (1 + 2)` を Applicative Order で評価した場合は評価は 3 回で、Normal Order で評価した場合は 4 回だから、1 回増えちゃってます。……ひょっとして、Applicative Order で停止する式であれば Applicative Order で評価すれば、Normal Order より評価の回数は減ったりしないでしょうか?

残念ながら世の中そんな甘くないでゲソ。確かに式 1 の場合だと Applicative Order の方が評価回数が少ないでゲソ。一方で、Normal Order の方が評価回数が少ない場合もあるでゲソ。

式 3: Normal Order の方が Applicative Order よりも評価回数が少ない例

```
(\y -> 1) (2 + 3)
```

式 3 は、「引数 y を受け取って 1 を返す関数」を `(2 + 3)` に適用しているでゲソ。この引数 x の値は結局使われないので評価するだけ無駄でゲソが、Applicative Order だと図 14.5 のように、律儀に評価してしまうでゲソ (式 2 も無限回 vs 1 回で多くかかっているとも言えるでゲソが、式 3 のように両方停止する場合でも Normal Order の方が評価回数が少ないことも実際あるでゲソ)。

```
(\y -> 1) (2 + 3)
→ (\y -> 1) 5
→ 1
```

図 14.5: 式 3: Applicative Order だと 2 回

Normal Order なら図 14.6 のように一発でゲソ! 関数 `(\y -> 1)` の中身 1 を取ってきて、その中にある仮引数 y を実引数 `(1 + 2)` に置き換えるでゲソが…y なんて一度も出てこないので、実引数は完全消滅して、1 が得られるでゲソ!

```
(\y -> 1) (2 + 3)
→ 1
```

図 14.6: 式 3: Normal Order だと 1 回

このように、「Applicative Order での評価が、最終的には要らなくなる引数を無駄に評価していた」場合は Normal Order の方が評価回数が少なく、一方で「Applicative Order では 1 回で済んでいた引数の評価が、Normal Order では複数個にコピーされて別々に評価された」場合は Applicative Order の方が評

価回数が少なくなるでゲソ。更に、Applicative Order でも Normal Order でもない評価順序の方が評価回数が少ない場合もあるでゲソ。たとえば

式 4: Normal Order でも Applicative Order でもない評価順序が回数最小である例

```
((\g -> (g (g (\x -> x)))) (\h -> ((\f -> (f (f (\z -> z)))) (h (\y -> y)))))
```

のような場合でゲソ[*9]。ちょっと長いので、

```
X = (\x -> x)
Y = (\y -> y)
Z = (\z -> z)
```

のように略記すると

式 4: 略記版

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
```

になるでゲソ。この式を Normal Order で評価すると、図 14.7 のようにコピーがたくさん発生して正規形までに 19 回かかるでゲソ。

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\h -> ((\f -> (f (f Z))) (h Y))) X))
→ ((\f -> (f (f Z))) (((\h -> ((\f -> (f (f Z))) (h Y))) X) Y))
→ ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((\f -> (f (f Z))) (X Y)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((X Y) ((X Y) Z)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (((Y ((X Y) Z)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((X Y) Z) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (((Y Z) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((Z Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (Y ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z)
→ ((((\f -> (f (f Z))) (X Y)) Y) Z)
→ ((((X Y) ((X Y) Z)) Y) Z)
→ (((Y ((X Y) Z)) Y) Z)
→ ((((X Y) Z) Y) Z)
→ (((Y Z) Y) Z)
→ ((Z Y) Z)
→ (Y Z)
→ Z
```

図 14.7: 式 4: Normal Order だと 19 回

Applicative Order だと図 14.8 のように 12 回で済むんじゃなイカ?

更に、最小の評価順序は図 14.9 のようなものでゲソ。これは Normal Order、Applicative Order のどちらでもないでゲソ。`(X Y)` の評価を更に 1 回減らして 11 回で評価できるでゲソ。

[*9] 式 4 は [Lamping 1990] に出てきた例をちょっと変えたものでゲソ

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\g -> (g (g X))) (\h -> ((h Y) ((h Y) Z))))
→ ((\h -> ((h Y) ((h Y) Z))) ((\h -> ((h Y) ((h Y) Z))) X))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) ((X Y) Z)))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) (Y Z)))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) Z))
→ ((\h -> ((h Y) ((h Y) Z))) (Y Z))
→ ((\h -> ((h Y) ((h Y) Z))) Z)
→ ((Z Y) ((Z Y) Z))
→ ((Z Y) (Y Z))
→ ((Z Y) Z)
→ (Y Z)
→ Z
```

図 14.8: 式 4: Applicative Order だと 12 回

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\h -> ((\f -> (f (f Z))) (h Y))) X))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\f -> (f (f Z))) (X Y)))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\f -> (f (f Z))) Y))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) (Y (Y Z)))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) (Y Z))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) Z)
→ ((\f -> (f (f Z))) (Z Y))
→ ((\f -> (f (f Z))) Y)
→ (Y (Y Z))
→ (Y Z)
→ Z
```

図 14.9: 式 4: 最小回数の評価順序はそのどちらでもなく 11 回

じゃあ、最終的にいずれ必要になる引数の場合だけ Applicative Order で評価して、残りの要らないっぽい引数は Normal Order で遅延評価する、みたいなことをすれば良いのではないでしょうか?

そうでゲソね。ただ、「最終的にいずれ必要になる引数」の判定は**正格性解析**とも関連する話で、完全な解析は決定不能でゲソ。近似アルゴリズムがいくつかあるようでゲソ。興味のある人は [Sakai 2010] あたりを見るといいんじゃなイカ? この場合は、評価順序を決めるのに解析が必要でゲソが、コンパイラに実装するには良いんじゃなイカ?

あと、Normal Order だと評価回数が増えるだけでなく、必要なメモリ量が激しく増大する場合もあるでゲソ (たとえば [Sakai 2010] 冒頭にそういう例が載っているでゲソ) が、今回そういう話もしないでゲソ。

まぁ、今日はその話は置いといて、単純で安心な Normal Order で評価しつつ、評価回数を評価ステップの実装の工夫で減らす方法を紹介するでゲソ。Normal Order の場合問題はコピーの発生だったので、コピーを最小限に抑える方向性を追求するでゲソ。こちらの方向性も、評価回数の本当の最小化、まで目指すと β 簡約以外の処理時間が指数時間になるとかいうアルゴリズムがあるでゲソが、そこまでがんばらない方法は今も主に Haskell などの遅延評価をするプログラミング言語で広く使われているでゲソ。

まとめ

1. 一般に、評価順序は複数ある
2. 評価順序によって計算が停止するか停止しないかが変わる場合がある
3. が、計算が停止する場合は計算結果 (正規形) は一意に定まる
4. 正規形に到達する評価順序が存在すれば、Normal Order での評価も必ず正規形に到達して停止する
5. 評価順序によって評価の回数は異なることがあり、Normal Order の方が良い場合も Applicative Order の方が良い場合もそれ以外の順序が良い場合もある

14.3　グラフ簡約しなイカ?

そういえばイカ娘さん、質問があるのですが。式 1 の Normal Order での評価の時にこんな風に

```
→ (1 + 2) * (1 + 2)
→ 3 * (1 + 2)
→ 3 * 3
```

評価していましたが、2 箇所ある `(1 + 2)` を別々に評価する必要はあるのでしょうか?

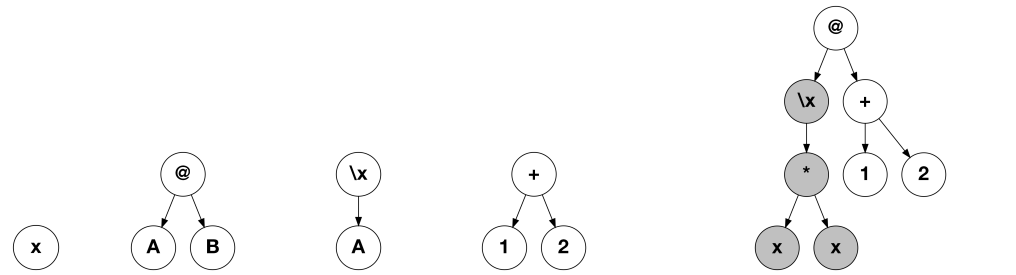
いい指摘でゲソ。文字列ベースで考えると、`(1 + 2)` は 2 回別々に出てくるので、2 回評価するのが筋な気がするでゲソ。

しかし、これだと Applicative Order の時よりも 1 回多く評価することになるでゲソ。この例だと評価回数の差は 1 回でゲソが、場合によってはもっとたくさん差がつくでゲソ。Normal Order だと評価回数が指数爆発することもあるでゲソ。

この 2 つの `(1 + 2)` は、もともと一つだったものが β 簡約の時に 2 つに増えたものでゲソ。この「β 簡約の時に引数がコピーされる」というのが Normal Order の評価回数の増加原因でゲソ。どうにかしてコピーをせず、余計な評価を増やさずに評価できないものか…それができる方法の一つに、グラフ簡約という評価の実装方法があるでゲソ。引数をコピーせずに 1 つだけ持っておいて共有することで、コピーも余分な評価も防ぐでゲソ。これが編み出されたことで、引数のコピーを防いで評価回数を減らすことができたでゲソ。

14.3.1 式のグラフ表現

グラフ簡約を説明するために、少し準備するでゲソ。さっきまでは式を文字列で書いていたでゲソが、ここでは式をグラフで表現する方法を説明するでゲソ。グラフにすると文字列よりも色々操作しやすいでゲソ[*10]。表 14.1 の式はそれぞれ、図 14.10 (a)〜(d) のようにツリーで表現できるでゲソ:



(a) 変数 `x`　(b) 関数適用 `(A B)`　(c) 関数 `(\x -> A)`　(d) 四則演算・数値定数 `1 + 2`　(e) `((\x -> x * x) (1 + 2))`

図 14.10: 式のグラフ表現

変数 `x`　は、図 14.10 (a) のように 1 つのノードに変数名を書いて表現するでゲソ。

関数適用 (`A B`)　は、図 14.10 (b) のように、関数適用を表す `@` ノードを使って表すでゲソ。`@`ノードの左右の子が、A, B にそれぞれ対応しているでゲソ。

関数/λ 抽象 (`\x -> A`)　は、図 14.10 (c) のように、λ 抽象を表す `\x` ノードを使って表すでゲソ (`x` の部分は変数名でゲソ)。`\x` ノードの子が関数本体 A に相当するでゲソ。

四則演算・数値定数　は、図 14.10 (d) のように、ノードに演算子 (この場合は `+`) や数値 (`1`, `2` など) を書いて表すでゲソ。演算子ノードの左右の子が被演算数でゲソ。この場合は `(1 + 2)` を表しているでゲソ。

図 14.10 (e) は、式 `((\x -> x * x) (1 + 2))` の抽象構文木表現でゲソ。`\x` とその子孫ノードからなるサブツリー (灰色の部分) が、部分式 `(\x -> x * x)` と対応しているでゲソ。

14.3.2 文字列ベース簡約のグラフ表現

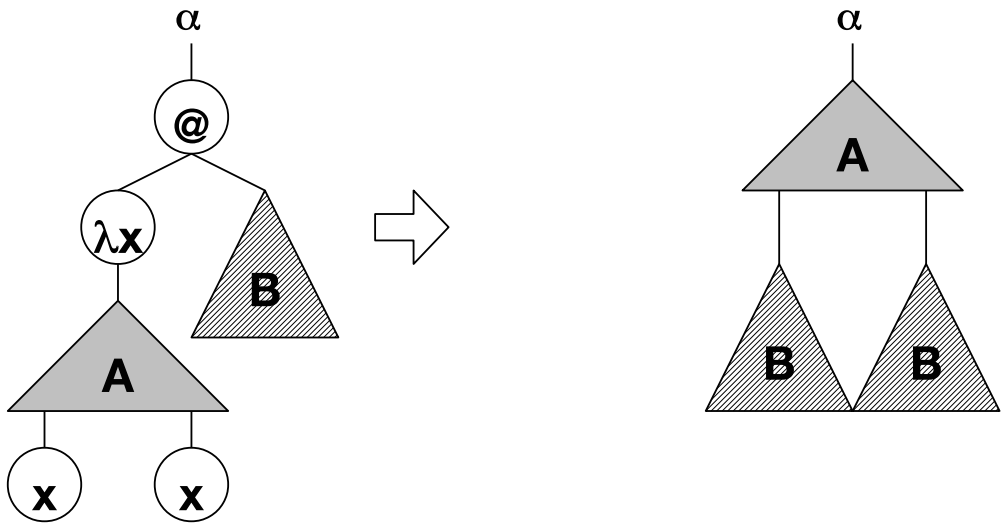
最初に説明した文字列ベースの β 簡約 (文字列ベース簡約) をグラフで表現すると図 14.11 みたいな感じになるでゲソ。これは $\alpha = (\lambda x.A)B$ の β 簡約を表しているでゲソ。図 14.11 の左側では、評価前の $\alpha = (\lambda x.A)B$ の状態を表しているでゲソ。左下の 2 つの x と書かれたノードが A の下にぶら下がっているのは、A の中に 2 回 x が出てくることを表現しているでゲソ。これを β 簡約すると右側のように、A の中の x があった場所に B をコピーしてつなげたものになるでゲソ。

図 14.12 で、式 1: `(\x -> x * x) (1 + 2)` を文字列ベース簡約による Normal Order での評価をグラフで表現してみたでゲソ。これは 図 14.2 に対応しているでゲソ。図 14.12 の灰色のノードとその下にあるノード (枠で囲まれている部分) が、次に評価される redex を表していて、図 14.2 で下線が引かれた部分に対応しているでゲソ。

14.3.3 グラフ簡約

式 1 の Normal Order 評価で評価回数が増えたのは、1 回目の評価で `(\x -> x * x)` の関数 body の中に `x` が 2 回出てくるのにあわせて、`(1 + 2)` を 2 つにコピーしたからでゲソ。図 14.11 でも、「B」のサブツリーが 2 つにコピーされているでゲソ。コピーした以上は、後でコピーごとに別々に評価されると

[*10] ここでは式は「抽象構文木」というツリーで表現できるでゲソが、後でツリーじゃなくて一般のグラフに拡張していくので、「グラフ表現」という言い方をしておくでゲソ。



(\x -> A) B の文字列ベースの β 簡約:

1. A の中に現れる x を B のコピーにそれぞれ置き換える
2. (\x -> A) B を 1. で書き換えた後の A に置き換える

図 14.11: 評価の実装方法その 1: 文字列ベース簡約

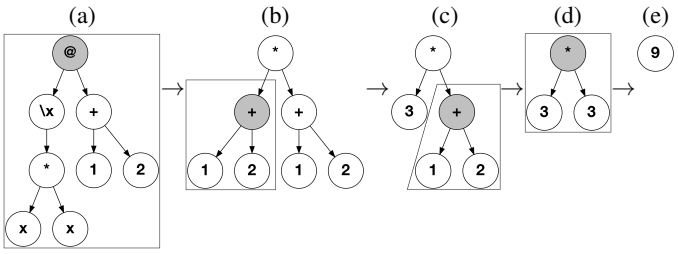


図 14.12: 式 1: Normal Order 評価、文字列ベース簡約

評価回数が無駄に増えるでゲソ。でも、(1 + 2) という式は参照透明なので何度評価しても変わらないし、コピーせずに 1 回だけ評価して済ませたいでゲソ。こんな感じで、1 回の評価で 2 回出てくる (1 + 2) を評価できればいいんじゃなイカ?

```
(\x -> x * x) (1 + 2)
→ (1 + 2) * (1 + 2)
→ 3 * 3
→ 9
```

それを実現する方法が **グラフ簡約 (Graph Reduction)** でゲソ。グラフ簡約では、関数適用の評価時に仮引数 x が関数 body の中に複数回出てきても、実引数をコピーせずに、実引数へのポインタをセットするなどして共有させることで、コピーを減らし、評価回数を減らしているでゲソ。元々の文字列ベースでの評価では式が構文木 (つまりツリー) になっていたでゲソが、グラフ簡約では部分式が共有されるので式がグラフになるのでこの名前になっているでゲソ。グラフ表現で図にすると図 14.13 みたいな感じじゃなイカ?

ここで、$\alpha = (\lambda x.A)B$ を β 簡約する時に、B を 2 つコピーせずに、共有しているでゲソ。これで、B の評価は 1 回だけで済むでゲソ!

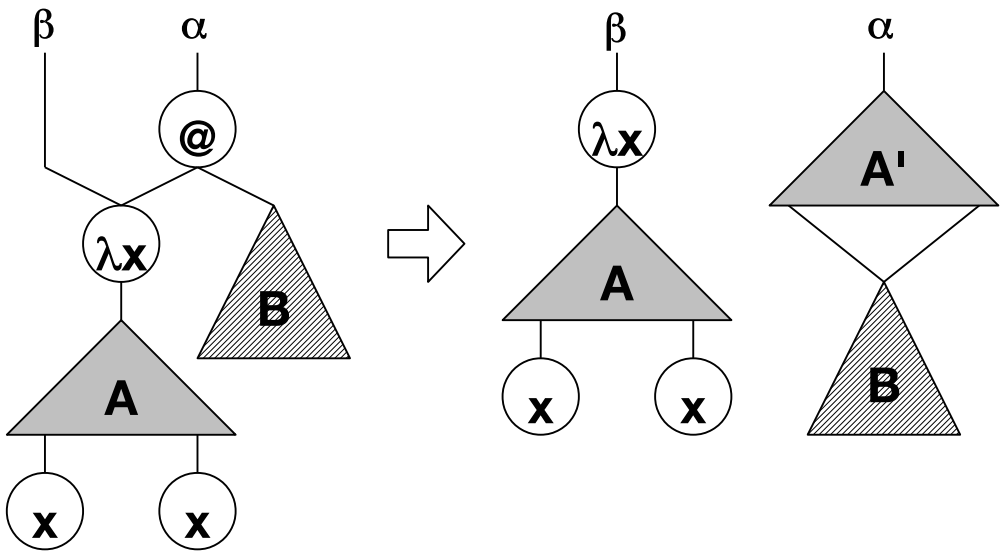
ちょっと待ってください。それはとても良いことなのですが、この図 14.13 を見ると今度は A がコピーされているように見えるのですが、これはなぜでしょうか?

それは、A が別の所 (図 14.13 の β) から共有されているかもしれないからでゲソ。なので、文字列ベース簡約の時のように直接 $(\lambda x.A)B$ の中の A を書き換えるとまずいでゲソ。そこで、A の部分をコピーして A' を作ってから、A' の中の x を B へのリンクに書き換えているでゲソ。文字列ベース簡約の場合はそもそも共有されるような場合がないからコピーせずに直接書き換えてよかったでゲソ。

図 14.14 に、式 1 のグラフ簡約での評価を示すでゲソ。ここでも灰色のノードとその下にあるノード (枠で囲まれている部分) が、次に評価される redex を表すでゲソ。以降の図でも灰色のノードが次に評価される redex を表すでゲソ (但し枠は別の意味で使っていたりするでゲソ)。図 14.12 (b) で 2 つにコピーされていた `(1 + 2)` の項が、図 14.14 (b) では共有されて 1 つになっているじゃなイカ! この共有された `(1 + 2)` が図 14.14 (b) → (c) で 1 回で評価されて、全体として 3 回で評価が完了するでゲソ!

グラフ簡約は部分式が共有されるので文字で表現しにくいでゲソが、共有部分式に `let` で名前を付けて表現すると図 14.15 のようになるでゲソ。`let x = hoge in fuga` と書くと、`hoge` に `x` という名前を付けて `fuga` の中で使うでゲソ。グラフ簡約でのノードに名前を付けて共有関係を示すためにここでは使っているでゲソ。`let` に関する操作は β 簡約とかではないので「評価ステップ」の回数にはカウントしないことにするでゲソ。

グラフ簡約は、Normal Order だけじゃなく、他の評価順序の場合でも評価回数を減らすこともできるでゲソ。また、文字列ベース簡約では評価順序をどのようにしてもできなかった少ない評価回数を、グラフ簡約で実現することもできるでゲソ。たとえば、文字列ベース簡約では 11 回までしか評価回数を減らせなかった式 4 も、グラフ簡約を使えば図 14.16 のように 10 回まで減らせるでゲソ! 図 14.16 (d) で `(X Y)` を、図 14.16



`(\x -> A) B` のグラフ簡約:

1. `A` をコピーして `A'` を作る
2. `A'` の中に現れる `x` を `B` へのポインタで置き換える
3. `(\x -> A) B` を 2. で書き換えた後の `A'` に置き換える

図 14.13: 評価の実装方法その 2: グラフ簡約

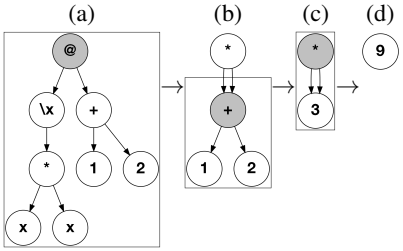


図 14.14: 式 1: Normal Order 評価、グラフ簡約

```
(\x -> x * x) (1 + 2)
→ let x = (1 + 2) in x * x
→ let x = 3 in x * x
→ 9
```

図 14.15: 式 1: Normal Order 評価、グラフ簡約

(h) で (Z Y) を評価しているでゲソが、これらは 2 箇所から共有されているでゲソ。図 14.8 で文字列ベース簡約をした場合には、(X Y) と (Z Y) の簡約が 2 回ずつ必要だったでゲソが、グラフ簡約では共有されているので 1 回ずつで済むので、全体で評価回数が 2 回減っているでゲソ。

14.4 Fully Lazy グラフ簡約

でもやっぱりこれだと A のコピーが増えてませんか?

……その通りでゲソ。A が丸ごとコピーされるので、その中にある関数適用は無駄に複数回評価されることがあるでゲソ。たとえば式 5 のような場合でゲソ。これをグラフ簡約を使って Normal Order で評価すると、図 14.17 (文字列表現)、図 14.18 (グラフ表現) のようになるでゲソ。

式 5: グラフ簡約 (Normal Order) でも無駄な評価が出る例

```
(\f -> (f 1) + (f 2)) (\x -> x + (\y -> y * 5) 6)
```

グラフ簡約でも 10 回かかるじゃなイカ! (\x -> (x+((\y -> (y*5)) 6))) の 1 と 2 への適用を評価する時に、(x+((\y -> (y*5)) 6))) がコピーされるでゲソ。図 14.18 (b) → (c) で、枠で囲まれた (x+((\y -> (y*5)) 6))) が 2 つに増えているのが分かるんじゃなイカ? それが後で x=1 の時 (図 14.18 (c) → (d) → (e)) と x=2 の時 (図 14.18 (g) → (h) → (i)) とで別々に評価されるでゲソ。このため、((\y -> (y*5)) 6) と (6*5) の評価がそれぞれ 2 回ずつ行われているでゲソ。

でも、((\y -> (y*5)) 6) の部分は x を含まないので、別々の x の値に対して適用しても結果は変わらないでゲソ。なので、この部分はコピーせずに共有して、全体で一回評価すればいいはずでゲソ。

一般的には、(\x -> A) B の評価をする時に、A を全部コピーするんじゃなくて、A のうち、「x に関係しない部分」はコピーせずに共有すればいいでゲソ。x と関係なければ、A が別の所で共有されていて、別の x の値に適用されても値は変わらないから問題ないんじゃなイカ! こういう方法を **Fully Lazy グラフ簡約 (Fully Lazy Graph Reduction)** と呼ぶでゲソ。

グラフで図示すると図 14.19 のようになるでゲソ: *A* の中の黒い部分 (*A* の中の「x に関係しない部分」を表しているでゲソ) はコピーせずに共有して、*A* のそれ以外の部分だけコピーしているでゲソ。

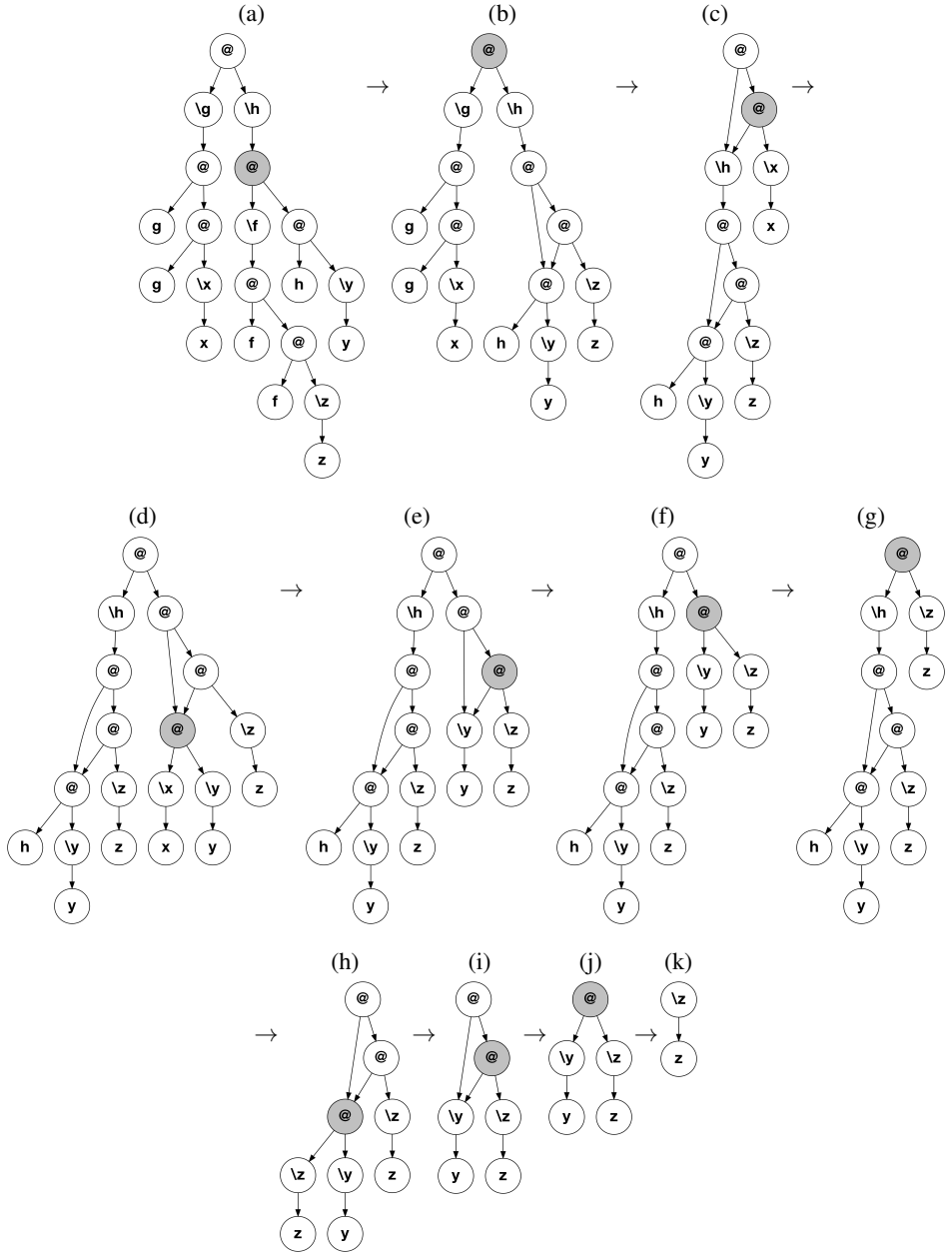


図 14.16: 式 4: Applicative Order 評価、グラフ簡約

具体的に「x に関係しない部分」が何かについてはいくつか手法あるでゲソが、一番簡単なのは Maximal Free Expression を使う方法でゲソ。**Free Expression** とは、自由変数を含まない式のことでゲソ。**Maximal Free Expression** とは、Free Expression のうち maximal なもの、つまり他の Free Expression の一部に含まれてはいないもののことでゲソ。例えば、`(x+((\y -> (y*5)) 6))` の場合、Free Expression は `5`, `6`, `(\y -> (y*5))`, `((\y -> (y*5)) 6)` でゲソ。この中で Maximal Free Expression は `((\y -> (y*5)) 6)` でゲソ。

`(\x -> A) B` の `A` の中にある Maximal Free Expression は自由変数を含まないので文脈とは独立に評価して値が決定できるのでゲソ。特に、`x` も含まないのでゲソ。なので、評価する時にもコピーする必要がないのでゲソ! Maximal とついているのは、単に効果を最大化するためでゲソ。

Maximum Free Expression を使って Fully Lazy グラフ簡約 (Normal Order) すると、式 5 の場合は図 14.20

```
((\f -> ((f 1)+(f 2))) (\x -> (x+((\y -> (y*5)) 6))))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((f 1)+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+((\y -> (y*5)) 6))+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+(6*5))+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+30)+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in (31+(f 2))
→ (31+(2+((\y -> (y*5)) 6)))
→ (31+(2+(6*5)))
→ (31+(2+30))
→ (31+32)
→ 63
```

図 14.17: 式 5: Normal Order 評価、グラフ簡約: 10 回

(文字列表現)、図 14.21 (グラフ表現) のようになるでゲソ。図 14.20 の 3 行目で、`f` を `1` に適用する前に、Maximum Free Expression である `((\y -> y * 5) 6)` を `z` として括りだして、4 行目でそれ以外の部分 `(\x -> (x+z))` はコピーして関数適用を評価しているでゲソ (括りだされた `z` はコピーしないでゲソ)。グラフで見ると、図 14.18 (c) では 2 つにコピーされている `((\y -> y * 5) 6)` (枠で囲まれた部分) が、図 14.21 (c) では共有されているのが分かるんじゃなイカ?

14.5 Optimal Reduction

ここまでくれば完璧、でしょうか?

実はまだまだ最適じゃないでゲソ! 最適への道は長いでゲソ……。イカのような式 6 を考えてみるでゲソ。

式 6

```
((\f -> ((f 1) + (f 2))) (\x -> ((\y -> (x + (y * 5))) 6)))
```

前節で示した Fully Lazy グラフ簡約だと、Normal Order の場合図 14.22 のように 10 回評価が必要でゲソ。

これは `((\y -> (x+(y*5))) 6)` が 2 個にコピーされて、`6` への適用と `y * 5` の掛け算がそれぞれ 2 回ずつ行われたから、その分回数が増えているでゲソ。`y * 5` は結局 `6 * 5` に評価されるので、`x` の値に依存しないでゲソが、`y * 5` は Free Expression ではない (`y` が自由変数) ので共有できないでゲソ。`((\y -> (x+(y*5))) 6)` の中にある Free Expression は `5` と `6` だけでゲソ。

これを解決する手法もあるにはあるでゲソ。究極的には、`(\x -> A) B` を評価する時に、「自由変数 `x` を含まない部分式」はコピーを省ける、かもしれないでゲソ (だって `x` を含まないのでゲソからね)。前節の Fully Lazy グラフ簡約では、「自由変数を全く含まない部分式」という、それより強い条件を課してコピーを省いていたので、こういう無駄が出てきたでゲソ。ただ、「自由変数 `x` を含まない部分式」だけだと他の自由変数を含むことがあったり諸々の事情で、アルゴリズムは一気に複雑化するでゲソ…。

そのうちのひとつ、[Lamping 1990] の手法の雰囲気を紹介するでゲソ[*11]。これは、β 簡約の回数を最小化するという意味で **Optimal Reduction** と呼ばれる手法の一つでゲソ。雰囲気としては、図 14.23 のような感じでゲソ。A の上に▽の形をしたノードと、A の下に△の形をしたノードがあって、$*$ と○の記号がついているでゲソ。これは、「上から辿った時に、▽の $*$ から入った場合 (つまり β から辿った場合) には、△から下

[*11] 難しすぎて理解できなかったので雰囲気だけでゲソ

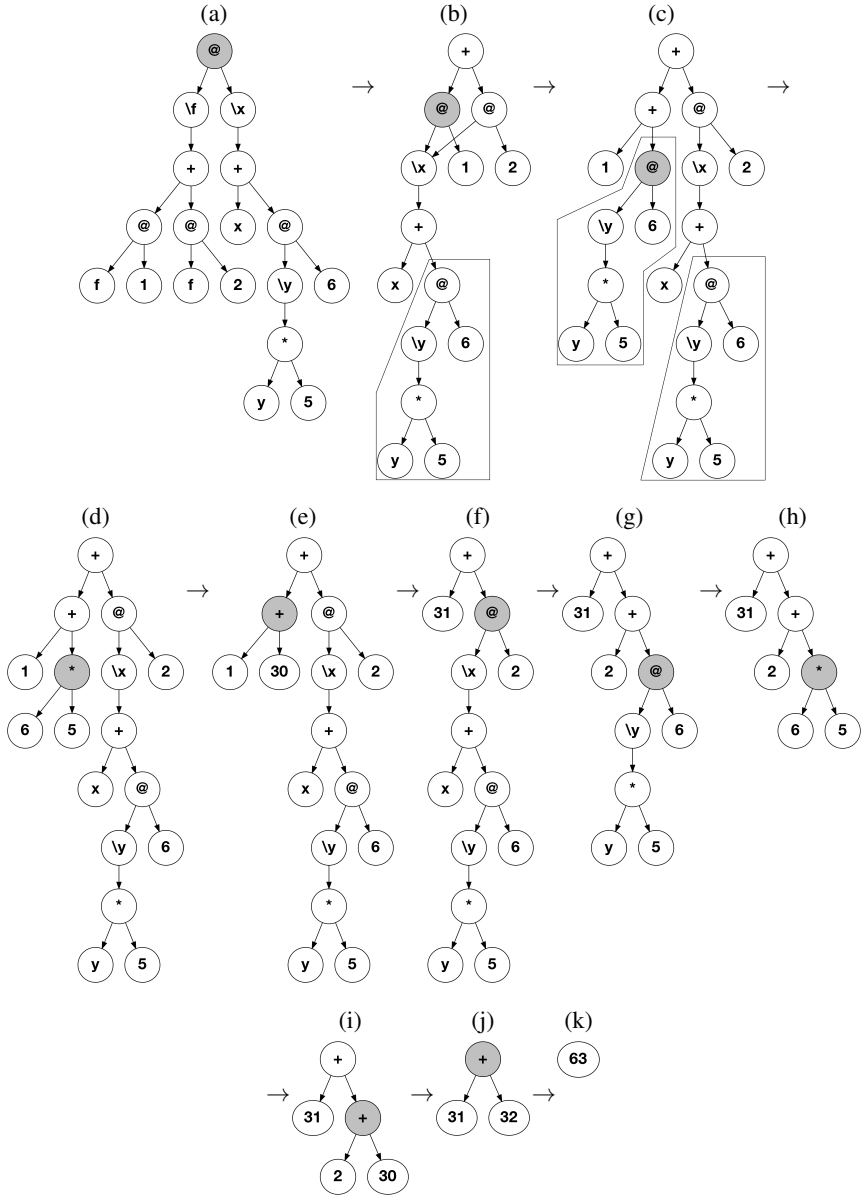


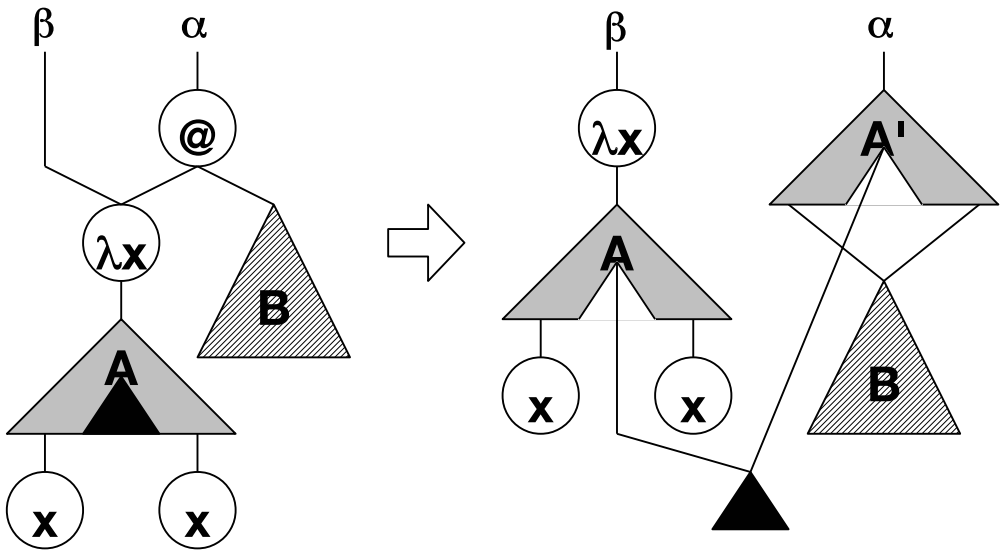
図 14.18: 式 5: Normal Order 評価、グラフ簡約: 10 回

に辿るときも ∗ の方 (*x*) に行く。逆に上から▽の○印の方から入った場合 (*α* からたどった場合) は、△から下に辿るときも○の方 (*B*) に行く。」という意味でゲソ。このようにすることによって、*A* の構造を共有したまま、*x* の部分だけが違う複数の文脈を表現できるでゲソ。

図 14.24 に、[Lamping 1990] による式 6 の評価の雰囲気を示すでゲソ。Normal Order じゃなくなっているでゲソが、雰囲気だけ感じて欲しいでゲソ。

(a) から始まって、(b) までは普通でゲソ。

(b) で (`\x -> (\y -> x + y * 5) 6`) を `1` と `2` に適用したものを評価して (評価 2 回分) (c) のようにするでゲソ。ここで、(`\y -> x + y * 5) 6`) はほとんどがコピーされずに残っていて、(`\y -> [q] + y * 5) 6` のようになっているでゲソ。[q] には `1` か `2` が入るでゲソが、そのどちらが入るかは、上からたどって [p] に入る時、[p] に向かう 2 本の枝 (`*`と `o` のラベルが付いた枝で、それぞれ (`\x -> (\y -> x + y * 5) 6`) を `1` と `2` に適用した場合に対応するでゲソ) のどちらが使われるかに連動して決まるでゲソ。[p] に入る時に



(\x -> A) B の Fully Lazy グラフ簡約:

1. A をコピーして A' を作る。但し、x に関係ない部分 (図中黒三角) はコピーせずに、A と A' とで共有する。
2. A' の中に現れる x を B へのポインタで置き換える
3. (\x -> A) B を 2. で書き換えた後の A' に置き換える

図 14.19: 評価の実装方法その 3: Fully Lazy グラフ簡約

```
((\f -> ((f 1)+(f 2))) (\x -> (x+((\y -> (y*5)) 6))))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((f 1)+(f 2))
= let z = ((\y -> (y*5)) 6) in let f = (\x -> (x+z)) in ((f 1)+(f 2))
→ let z = ((\y -> (y*5)) 6) in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = (6*5) in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = 30 in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = 30 in let f = (\x -> (x+z)) in (31+(f 2))
→ let z = 30 in (31+(2+z))
→ (31+32)
→ 63
```

図 14.20: 式 5: Normal Order 評価、Fully Lazy グラフ簡約: 8 回

*の枝を使えば [q] の値も*の枝を、[p] に入る時に o の枝を使えば [q] の値も o の枝を使う感じでゲソ。このような苦肉の策によって、(\y -> x + y * 5) 6 の構造のほとんどを共有したまま、評価を進めることができるのでゲソ。

(c) → (d) → (e) で、y * 5 あたりを重複なしに評価した後、(e) → (f) では (\x -> (\y -> x + y * 5) 6) を 1 に適用した時相当の評価を*の枝を使ってしているでゲソ。一番上の + ノードの左側の子ノードを評価しているでゲソが、[p] に*の枝から入っているので、[q] の値はそれと連動して 1 になるでゲソ (太い矢印で示したツリーが使われるでゲソ)。同様に、(f) → (g) では (\x -> (\y -> x + y * 5) 6) を 2 に適用し

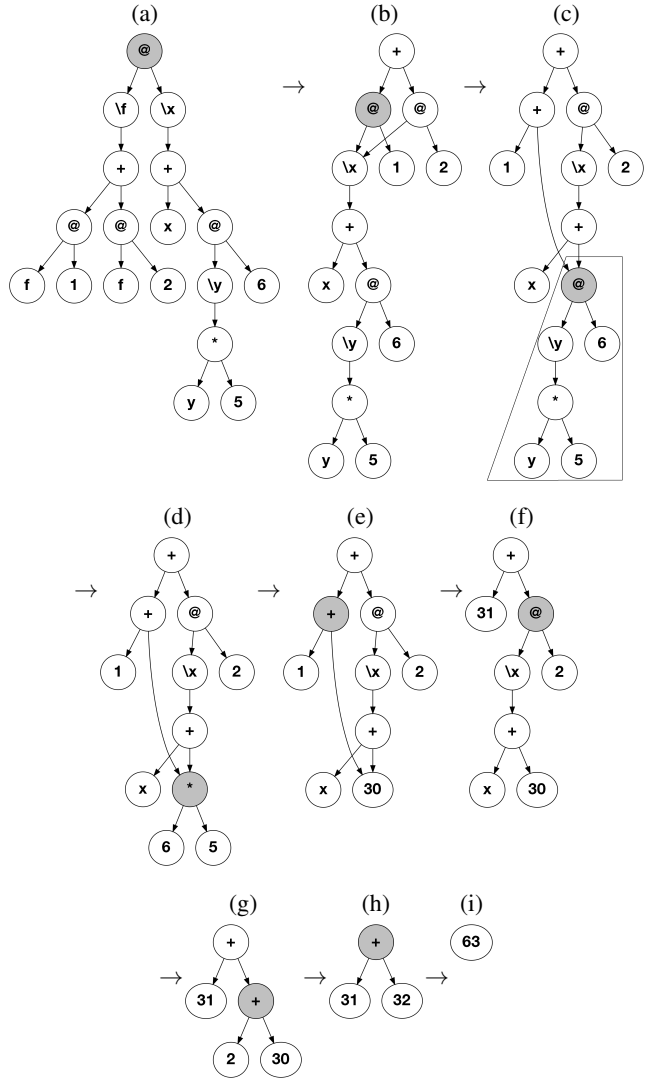


図 14.21: 式 5: Normal Order 評価、Fully Lazy グラフ簡約: 8 回

た時相当の評価を o の枝を使ってしているでゲソ。このようにして、8 回の評価で評価が完了したでゲソ!! ここでは説明の都合上 Normal Order じゃない評価を示したでゲソが、本式の Optimal Reduction では Normal Order 評価でも 8 回で済むはずでゲソ。

この [Lamping 1990] の手法は、元論文とか見るともっと多くの種類のノードとかエッジとかがあって、グラフ簡約ルールが 30 個以上もある複雑なものでゲソ。どうやら、これは β 簡約回数を最小化するらしいでゲソが、β 簡約以外にすることが増えて (▽ノードとかの管理でゲソね)、評価 1 回あたりの計算量が指数時間かかる、というヤバイ状況らしいでゲソ。後に他の手法も考えられたようでゲソが、どのみち私には難しすぎてよく分からなかったでゲソ。追求したい人はなんかそれだけで一冊本 [Asperti+ 1998] が出ているようなのでそれ読めばいいんじゃなイカ?[*12]

[*12] 私は読んでないでゲソ…

```
((\f -> ((f 1)+(f 2))) (\x -> ((\y -> (x+(y*5))) 6)))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((f 1)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in (((\y -> (1+(y*5))) 6)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((1+(6*5))+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((1+30)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in (31+(f 2))
→ (31+((\y -> (2+(y*5))) 6))
→ (31+(2+(6*5)))
→ (31+(2+30))
→ (31+32)
→ 63
```

図 14.22: 式 6: Normal Order 評価、Fully Lazy グラフ簡約: 10 回

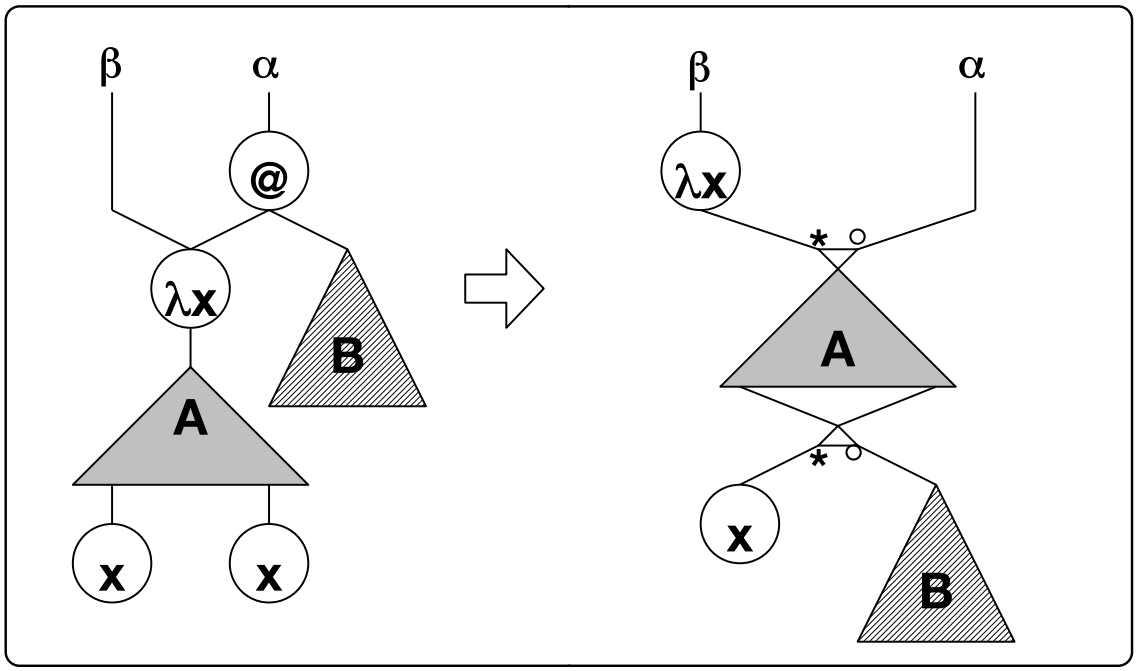


図 14.23: 評価の実装方法その 4: Optimal Reduction

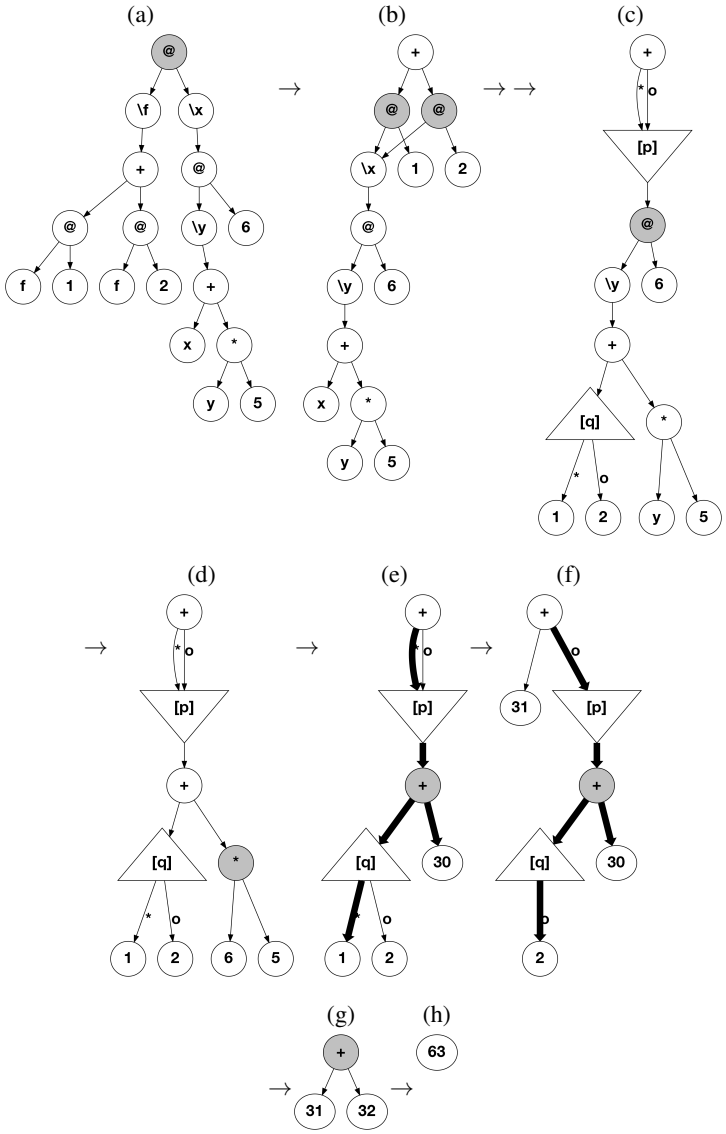


図 14.24: 式 6: Optimal Reduction

今日は、評価の回数を「評価順序の変更」と「評価方法の改良」の二つの点から少なくする話をしたでゲソ。割と「プログラムをそのまま」「Normal Order で」評価することにこだわっていたので、プログラム解析をしたりプログラムを変換してから実行するような話は扱わなかったでゲソ。
そういう、今回扱わなかった話の一つ目は、コンパイラ最適化でゲソ。たとえば、今日の話は `(1 + 2)` がコピーされてどんどん増えていくのを防いで評価回数を減らそう、という話でしたでゲソが、最初からイカのような式が書かれていた時にはどうにもならないでゲソ。

式 7
```
((1 + 2) * (1 + 2))
```

この場合、二つの `(1 + 2)` はコピーで増えたわけではないので、今日の話の範囲ではこのまま別々に評価することになるでゲソ。

でも、解析すれば同じ式であることを検知して一つにまとめることはできそうですよね?

もちろん、できるでゲソ。それは、評価方法の改良というよりは、コンパイラ最適化の一つである**共通部分式除去 (Common Subexpression Elimination (CSE))** でゲソ。他にもコンパイラの最適化手法はたくさんあるので、色々早くすることができるはずでゲソ。
扱わなかった二つ目は、(Fully Lazy) Lambda Lifting 関連の話でゲソ。今回は、Fully Lazy グラフ簡約のあたりで Free Expression を評価の時に括りだしていたでゲソが、そういうのを Lambda Lifting というプログラム変換を使って前もって括りだしておく方法もあるでゲソ。これもコンパイラ実装向きな話じゃなイカ?

14.6 まとめと参考文献

結局、今日の話をまとめるとどんな感じになるんでしょうか?

迷ったらとりあえず Normal Order でグラフ簡約すればいいんじゃなイカ? 評価順序も、評価の実装方法も色々工夫できるので面白いでゲソが、突き詰めすぎると底なし沼でゲソ!

参考文献

SPJ 1987 S. Peyton Jones. The Implementation of Functional Programming Languages. Prentice-Hall, Inc., 1987.
`http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/`
評価順序の話から基本的なグラフ簡約の話などだいたいカバーしているでゲソ! 今回は省いた実装やWHNF、再帰関連の話なども豊富でゲソ。

Sumii 2005 住井英二郎. ラムダ計算入門 (2005 年度「計算機ソフトウェア工学」授業資料)
`http://www.kb.ecei.tohoku.ac.jp/~sumii/class/keisanki-software-kougaku-2005/lambda.pdf`
λ計算・β簡約などの基礎の話がまとまっているでゲソ。

shelarcy 2007 本物のプログラマは Haskell を使う 第 8 回 遅延評価の仕組み
`http://itpro.nikkeibp.co.jp/article/COLUMN/20070305/263828/`.
遅延評価の基本から、Haskell における遅延評価の実際、弱頭部正規形 (Weak Head Normal Form (WHNF))、豊富な参考リンクなど。

Sakai 2010 酒井 政裕. Haskell Advent Calendar 2010 22 日目: 正格性解析 (Strictness Analysis)
`http://msakai.jp/d/?date=20101228`.
正格性解析を用いて評価順序を変える話でゲソ。

Lamping 1990 J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30, 1990.

Optimal Reduction のアルゴリズムの一つでゲソ。ファンシーな部品がたくさんあるグラフの図が載っているでゲソ。

Levy Jean-Jacques Le'vy, Reductions and Causality (講義スライド)
`http://pauillac.inria.fr/~levy/courses/tsinghua/reductions/`
Optimal Reduction 理解の参考にしたでゲソ。

Asperti 2009 Andrea Asperti, "The optimal reduction of lambda expressions" (スライド)
`http://www.cs.unibo.it/~asperti/SLIDES/optimal.pdf`
Optimal Reduction のヤバさを感じるスライドでゲソ。

Asperti+ 1998 A. Asperti, S. Guerrini, "The Optimal Implementation of Functional Programming Languages", Cambridge Tracts in Theoretical COmputer Science n.45, Cambridge University Press, 1998.
Optimal Reduction 関係で本一冊書けるらしいでゲソ。

第15章

オール・アバウト・ケーゾク・イン・スキーム

— @yryozo

お主たち！ 今日も元気に括弧をつけてるでゲソか?

括弧の乱れは心の乱れ、心の乱れは家庭の乱れ、(略)、そして国の乱れは宇宙の乱れ、とどこかの偉い人も言ってたような気がするでゲソ。これからも精進してカッコいいプログラマーを目指すでゲソよ。

さて、前回は Scheme にとって大事な「末尾呼び出しの最適化 (Tail Call Optimization, TCO)」の話をしたでゲソ [*1]。でも Scheme にとって大事なものはもう一つ [*2]あるでゲソ。そう「継続 (Continuation)」でゲソね。というわけで、今回は「継続」の話をしておこうと思うんでゲソ。

[*3]

15.1 「継続」について勉強しなイカ?

まず最初に「そもそも継続ってなんなのか」を簡単に説明しておくでゲソ。

といっても、こんな本を読んでいるくらいだから「継続」っていう言葉は既に聞いたことがあるんじゃなイカ? そんなよく訓練されたお主たちのために超スピードで説明してみるでゲソ!

(1) 継続って何かというと「その後に行われる計算処理全体」のことでゲソ。
(2) さらに、一部の言語ではこの「継続」を実行時に捕捉して使うことができるんでゲソ (一部の言語っていうのは、例えば Scheme とか Scheme とか Scheme とかでゲソ。後は Ruby とか Standard ML とかもそうでゲソ)。
(3) で、そういった言語では「捕捉した継続」を好きなときに「起動」できるんでゲソ。継続を起動すると、捕捉されていた「その後に行われる計算処理」が実行されるんでゲソ。

... うむ、正直、何がなにやらよく分からないでゲソね (汗。なんというか、「わけがわからないよ」とか「この概念は出来損ないだ。使えないよ」という雰囲気がプンプンするでゲソ。

じゃあ、これをちょっと別の言い方で説明してみるでゲソ。

(1) 継続って何かというと、「その時点でのスレッドの状態」のことでゲソ。
(2) 一部の言語には、「スレッドの状態のコピーを取る」機能が用意されているんでゲソ。(そしてこの「コピーを取る処理」のことを「継続の捕捉」と呼ぶんでゲソ)
(3) で、取得したコピーを使うと、スレッドの状態を「コピーを取ったときの状態」に変更することができるんでゲソ。(そしてこの「状態を変更する処理」のことを「継続の起動」と呼ぶんでゲソ)

... え、さっきと言ってることが違い過ぎなイカって? うーん、確かに一見すると全然違う話のように見えるでゲソね。だけど言ってることはほとんど同じなんでゲソ。

さっきの話だと、継続というのは「今後行う計算処理」だったでゲソ。だから継続を捕捉するというのは

[*1] といっても、前回の話との繋がりは一切ないので今回から読んでもらって全く問題ないでゲソ。

[*2] もちろん正確にはもっと一杯あるでゲソ。なにしろ Scheme は全てが大事と言っても過言ではないでゲソ。

[*3] ところで、タイトルはちょっとニンジャ○レイヤーっぽいでゲソが、中身はまるっきり関係ないでゲソ。タイトルだけの完全な出落ちでゲソね。まぁ、バー○ンハウスにでも引っかかったと思って許して欲しいでゲソ。

「今後行う計算処理を取得すること」で、継続を起動するのは「取得した残りの計算処理を行うこと」ということでゲソ?

ところで、この「今後行う計算の取得」と「その実行」って、OS が日常的に行ってる「ある処理」に似てなイカ?

そう、スレッドのコンテキストスイッチでゲソ! スレッドを「いったん寝かせて処理を停止」しておいて、別のタイミングで「起こして計算を再開」させるというのは、まさに「今後行う計算を取得」しておいて「その処理を実行」することでゲソ。実際問題、計算処理を実行しているスレッドの状態を完全に記録することができれば、そのスレッドがこれから行おうとしていた処理を完全に再現することは可能でゲソ? こう考えれば、「継続」=「その後に行われる計算処理」=「スレッドの状態」というのも自然じゃなイカ[*4]。

さて、これでなんとなく分かってきたかもしれないでゲソが、「まだ具体的なイメージが湧かないよ」という人もいると思うので、もうちょっとだけ具体的に説明しておくでゲソ。

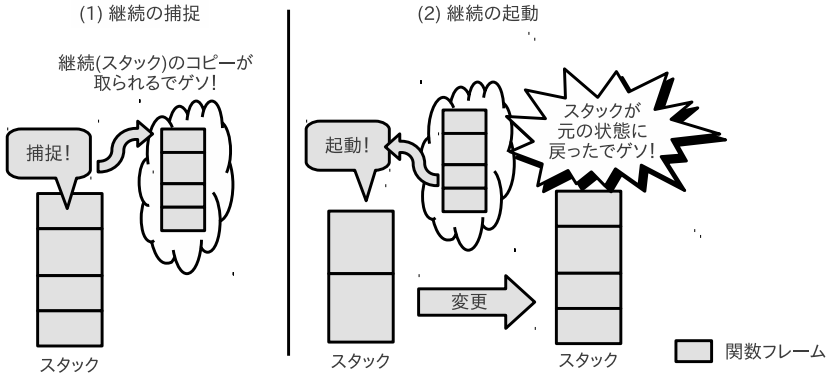
「スレッドの状態」というのは (最近の大抵の言語処理系では)「スタックの中身」のことでゲソ? [*5]例えば、今プログラムのどこを実行しているか (いわゆるプログラムカウンタの値でゲソね) とか、リターンが起こった時にどこに制御を戻すか (いわゆるリターンアドレスでゲソね) とか、今後の実行を続けるために必要な情報は全部スタックの中に入っているじゃなイカ?

だからぶっちゃけてしまうと、(大抵のケースでは) 次のように考えても問題ないでゲソ[*6]。

(1) 継続って何かというと、「スタックの中身」のことでゲソ。
(2) 一部の言語には、「スタックの中身のコピーを取る」機能が用意されているんでゲソ。(そしてこの「コピーを取る処理」のことを「継続の捕捉」と呼ぶんでゲソ)
(3) で、取得したコピーを使うと、スタックの内容を「コピーを取ったときの中身」に変更することができるんでゲソ。(そしてこの「スタックを変更する処理」のことを「継続の起動」と呼ぶんでゲソ)

次のページに図も描いてみたのでそれも参考にして欲しいでゲソ。

で、この継続でゲソが、面白いのはやっぱり (2)(3) の捕捉処理&起動処理を用意してある言語の場合でゲソ。(もちろん捕捉処理がない言語でも (1) はあるわけでゲソが、これだけだと「だから何だ」という感じじゃなイカ? [*7])



「継続の捕捉/起動が出来る = スタックの待避/復帰が出来る」といってもいいので、工夫次第で結構いろん

[*4] まだ納得いかないという人のためにもうちょっと説明すると、昔の論文の中には本当に継続相当の代物を “state” と呼んでるものもあったんでゲソ [10] [11]。ちなみに、この論文を書いた Peter Landin という人は、ML や Haskell の祖先である ISWIM という言語を作ったり、1965 年の時点で IO monad に近いものを提案してたりと [9]、黎明期の関数型言語の立役者のような人でゲソ。

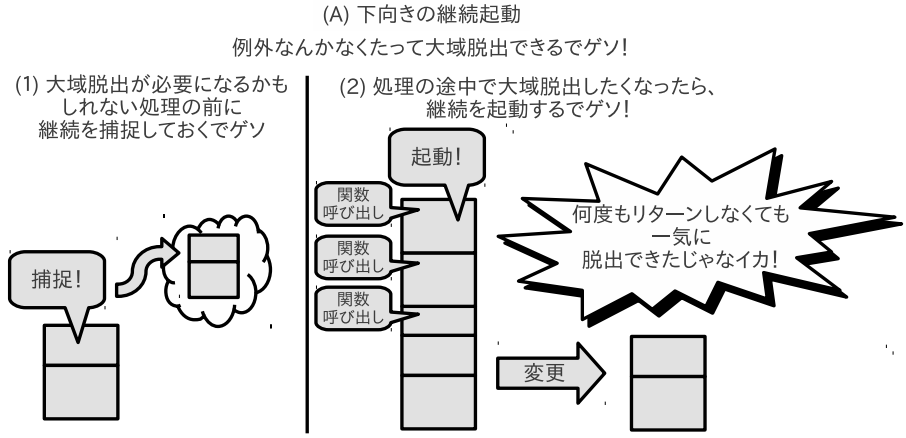
[*5] ところで細かい話でゲソが、レジスタがスタックに待避されていない状態を想定するなら「スタックの中身 + レジスタの中身」ということになるでゲソ。でももう面倒だから、継続を捕捉する処理が実行される際には全てのレジスタはスタックに待避されてる、ということにして話を進めるでゲソ。

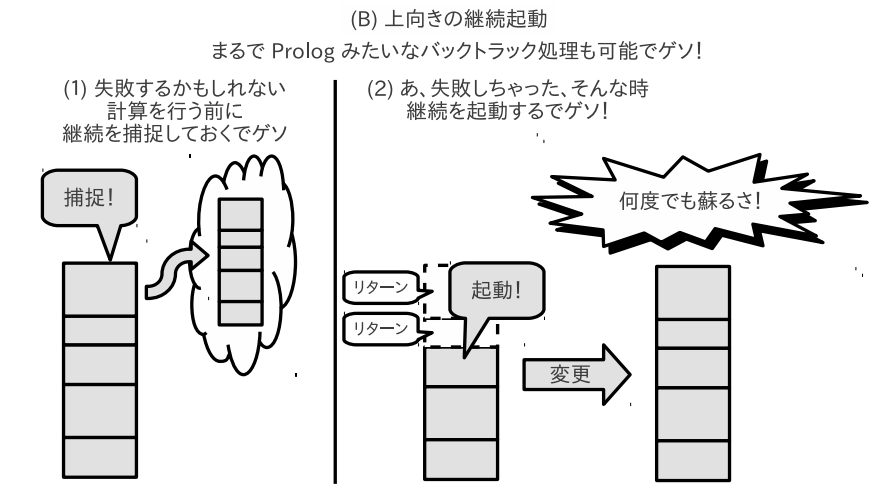
[*6] 実際にはこんな風にはコピーしない実装もいろいろあったりするでゲソが、今回はイメージ重視ということで許して欲しいでゲソ。あと、Scheme みたいに set! とかで環境を破壊的に変更できる場合には、「環境を共有してる他の継続やクロージャーにその変更が見えなきゃ」とか言い出すと単純なコピーだけじゃ駄目なケースがあったりなかったりでゲソが、そういう細かい話も今回は無視するでゲソ。

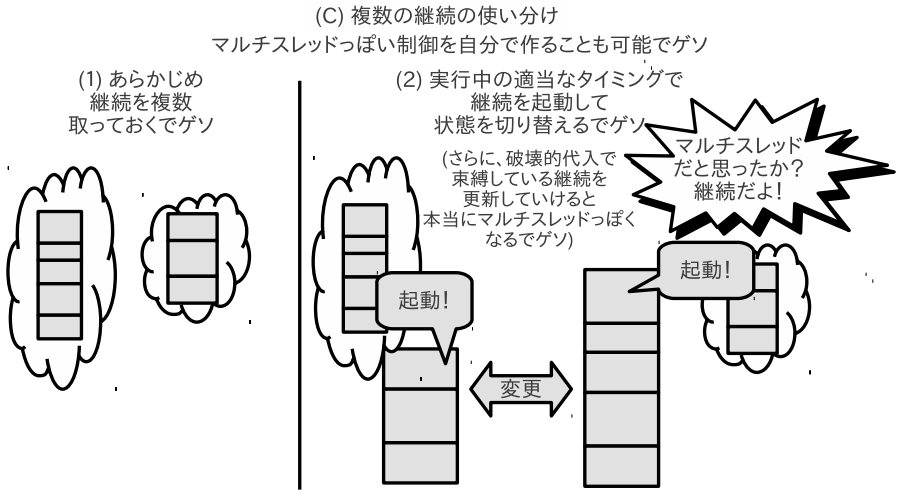
[*7] 実際には (2)(3) がない言語でも「スタックを積まないようにプログラムを書けば捕捉処理なんかそもそも必要ないぜ」っていう「継続渡しスタイル (Continuation-passing style, CPS)」という話があるんでゲソが、それは後で説明するでゲソ。

なことに使えるんでゲソ。例えば次のような使い道が有名でゲソ。

(A) 大域脱出 (例外送出的な処理, setjmp() 的な処理, etc)
(B) バックトラック的な処理 (amb 関数, etc)
(C) ユーザーレベルスレッド (グリーンスレッド, コルーチン, ジェネレータ, ファイバー, etc)



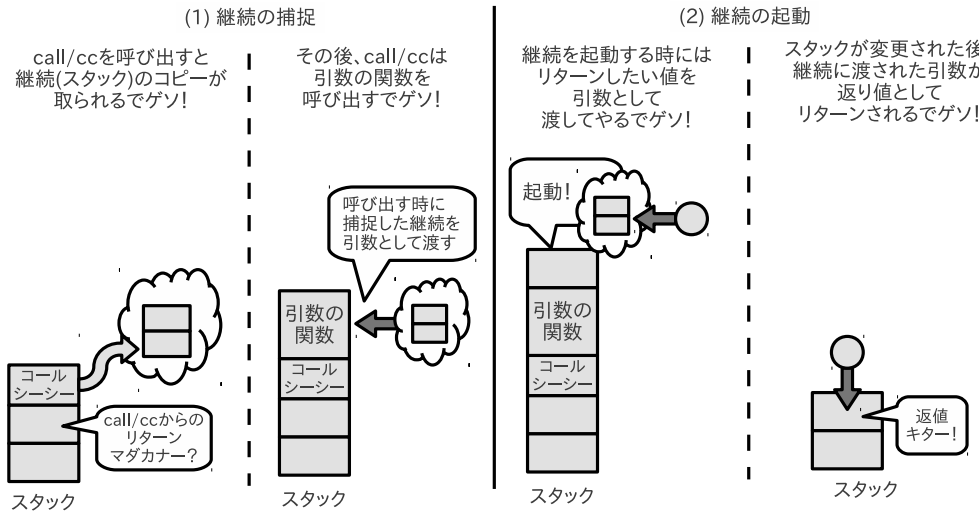




15.1.1 Scheme で「継続の捕捉」を体験してみるでゲソ

さて、何となくイメージが湧いたところで、実際に Scheme で継続の捕捉をしてみようじゃなイカ! Scheme では call/cc [*8] という関数で継続の捕捉が行えるんでゲソ。

最初に call/cc の挙動をちょっとだけ説明しておくでゲソ。下に図があるのでそれを見ながら読んで欲しいでゲソ。まず、call/cc は関数を引数に取るんでゲソ。そして「call/cc が捕捉した継続」がその関数に引数として渡されるんでゲソよ。次に、捕捉した継続を起動したくなったら関数のように引数を適用すればいいんでゲソ[*9]。継続が起動されると、スタックが元の状態に戻された後、継続に渡された引数が返り値として返されるんでゲソ。



じゃあ、実際にプログラムの例を紹介してみるでゲソ。call/cc を使ったプログラムは、例えばこんな感じになるでゲソ。

```
(* (call/cc            ; (1) call/cc が継続を捕捉するんでゲソ。
    (lambda (k)        ; (2) 捕捉した継続を引数として、
      (* 42            ;       この関数が呼び出されるでゲソ。
         (k 4))))      ;--(ここまでが call/cc 呼び出しの式でゲソ)
   (- 0
      (/ 4 2)))
```

さて、この式が評価されるとどうなるか考えてみるでゲソ。

まず、この例で call/cc が捕捉する継続について考えてみようじゃなイカ。1 行目から始まっている (call/cc ...) という式の「継続」ってどういうものなんでゲソね? ところで、上では「継続」=「その後に行われる計算処理」=「スレッドの状態」=「スレッドのスタックの中身」と説明したでゲソが、「スレッドの状態」とか「スレッドのスタックの中身」は文字で書き表すのが面倒でゲソ。というわけで、ここでは主に「その後に行われる計算処理」として考えて、「捕捉される継続」がどういう計算処理をするものなのかを考えてみるでゲソ (「継続」=「スレッドの状態」と考えるなら「そういった処理を実行する直前でのスレッドの状態」、「継続」=「スレッドのスタックの中身」と考えるなら「そういった処理を実行する直前でのスタックの中身」と考えてもらえばいいでゲソ)。

じゃあ「継続」が行う処理 (=「call/cc のリターン後に行う計算処理」) について考えてみると、上のプログラムを素直に見ていると、何となく次のような計算処理という気がしなイカ?

[*8] ちなみにこの関数、正式名称は call-with-current-continuation というんでゲソ。でもあまりに長すぎて、正式名称より略称で呼ばれることの方が多いでゲソ。まるで「モジャ公」みたいなものでゲソね。

[*9] ちなみに何で関数のように引数を適用するのかというと、継続と関数には関連性があるからなんでゲソ。それについては CPS の節で一緒に説明するでゲソ。

```
(* □            ; □と書いている箇所は
   (- 0         ;    「もう既に計算が終わっている箇所」だと思ってほしいでゲソ。
      (/ 4 2))) ;    (ちなみに□の箇所は元々は call/cc 呼び出しの式があった箇所でゲソ)
```

実際これで正解でゲソ。

あるいはこれを関数のようなものと考えると、この後で行われる計算は次のような関数だとも言えるでゲソ。継続を使うときには関数のように引数を適用するので、関数っぽく考えた方が分かりやすいかもしれないでゲソ。というわけで、この先の話ではこの形で考えてみるでゲソ。

```
(lambda (n)
  (* n
     (- 0
        (/ 4 2))))
```

じゃあ継続を捕捉するところまでは分かったので、次に call/cc が引数の関数を呼び出した後について考えてみるでゲソ。引数の関数が呼び出された後の式の形は、だいたいこんな感じでゲソ?

```
(* (* 42        ; この 2 行分が引数の関数の中身でゲソ
      (k 4))    ;    (ちなみに k は call/cc から引数として渡された継続でゲソ)
   (- 0
      (/ 4 2)))
```

ところで、今回の引数の関数は (* 42 (k 4)) という式を計算するんでゲソが、この k は call/cc から渡された継続でゲソ。ということは、ここに出てくる (k 4) という式は、関数の適用じゃなくて「継続を起動する式」でゲソ。

じゃあ次に、この継続 k が起動されるとどうなるかを考えてみるでゲソ。上の図にも描いたように、継続が起動されるとその時点での継続は全て捨てられて、代わりに起動された k で置き換えられるんでゲソ (上の図では、スタックの中身が捨てられて継続の中身に置き換えられる、という感じになっているけど要は同じことでゲソ)。

というわけで、(k 4) という式を評価した瞬間、式全体は次のようになるでゲソ。

```
((lambda (n)                  ;-- ここからが call/cc が捕捉した継続
   (* n                       ;
      (- 0                    ;
         (/ 4 2))))           ;-- ここまでが捕捉した継続
 4)                           ; ここの 4 が継続に適用される引数
```

(k 4) を評価する前はその外側に (* (* 42 □) (- 0 (/ 4 2))) という式があったはずでゲソが、そこの部分は「継続の起動」によって捨てられてるんでゲソ [*10]。

で、さらに評価を進めてみると、式はこんな感じになるでゲソ。

```
-8      ; 評価値 -8 が得られたでゲソ
```

というわけで、最終的な評価値は -8 でゲソ。

さて、何となく「継続の捕捉」ってどんな感じか分かったでゲソ? 始めて見るとなかなかエキセントリックな感じに見えるでゲソが、よくよく見てみるとそんなに変な感じでもない (はず) でゲソよ。

ちなみに、今回の例の結果をプログラムの最初の式と見比べてみると、「継続の起動」によって (* 42 □) という計算を省略して脱出したようにも見えなイカ? 実際、今回の例は上の (A) (B) (C) で言うと (A) の大域脱

[*10] ところで、この挙動についてもう少し考えてみると、継続は「関数みたいなものだけど絶対にリターンしてこないところが違う特殊な関数」という風にも見えなイカ? まるで exit() 関数とか execve() システムコールみたいな感じでゲソ。この「継続 = 絶対にリターンしてこない関数」という話については、CPS の節でもう一度説明するでゲソ。

出に近い使い方でゲソ [*11]。じゃあ call/cc をどう使ったら (B) や (C) が実現できるのか? これについては宿題にするので少し考えてみて欲しいでゲソ。

15.2 継続って誰が考えたんでゲソね?

ところで「継続」なんていう素晴らしい概念、いったい誰が考え出したのか気にならないでゲソ?

今では「継続といえば Scheme」みたいな感じがあるので Scheme 使いが考えたと思ってるかもしれないでゲソが、実は (残念ながら) そうではないんでゲソ。その証拠に Scheme の仕様書である R6RS にも次のような記載があるんでゲソ。

[R6RS Rational 11.9 Control features より]

"Most programming languages incorporate one or more special-purpose escape constructs with names like exit, return, or even goto. In 1965, however, Peter Landin [23] invented a general-purpose escape operator called the J-operator. John Reynolds [28] described a simpler but equally powerful construct in 1972. The catch special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the catch construct could be provided by a procedure instead of by a special syntactic construct, and the name call-with-current-continuation was coined in 1982."

- 引用元: Revised6 Report on the Algorithmic Language Scheme -Rationale-, 2007
  http://www.r6rs.org/final/r6rs-rationale.pdf

[日本語訳]

"ほとんどのプログラミング言語は exit や return、さらには goto といったような名前の目的特化の脱出構文をひとつ以上組み込んでいる。しかしながら、1965 年、Peter Landin [23] が J 演算子と呼ばれる汎用の脱出演算子を発明した。1972 年、John Reynolds [28] は、より単純で、同様に強力な構文について述べた。1975 年に Scheme についての報告書で Sussman と Steele によって説明された catch 特殊形式は、名前こそ MacLisp のより一般性のない構文に由来したものであったが、Reynolds の述べた構文とまったく同じであった。複数の Scheme 実装者は、catch 構文の能力のすべては特別な構文構造ではなく手続きとして提供できることを指摘し、1982 年に call-with-current-continuation という名前が作られた。"

- 引用元: R6RS:翻訳:Rationale:11.9 Control features, 2008
  http://practical-scheme.net/wiliki/wiliki.cgi?R6RS%3A%E7%BF%BB%E8%A8%B3%3ARationale%3A11.9%20Control%20features

この通り、継続は Scheme が誕生する以前から研究されてたんでゲソよ。じゃあ継続っていったいどこからきたのか不思議じゃなイカ?

実は、継続は手続き型言語の (特に Algol 60 の) goto の研究から生まれてきたんでゲソ。今では「継続」っていうと関数型言語の世界で聞くことが多いから、ちょっと意外でゲソね。

この「継続」の概念がどんな感じで発展してきたかは、上の R6RS の文中でも触れられている John Reynolds 先生が "The Discoveries of Continuations" [12] という論文にまとめているでゲソ [*12]。詳細はこの論文を読んでもらうとして、ここでは簡単に表示的意味論 (denotational semantics) [*13]にまつわる goto と継続の関わり

[*11] といっても、今回は 1 段しか脱出してないので C 言語系列で言うところの return 程度の脱出にしかなってないでゲソが…

[*12] 日本語サイトだと Gauche の作者である川合史郎さんのページに分かりやすいまとめが載ってるでゲソ。
`http://blog.practical-scheme.net/shiro/20120122-origin-of-continuations`

[*13] 「ヒョージテキ・イミロン」というのは、「IT 系の人間はバズワードばっかで何言ってるのかワケわかんねーお。この前も、英語の次はジャバ語、っていうからジャワ島の言葉だと思って知ったかぶったら違ってたんだお。もう全部数学の言葉でしゃべって欲しいんだお」という数学者の期待に応えて、プログラミング言語の全ての概念を数学の概念だけで説明し尽くそうという研究分野でゲソ。「ソーサテキ・イミロン」や「コーリテキ・イミロン」と並んで、プログラミング言語の「意味」を考えようという

を紹介してみようと思うでゲソ [17]。ちなみに、なんで表示的意味論の話かというと、後で説明するように Scheme 的に少し意味があるからでゲソ! ただし、この話ちょっと面倒なので、時間が無い人は飛ばして次の節に進んでもらっても問題ないでゲソ。

15.2.1 手続き型言語を数学的に説明しちゃうでゲソ

ではさっそく表示的意味論の話に移るでゲソ。手続き型言語の場合にどうやって数学的な概念で説明するか (= どうやって表示的意味論を付けるか) というと、手続き型の本質的な内容は「プログラムの状態の変更」(もっとぶっちゃけるとメモリとかテープとかそういった類いのものの書き換え) と言えるので、プログラミング言語の「文」を「古い状態を受け取って新しい状態を返す関数」として説明するのが一般的でゲソ。この「状態」については、ほんとにメモリの内容のことだと思ってくれてもいいし、もう少し一般化して「何らかの場所を示す値 (変数名とかアドレスとか何でもいいでゲソ) から値への写像」なんだと思ってくれてもいいでゲソ。

じゃあまず goto がない場合について考えてみるでゲソ。 goto がない場合の手続き型言語の代入文は次のような関数として説明できるでゲソ。

元の代入文:

```
X = a
```

対応する数学的な関数:**(数学記号を使うのが面倒なのでちょっと** `Haskell` **っぽく書くでゲソ)**

(`\s -> s[n/X]`**)　(ただし、**`n`** は** `a` **という式の評価値に対応する数学的な要素でゲソ)**

いきなり勝手に記号を使ったんで読み方が分からない気もするでゲソが、要は「状態 s を受け取って s[n/X] という状態を返す関数」だと思って欲しいでゲソ。で、s[n/X] というのは「X に対応する箇所の値は n で、それ以外の箇所については s と同じ」という状態だと考えて欲しいでゲソ。実際、代入文は対応する変数の箇所を書き換えてそれ以外は書き換えないんだからこれで説明が付きそうな気がしなイカ? (ところで、n については上で書いたとおり「a という式の評価値に対応する数学的な要素」でゲソが、これを具体的にどういう数学要素で説明するかは今回はあんまり重要じゃないので、特に考えないで進もうと思うでゲソ[*14]。)

次に、(goto がない場合の) 複数の文からなる逐次実行文について考えてみるでゲソ。これは、関数合成として説明できるでゲソ。要は、それぞれの文に対応する関数を合成するだけでゲソね。実際、先の文によって状態が変更されたあとで後の文によってさらに状態が変更されるんだから、関数の合成 (先の関数の返値を次の関数の引数にする) とそっくりでゲソ?

元の逐次実行文:

```
S1; S2
```

対応する数学的な関数:**(数学記号を使うのが面倒なのでちょっと** `Haskell` **っぽく書くでゲソ)**

(`S2` **に対応する関数)** `.` **(**`S1` **に対応する関数)　('.(ドット)' は関数合成演算子でゲソ)**

だから、例えば次のようなプログラムはこんな感じで説明できるんでゲソ。

元々のプログラム:

```
X = 1;
Y = 2;
Z = 3;
```

対応する数学的な関数:

```
(\s -> s[3/Z]) . (\s -> s[2/Y]) . (\s -> s[1/X])
= (\s -> s[1/X][2/Y][3/Z])
```

これも勝手に記号を使っちゃってるでゲソが、s[1/X][2/Y][3/Z] というのは ((s[1/X])[2/Y])[3/Z] の略だと考えて欲しいでゲソ。つまり、「Z の箇所は 3、Y の箇所は 2、X の箇所は 1、それ以外は s と同じ」という状態

分野の~~3 バカトリオ~~ 3 大巨頭の 1 つでゲソね。

[*14] きっちりやりたい人は表示的意味論の教科書を見るといいと思うでゲソ (マルナゲ!)

でゲソ [*15]。

念のため、この関数でちゃんと元のプログラムが説明できているのかどうか考えてみるでゲソ。元々のプログラムを実行すると、変数 X の値は 1、変数 Y の値は 2、変数 Z の値は 3 になり、それ以外の変数はそのまま、でゲソね。一方対応する関数の方はというと、状態 s を受け取って「Z の箇所は 3、Y の箇所は 2、X の箇所は 1、それ以外は s と同じ」という状態を返しているでゲソ。というわけで、確かにこのプログラムの意味がちゃんと関数で表現できているじゃなイカ!

じゃあ次に、ここに goto があったらどうなるか考えてみるでゲソ。ただし goto に対応する関数はまだ決めてないので、とりあえず (“goto L1” に対応する関数) という風な書き方をしてみるでゲソ。

元々のプログラム:

```
      goto L1;  /* L1 ラベルまで goto しちゃうでゲソ */
      Y = 2;    /* この文は goto で飛ばされるので実行されないでゲソ */
  L1: Z = 3;
```

対応する数学的な関数??:

```
(\s -> s[3/Z]) . (\s -> s[2/Y]) . ("goto L1"に対応する関数)
```

さて、この関数がちゃんと対応しているのかどうかを考えてみるでゲソ。元々のプログラムを実行すると、変数 Z の値は 3、それ以外の変数はそのまま、でゲソね。じゃあ、対応する関数の方はというと、(“goto L1” に対応する関数) のところがはっきりしないでゲソが、`(\s -> s[3/Z])` と `(\s -> s[2/Y])` を関数合成しちゃってるので、少なくとも Z の箇所は 3、Y の箇所は 2 になった状態が返されるはずでゲソ? だけど、“Y = 2” の文は goto で飛ばされて実行されないはずだから、この効果まで関数の方に入ってしまってるのはまずいじゃなイカ!

15.2.2 どうやったら goto に対応できるんでゲソ?

というわけで、単純な拡張では goto は説明できなさそうなことが分かったところで、何でうまくいかないのかを少し考えてみようと思うでゲソ。

そもそも以前のアプローチは何をしてたかというと、「1 つ 1 つの文に対してそれぞれ対応する関数が見つかるはず」という考えで関数を見つけようとしてたでゲソ。実際、goto がない場合には 1 つ 1 つの文にちゃんと対応する関数があったので、それらを見つけて関数合成していくことでプログラム全体に対応する関数が作れたんでゲソ。

だけどこのアプローチって goto 文にも通用するんでゲソか?

というのも、goto 文ってそれ単体で意味が付くのかどうかちょっと怪しくなイカ? 単体で意味が付くってことは、他の文とは無関係に、状態に (=メモリの内容とかに) どう影響を及ぼすかがはっきり言えるということでゲソ。だけど goto 文はどちらかというと「制御フローを変える (=後続の処理を変える) ことで間接的に状態に影響を与える」という感じでゲソ。ということは、goto 文については (単体では意味は付けようがなくて) 飛び先とセットで考えないと駄目なんじゃなイカ?

というわけで、ここで求められるのは発想の逆転でゲソ。「逆に考えるんだ。「1 つ 1 つの文に対して対応する関数が見つからなくってもいいさ」と考えるんだ」ということでゲソね。

で、ここまで説明してようやく「継続」の出番でゲソ。1 つ 1 つの個別の文に対して関数が見つからなくても、「その文から始まってプログラム終了までに実行される全ての文の列」に対してならどうでゲソ? 言い換えると、文単体じゃなくて「文 + その文の継続」というセットで考えればいいんじゃなイカ、というのが新しいアプローチでゲソ。

15.2.3 新しいアプローチでもう一回説明し直してみるでゲソ

じゃあ具体的に「セットで考える」というのがどういうことか、例を出しながら説明してみるでゲソ。で、最初にまず注意が必要なんでゲソが、上では「文」を「古い状態を受け取って新しい状態を返す関数」として説明したじゃなイカ? で、これは「その文の実行直前の状態→その文の実行直後の状態」という関数だったでゲソ。でも、今度は「文 + その文の継続」のセットで考えるので、「その文の実行直前の状態→プログラ

[*15] 正確に言うなら、「Z に対応する箇所の値は 3 で、それ以外の箇所については「Y に対応する箇所の値は 2 で、それ以外の箇所については「X に対応する箇所の値は 1 で、それ以外の箇所については s と同じ」」」ということでゲソ。

ム終了時の状態」という関数として説明することになるんでゲソ (正確に言うと、これもちょっと違うので正しい形を後でもう一度紹介するでゲソ。とりあえずはこういうことにして先に進んで欲しいでゲソ)。

じゃあ、もう一度代入文について考えてみるでゲソ。

元の代入文:

```
X = a
```

対応する数学的な関数:**(数学記号を使うのが面倒なのでちょっと Haskell っぽく書くでゲソ)**

```
(\k  ->  k . (\s -> s[n/X]))
```

何やら記号が増えて読みづらくなってるでゲソね。でもやってること自体はそんなに難しくないでゲソ。

上で説明した「セットで考える」というのは、(この代入文だけじゃなく) その後でプログラムが終了するまでに実行される全ての文も一緒に考えないとイカん、ということでゲソ。とはいえ、この代入文の後にどんな文が実行されるかなんて、この代入文だけからでは分からないでゲソね。というわけで、分からないところは未知数として残すということで、上の関数だと k という引数でそれを表しているんでゲソ [*16]。

で、この k、具体的には何かと言うと「この代入文の次の文からプログラム終了時までに実行される全ての文」に対応する関数でゲソ。言い換えると、「この代入文の次の文の実行直前での状態→プログラム終了時の状態」という関数でゲソ。

さて、元々の目的はこの代入文を関数として説明することだったでゲソ。そのために今作りたいのは「この代入文の実行直前の状態→プログラム終了時の状態」という関数でゲソ。で、上のような k が引数として与えられたとすると、これを作るのは簡単じゃなイカ?

具体的にどうするかと言うと、まず "`(\s -> s[n/X])`" という関数を作ると、(これは上で説明したとおり)「この代入文の実行直前の状態→この代入文の実行直後の状態」という関数になってるでゲソ。で、この関数と上の k を関数合成すると、「この代入文の実行直前の状態→この代入文の実行直後の状態」と「この代入文の次の文の実行直前での状態→プログラム終了時の状態」の合成と言うことになるでゲソ? これって、「この代入文の実行直前の状態→プログラム終了時の状態」という関数のことじゃなイカ! 実際に上で紹介した数学的な関数の定義を見てみると、この2つの関数を合成しているだけでゲソ。

というわけで、代入文についてはちゃんと説明できそうじゃなイカ。ただし、説明に使用する関数の形は以前とは少し変わってるでゲソね? 具体的に言うと、こんな感じになってるでゲソ。

[以前のアプローチ]

その文の実行直前の状態→
その文の実行直後の状態

[今回のアプローチ]

(次の文の実行直前での状態→プログラム終了時の状態) →
その文の実行直前の状態→
プログラム終了時の状態

これだけ見るとなんだか無駄に複雑になっているように見えるでゲソ... でも全ての文をこの形の関数で説明すると、goto 文にも上手く説明が付くんでゲソ。まぁとりあえずは次に進もうじゃなイカ。

じゃあ次に、逐次実行文について考えてみるでゲソ。

元の逐次実行文:

```
S1; S2
```

対応する数学的な関数:**(数学記号を使うのが面倒なのでちょっと Haskell っぽく書くでゲソ)**

(\k -> S1 に対応する関数 (S2 に対応する関数 k))

逐次文の場合も、「この逐次文から始まってプログラム終了時までの文全体」を考えないとイカんというこ

[*16] 継続について詳しい人は「継続渡しスタイル (Continuation-passing style, CPS)」だと思ってくれればいいでゲソ。CPS については後で説明するでゲソ。

とになるでゲソ。といっても、この場合も逐次文の後にどんな文が実行されるかなんて分からないでゲソ。なので、分からない部分を k という引数として受け取る、という形にしているのは代入文の場合と同じでゲソ。

じゃあ代入文の時と同じように、対応する関数についてもう少しよく考えてみるでゲソ。今欲しかったのは、上で説明したように「(この逐次文全体の次の文の実行直前での状態→プログラム終了時の状態) →この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数でゲソ。

で、まずこの逐次文の要素のうち S2 文の方について考えてみるでゲソ。上と同じように考えると、S2 文に対応する関数は「(S2 の次の文の実行直前での状態→プログラム終了時の状態) → S2 の実行直前の状態→プログラム終了時の状態」ということになるでゲソ?

ところで、「S2 の次の文」って「この逐次文全体の次の文」のことじゃなイカ?

ということは、S2 に対応する関数の第一引数である「(S2 の次の文の実行直前での状態→プログラム終了時の状態)」というのは、引数の k のことでゲソ! ということで、S2 に対応する関数に k を適用してやれば、「S2 の実行直前の状態→プログラム終了時の状態」という関数が作れるでゲソ (カリー化してると考えて欲しいでゲソ)。

次に S1 文の方について考えてみるでゲソ。 S1 文に対応する関数も上と同じで「(S1 の次の文の実行直前での状態→プログラム終了時の状態) → S1 の実行直前の状態→プログラム終了時の状態」ということになるでゲソ。

ところで、「S1 の次の文」って S2 でゲソね?

ということは、この関数の第一引数である「(S1 の次の文の実行直前での状態→プログラム終了時の状態)」って、さっき作った「S2 の実行直前の状態→プログラム終了時の状態」のことじゃなイカ。なので、さっきと同じように S1 に対応する関数に “(S2 に対応する関数 k)” を適用してやれば、「S1 の実行直前の状態→プログラム終了時の状態」という関数が作れるでゲソ!

で、そもそも何が作りたかったのかをもう一度考えてみると、作りたかったのは「(この逐次文全体の次の文の実行直前での状態→プログラム終了時の状態) →この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数だったでゲソ? ところで、「この逐次文全体の実行直前の状態」って「S1 の実行直前の状態」と同じ意味じゃなイカ?

ということは、「この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数は、さっき作った “(S1 に対応する関数 (S2 に対応する関数 k))” のことでゲソ。

というわけで、引数 k を受け取って (S1 に対応する関数 (S2 に対応する関数 k)) を返せば目的の関数になるでゲソ! (実際、上で紹介した数学的な関数の定義はその通りになっているでゲソね)

うーん、なんだか代入文や逐次実行文は新しい方式でも上手く説明できそうでゲソね。

それでは、最後にいよいよ goto 文について考えてみるでゲソ。 goto 文についても同じように「(goto 文の次の文の実行直前での状態→プログラム終了時の状態) → goto 文の実行直前の状態→プログラム終了時の状態」という関数として説明できるでゲソか?

元の `goto` 文:

```
goto L1;
```

対応する数学的な関数:(数学記号を使うのが面倒なのでちょっと `Haskell` っぽく書くでゲソ)

`(\k -> k1)` **(ただし、`k1` は「`L1` ラベルが付いている文 + その継続」に対応する関数)**

さて、goto 文は (予想に反して (?)) なんだか結構シンプルな関数になってるでゲソね。

じゃあ、まずは第 1 引数である「(goto 文の次の文の実行直前での状態→プログラム終了時の状態)」について考えてみるでゲソ。

ところでこの「次の文」の扱いなんでゲソが、goto 文の場合はこれまで考えた代入文や逐次実行文と少し違うでゲソね? 代入文や逐次実行文は、実行が終わったらそのまま次の文に制御が移ってたじゃなイカ。でも、goto 文って (引数のラベルに飛んでっちゃうから) 次の文とか関係ないでゲソ。

というわけで、goto 文の場合は第 1 引数の「(goto 文の次の文の実行直前での状態→プログラム終了時の状態)」については全く使う必要がないんでゲソ (というか、むしろ使っちゃ駄目というか)。実際、上の数学的な関数の中でも引数の k は使われてないでゲソね。そしてこれが以前のアプローチとの違いなんでゲソ。つ

まり、以前のアプローチだと問答無用で「次の文」と関数合成されちゃってたんでゲソが、今回のアプローチだと「次の文」が重要じゃなければその効果を勝手に捨てちゃうことができるんでゲソ。

じゃあ次に、残りの「goto 文の実行直前の状態→プログラム終了時の状態」の部分を考えてみるでゲソ。といっても、goto 文自体はプログラムの状態を変更したりはしないじゃなイカ? goto 文が実行されたら飛び先の文にジャンプして、後はそこから普通に実行が続けられるだけでゲソ。ということは、この「goto 文の実行直前の状態→プログラム終了時の状態」って「goto 文の飛び先の文の実行直前の状態→プログラム終了時の状態」と同じでないと変でゲソ?

というわけで、この話をまとめてみると、(引数 k を受け取るんだけど、この k についてはさっぱり無視して)「飛び先の文 + その継続」に対応する関数を返せばいい、ということでゲソね。これが上で紹介した数学的な関数の箇所に書いてある内容でゲソ。

どうでゲソ? たしかに goto 文の挙動が説明できていそうな感じじゃなイカ? じゃあ、この新しいアプローチでちゃんとプログラムが説明できるのか確かめてみるでゲソ!

例 1: 以前のアプローチでも上手くいっていた例

元々のプログラム:

```
X = 1;
Y = 2;
Z = 3;
```

対応する数学的な関数:

```
(\k ->                                  # この逐次文全体の継続 k を受け取る
     (\k1 -> k1 . (\s -> s[1/X]))         #（これが 1 行目に対応する関数）
        (\k2 -> k2 . (\s -> s[2/Y]))      #（これが 2 行目〃）
           ((\k3 -> k3 . (\s -> s[3/Z])) #（〃 3 行目〃）
              k))                         # 全体の継続である k を適用
= (\k -> \s ->  k (s[1/X][2/Y][3/Z]))  # 整理してまとめるとこんな感じ
```

例 2: 以前のアプローチでは上手くいかなかった例

元々のプログラム:

```
     goto L1;  /* L1 ラベルまで goto する，という意味でゲソ */
     Y = 2;    /* この文は goto で飛ばされるので実行されないでゲソ */
L1:  Z = 3;
```

対応する数学的な関数:

```
(\k ->                                  # この逐次文全体の継続 k を受け取る
     (\k1 ->                              #（これが 1 行目に対応する関数）
          ((\k3 -> k3 . (\s -> s[3/Z]))  #    飛び先である 3 行目に対応する関数と
               k))                       #    飛び先の継続である k を
                                         #    合成したものを返す
        (\k2 -> k2 . (\s -> s[2/Y]))     #（これが 2 行目〃）
           ((\k3 -> k3 . (\s -> s[3/Z])) #（〃 3 行目〃）
              k))                         # 全体の継続である k を適用
= (\k -> \s ->  k (s[3/Z]))             # 整理してまとめるとこんな感じ
```

まずは元々上手くいっていた例について、もう一度見ておくでゲソ (もちろんさっき上手くいったからもう考えなくてもいいんでゲソが、念のため新しいアプローチでも大丈夫かどうかを確認しておくでゲソ)。ちょっと注意が必要なのは、さっきと違って、このプログラムだけじゃなく「このプログラム + その継続」というセットで考えるんでゲソ。(つまり、この例では3つの文しかないでゲソが、この後ろにも色々と処理

があるんだと思って考えて欲しいでゲソ。[*17])

元々のプログラムを実行した時に起こることを (その継続まで含めて) 考えると、まず上の 3 行によって「変数 X の値は 1、変数 Y の値は 2、変数 Z の値は 3、それ以外の変数はそのまま」という状態になるでゲソね。そしてその後、このプログラムの「継続」によってさらに状態が変更されるでゲソ。

一方対応する関数の方はというと、継続 k と状態 s を受け取って「Z の箇所は 3、Y の箇所は 2、X の箇所は 1、それ以外は s と同じ」という状態を作り、それを k に適用した結果を返すでゲソ。というわけで、やっぱりこのプログラムの意味がちゃんと関数で表現できているじゃなイカ!

じゃあ次に、前は上手くいかなかった例について考えてみるでゲソ。今度もさっきと同じように「このプログラム + その継続」というセットで考えてみるでゲソよ。

元々のプログラムを実行すると、上の 3 行によって「変数 Z の値は 3、それ以外の変数はそのまま」という状態になり、その後でさらにこのプログラムの「継続」によって状態が変更されるでゲソ。

じゃあ対応する関数の方はというと、継続 k と状態 s を受け取って「Z の箇所は 3、それ以外は s と同じ」という状態を作り、それを k に適用した結果を返すでゲソ。うむ、これはまさしく上のプログラムの挙動そのものじゃなイカ!

というわけで、「継続」という考え方を入れてみると goto 文があってもプログラムの意味をちゃんと関数で説明できるということが分かったでゲソ? 今回紹介したのは簡単な例でゲソが、if とか while、あるいは関数みたいな制御構造と組み合わせても同じやり方でちゃんと goto 文が説明できるんでゲソ。気になる人は、上の論文を見ながら自分で色々な例を確かめてみて欲しいでゲソ。

15.2.4 最後に Scheme との関係を話しておくでゲソ

さて、長々と説明してきたんでゲソが、このちょっと面倒な表示的意味論の話をしたのは Scheme 的には 1 つ理由があるんでゲソ。 Scheme の意味論も R5RS(1 世代前の言語仕様) [2] の頃は表示的意味論で定義されてたんでゲソが[*18]、そのベースになっていたのがこの Algol 用に作られた「継続」を使ったやり方なんでゲソ。ということは、この表示的意味論の読み方を知っていれば Scheme の意味についてはバッチリ [*19]、ということでゲソ! 嘘だと思ったら R5RS に書かれているルールとこの論文のルールを見比べてみるでゲソ。if 式だとか nop だとか逐次実行みたいに、(表記法はちょっと違うでゲソが内容としては) ほとんど丸写しになっている箇所まであるでゲソよ。

[*17] あるいは、この後ろに本当に処理がない場合なら NOP(何もしない命令) が継続だと考えてくれればいいでゲソ。その場合、数学的な関数の方では「恒等関数 (\x -> x) が継続」ということに対応するでゲソ。

[*18] ただし現行仕様である R6RS になった時に操作的意味論による定義に変更されてしまったでゲソが...

[*19] 上に書いたように「R5RS については」でゲソが...

付録 : Algol 60 と愉快な goto たち

上で簡単に触れた通り、継続の研究に大きな影響を与えたのは Algol 60 なんでゲソが、これは Algol 60 の goto が特段ややこしかったからという理由もあるんでゲソ。 Algol 60 は lexical scope を導入した言語としても有名でゲソが、さらに関数の中で関数を定義できる (nested function) という機能も持っていたんでゲソ。
え、「それがどうしたんでゲソ? 」って?
問題になったのは goto が使用する「飛び先のラベル」も lexical scope を持ってたってことでゲソ。つまり、「内側の関数から外側の関数内のラベルに goto する」という芸当が Algol 60 ではできたんでゲソ!
少し例を書いてみるでゲソ。

```
procedure S;
begin
  procedure T;
  begin
    integer procedure U;
    begin
      ...
      goto L1     # (3) U の中からラベル L1 まで 2 段階の大域脱出でゲソ!
    end U;
    integer a;
    a := U        # (2) T の中で、U が呼び出されるでゲソ。
  end T;
  T;              # (1) 関数 S の中で、まず T が呼ばれるでゲソ。
L1:               # (4) 関数 S よ! 私は帰ってきた!!
                  #     (ってそれは goto じゃなくてガトー (Gato) さんでゲソ)
end S;
```

さぁ、何が起こるか分かるでゲソ? この場合、関数呼び出し 2 つ分の大域脱出になるので、当然スタックフレームを 2 つ破棄しないといけないんでゲソ。というわけで、goto なのにスタックの中身も一緒に考えないと説明できなくなったんでゲソ。
実際問題として、こんな面倒くさい挙動がなければ、プログラムを goto のない部分ごとに分割して (コンパイラ用語で言うところの基本ブロックでゲソね) それぞれに対して対応する関数を見つけてうまくつなぎ合わせる、という方法も取れたんでゲソ。だけど、こんな風に関数の外側に向けて goto されたりすると流石に基本ブロックとかでは説明が付かないので、もうちょっと手の込んだ解決策が必要になったんでゲソ。これが「goto って結局何なんだ? 」という話になって、継続の研究につながったそうでゲソ。
ちなみに、Algol 60 は後で説明するように Scheme にも大きな影響を及ぼしたんでゲソ。いろんな意味ですごい言語だったんでゲソね。

15.3　なんで Scheme と継続が関係あるのか不思議じゃなイカ?

前の節で説明したように、元々は「継続」は Algol とか goto に絡んだ話だったんでゲソ。それがなんで Scheme と結びついたのか不思議じゃなイカ?
実はこれ、Scheme の生い立ちと関係があるんでゲソ。というわけで、少し Scheme の誕生のいきさつを話

しておこうと思うでゲソ [*20]。

Scheme が誕生したのは 1970 年代の~~まちゃちゅーちぇっちゅちゅー~~マサチューセッツ工科大学 (MIT) でゲソ。当時の MIT は人工知能研究のメッカで、人工知能の研究用にいろんなプログラミング言語が作られてたんでゲソ。特に人工知能にはどんな制御構造がいいのか、という話が盛んで (例えば後の Prolog で使われるバックトラックみたいなものも含めて) いろんな制御構造が研究されていたそうでゲソ。

そんなある日、当時生まれたばかりの Simula や Smalltalk といったオブジェクト指向言語を見ていた MIT の Carl Hewitt 先生は、アクターモデル (Actor model) という新しいプログラミング・パラダイムを思いつくんでゲソ。このアクターモデル、オブジェクト指向言語から思いついたので基本的にはオブジェクト指向に似た感じのパラダイム [*21] なんでゲソが、特徴としてすごく画期的な制御構造を持っていたんでゲソ。

こう言われると、アクターモデルの制御構造が気になってこなイカ? 実はアクターモデルの制御構造は基本的に 1 つしかなくて、アクターからアクターにメッセージを送信する (メッセージパッシングする)、というものだったんでゲソ!

えっ、あんまり画期的な感じがしない? というか、普通のオブジェクト指向言語における「メソッド呼び出し」のことじゃなイカって?

うむ、確かにメッセージパッシングは少し「メソッド呼び出し」と似てるでゲソ。ただし、この 2 つには大きく違うところがあるんでゲソ。それは、メッセージパッシングでは「呼び出し元に値をリターンする」という概念がないことでゲソ。つまり、アクターでは「メッセージを送る」という制御しかなくて、(基本的には向こうからは返ってこない) 一方通行の制御フローになるんでゲソ。

うん? なんだか画期的というよりは使い勝手が悪そうでゲソって?

確かにこの説明だけ聞くと、ちょっと柔軟性に欠けそうな感じがするかもしれないでゲソ。一応、それぞれのアクターにはインスタンスフィールドみたいなもの [*22] もあって、他のアクターへの参照を入れておけばそのアクターにはメッセージを送れるでゲソ。だから、お互いに参照を持ち合っていればメッセージを往復させたりするのは簡単でゲソ。

だけど、あらかじめ決まった相手にしかメッセージを送れないとなるとなんだか不便そうでゲソね。例えば「数学関数を実装したオブジェクト」みたいなものを作りたくなったらどうするでゲソ? 普通のオブジェクト指向言語なら「誰からでも呼び出せて数学関数の処理結果を返り値として受け取れる汎用的な数学処理オブジェクト」を作ることができるでゲソ。こんな風な機能をアクターで実現したくなったらどうすればいいんでゲソ?

でも心配はいらないんでゲソ。この制御パラダイムが画期的だったのは、あるアイデアを使うことで (こういったメソッド呼び出しどころか、もっと複雑な制御まで含めて) いろんな制御を全て実現できたからなんでゲソ。じゃあそのアイデアが気になるでゲソね? 実はそれは「メッセージ内に計算結果の受け取り先になるアクターも一緒に入れておけばいいじゃなイカ!」というものなんでゲソ。

例がないとちょっと分かりにくいので、アクターモデルで作られた cons セル (コンスセル) を紹介してみるでゲソ [*23]。Carl Hewitt 先生がアクターの実証用に作った “Plasma” という言語で書くと、こんな感じになるみたいでゲソ (正確には、いい例が見当たらなかったので適当に Plasma の論文 [7] を見ながら作ってみた例でゲソ。文法的なレベルでかなり間違ってそうな気がするので、雰囲気だけ感じとって欲しいでゲソ)。

```
[CONS ≡
        ; (1) 次の行は CONS アクターが受け取るメッセージを示しているでゲソ
        ;     (パターンマッチしてるんでゲソが、要は A,B,C の3要素を受け取るんでゲソ)
  (≡≡>  (request: [=A =B] (reply-to: =C))
```

[*20] この話を詳しく知りたい人は、bit 誌に掲載されていた “Scheme 過去◇現在◇未来” という記事がお勧めでゲソ。“Scheme” という名前の由来なんかも分かるでゲソよ。

[*21] 例えば、アクターモデルでは (オブジェクト指向の「オブジェクト」に相当する)「アクター」という操作対象を持つんでゲソ。そして、全てのモノをアクターとして説明するんでゲソ。オブジェクト指向でいうところの「全てのモノはオブジェクト」みたいな感じでゲソ。

[*22] アクター用語では “acquaintances(知り合い)” というようでゲソ。

[*23] ところで cons セルって Lisper じゃない人にどのくらい知名度があるのか若干気になるでゲソ。知らない人は普通に 2 要素のタプルだと考えてもらえばいいでゲソ。ちなみに、歴史的な経緯により、1 番目の要素を取り出す関数を car(カー)、2 番目の要素を取り出す関数を cdr(クダー) と呼ぶんでゲソ。

```
        ; (2)CONS アクターは出来た結果を C に送信するんでゲソ。
        ;     この下の (CASE ..) の部分が渡すメッセージでゲソ。
        ;     (ちなみに、この (CASE ..) 自体も
        ;      新しいアクター (cons セルアクター) でゲソ)
    C <==
      (CASES

        ; (3) cons セルアクターは、CAR というメッセージを
        ;     (K というアクターと一緒に) 受け取ると
        ;     K に 1 番目の要素 (つまり A) を送信するでゲソ。
        (==>  (request: CAR (reply-to: =K))
              K <== (reply: A))

        ; (3) cons セルアクターは、CDR というメッセージを
        ;     (K というアクターと一緒に) 受け取ると
        ;     K に 2 番目の要素 (つまり B) を送信するでゲソ。
        (==>  (request: CDR (reply-to: =K))
              K <== (reply: B))

        ; (3) cons セルアクターは、LIST?というメッセージを
        ;     (K というアクターと一緒に) 受け取ると
        ;     K に「そうです (YES)」と送信するでゲソ
        (==>  (request: LIST? (reply-to: =K))
              K <== (reply: YES))))]
```

慣れない文法のソースコードは読みにくいでゲソね。でも何となく意味は分かったりしないでゲソ? "==>" というのがメッセージ受信時の処理を、"<==" というのがメッセージ送信を表してるんでゲソ。

さて、格好いい (?) 文法とかはさておくとして、ここで重要なのは C とか K なんでゲソ。CONS アクターも cons セルアクターも、メッセージの一部として結果の受け取り先となるアクター (C とか K) を受け取るんでゲソ。こんな風に受け取り先を指定できるようになってるので、どのアクターからでも CONS アクターや cons セルアクターが利用できるんでゲソ。ところでこの C とか K って、CONS アクターや cons セルアクターの立場から見ると「自分が渡した値を使ってこの後の計算処理を行うアクター」ということにならなイカ?「この後の計算処理」つまり「継続」でゲソ。

実際、上の論文中でもこの C とか K は「継続」って呼ばれてるんでゲソ。つまりアクターモデルでは、メッセージパッシングという一方通行な制御だけで関数的な制御を実現するために、「継続」を渡す、というテクニックを使ってるんでゲソ。で、メッセージを送られた側では計算結果をリターンする代わりに「継続」に結果を送信するんでゲソ。

初めて見た人にはかなり変わった制御のように見えるかもしれないでゲソ。だけど、Carl Hewitt 先生はこのメッセージパッシングと継続という構造だけでいろいろな制御構造が作れる、ということを次々に発見していくんでゲソ [7]。

で、このアクターモデルに興味を持ったのが Gerald Sussman 先生と Guy Steele 先生。Scheme の生みの親になる 2 人でゲソ。

ところが、当時のアクターモデルは初心者にはなかなか厳しいものだったようでゲソ。というのもさっきの Plasma 言語が複雑すぎて、アクターの何がそんなにすごいのか、2 人にもよく分からなかったそうでゲソ (さっきの例だけ見ても、Plasma が結構複雑な言語だというのは理解してもらえるんじゃなイカ?)。そこで 2 人は「自分たちでアクター用の言語を 1 つ作ってみればアクターの本質が分かるんじゃなイカ」と思いついたんでゲソ、、、って勘のいい人ならここら辺で気づくかもしれないでゲソが、この言語こそが Scheme なん

でゲソ。つまり、Scheme は元々はアクターモデルを勉強するための言語だったんでゲソね [*24]。

というわけで、これが Scheme と「継続」の縁が深い理由なんでゲソ。そもそもの Scheme が生まれるきっかけになったアクターモデルが「継続」をとても重要視したんでゲソね。

ところで、今の Scheme には存在しないでゲソが、開発当初の Scheme にはちゃんとアクターを作る関数 [*25] が用意されていたようでゲソ [*26]。

```
(alpha (parameters ... ) body)
```

この alpha は関数を作る lambda に (見た目も含めて) よく似てるんでゲソが、アクターなので値をリターンしたりはしないんでゲソ。代わりに計算結果を他のアクターにメッセージパッシングするんでゲソよ。例えば階乗計算を行うアクターはこんな感じになるんでゲソ (もう面倒だから説明はしないでゲソ。CONS アクターが読めたならきっとこれも読めるはずでゲソ (?))。

```
(define factorial
  (alpha (n c)
    (= n 0
      (alpha () (c 1))
      (alpha ()
        (- n 1
          (alpha (z)
            (factorial z
              (alpha (y)
                (* n y c)))))))))
```

ところが、実際に Scheme インタープリタが出来てみると、驚くべきことが分かったんでゲソ。なんと、インタープリタ内でアクターを処理する部分と関数 (クロージャ) を処理する部分の実装が同じになってしまってたんでゲソ！ これはどういうことでゲソ? 本質的に違いがあるものなら全く同じに処理できるわけはないでゲソ。ということは...

だいたい予想が付いたかもしれないでゲソが、この結果を見て 2 人が出した結論は、「アクターは Lisp の関数と同一のものだ」というものだったんでゲソ[*27]。ちなみに、アクターのメッセージパッシングは Lisp の関数では末尾位置での呼び出しに対応したんでゲソ。というわけでより正確に言うと「アクターは末尾呼び出ししかしない Lisp の関数だ」でゲソね [*28]。(ついでに言うと、「継続」は「末尾位置で呼び出される関数」のことでゲソね)

上で紹介した alpha が今の Scheme に残っていない理由もこれで納得いったでゲソ？ alpha と lambda は本質的に同じなんだから両方用意しておく必要はないでゲソ。

ただし、アクターを Lisp の関数と同一視するにはちょっと問題もあったんでゲソ。アクターモデルでは、アクター同士で何度メッセージパッシングしてても特に問題は起こらないんでゲソ。ところが Lisp の関数の

[*24] ところで、Scheme が Lisp っぽい言語になったのは当時 MIT で Lisp が主流だったからでゲソが、伝統的な Lisp ってみんな dynamic scope だったんでゲソ。それなのになんで Scheme が lexical scope を持つようになったかというと、(アクター理論では lexical scope が必要そうに見えたというのもあるんでゲソが、もう 1 つの理由として) Sussmann 先生が Algol 60 にハマっていたから、というのもあるんでゲソ。つまり Algol がなければ Scheme は全然違った言語になっていたかもしれないんでゲソ。で、実際この lexical scope を採用したことが後で大きく効いてくるんでゲソ。上で書いた「Algol が Scheme に大きな影響を与えた」というのはこういう意味なんでゲソ。

[*25] (細かいことなのでどうでもいいでゲソが) 正確には関数じゃなくていわゆる「特殊形式 (Special form)」でゲソ。

[*26] ちなみに、アクターにメッセージパッシングする send という特殊形式もあったようでゲソ。構文としては “(send actor argument1 ... argumentN)” のように書いたそうでゲソ。ただ、残念なことに (?) インタープリタ内では関数とアクターは区別が付いちゃうんでゲソ。ということで、(関数呼び出しもメッセージパッシングも) どっちも関数適用と同じ記法にしておいて、関数なら関数呼び出し、アクターならメッセージパッシングにした方が便利じゃなイカ、ということでこちらは早々に削除されたようでゲソ [15]。

[*27] ちなみに、これが lexical scope を採用したことによる効果だったんでゲソ。つまり、伝統的な dynamic scope な Lisp ではアクターは関数とは一致しないでゲソ。

[*28] ちなみに、Carl Hewitt 先生は「アクターは並列性とかの面でクロージャーに勝っていてむしろアクターの方がすごいだろ」と主張されてたりするようでゲソ。でも、今回は主に Scheme 派の主張を取り上げるでゲソよ。気になる人はアクターの方も調べてみるといいと思うでゲソ。

場合は、関数呼び出しをナイーブに実装していると (末尾だろうと末尾でなかろうと) とにかく関数を呼び出すたびにスタックフレームを積んでしまうでゲソ。ということは、アクターを真似して末尾呼び出しを延々と繰り返していると、そのうちスタックオーバーフローしてしまうじゃなイカ！ これでは、(理論上はさておき) 実用上はとても同一のものとは言えないでゲソ。

そこで、アクターパラダイムを完全に Lisp の世界に取り込むために「末尾呼び出しは最適化して goto にすべし (スタックを積んではならん)」という言語仕様が Scheme に取り込まれたんでゲソ。これが、Scheme において「継続」と並んで「末尾呼び出しの最適化 (Tail Call Optimization, TCO)」が特別に重要視される理由なんでゲソ。

ちなみに末尾呼び出しの最適化という概念自体、関数型の世界に登場したのはこのときが初めてだったようでゲソ。今となっては当たり前の末尾呼び出しの最適化にも、そもそもはこういう経緯があったんでゲソ。

付録: Lambda-ka: The Ultimate Invader
（または Lisp が如何にして心配するのを止めてラムダ計算を愛するようになったか）

上で説明したように Scheme はアクター理論と Lisp を結びつけたんでゲソが、実はもう一つ大事なものを結びつけたんでゲソ。 Scheme の特徴である「lexical scope」と「末尾呼び出しの最適化 (TCO)」、これって、Lisp というより、その元になったラムダ計算に近くなイカ?
例えば、ラムダ計算の評価規則であるβ簡約って、変数の参照先はプログラム中の構文構造で決まるでゲソ? これはまさしく lexical scope でゲソね。
もう一つの TCO については、次の fact の例を見てみるでゲソ。(もちろん本当のラムダ計算には数値とか named-let みたいなものはないでゲソが、ここではチャーチ数とか Y コンビネータのシンタックスシュガーだと思って欲しいでゲソ)

末尾再帰の場合:

```
(define (fact n)
 (let loop ((n n) (acc 1))
  (if (= n 1) acc
   (loop (- n 1) (* acc n))))))
```

末尾再帰じゃない場合:

```
(define (fact n)
 (if (= n 1) 1
  (* n (fact (- n 1))))))
```

この 2 つをβ簡約していくと次のようになるでゲソ ？ここで見て欲しいのは簡約途中のネストの数でゲソ。β簡約していくと、末尾再帰じゃない方はネストが増えていくのに末尾再帰版は増えないじゃなイカ! これって TCO のことだとは考えられないでゲソ?

末尾再帰の場合:

```
(fact 3)
→ (loop 3 1)
→ (loop (- 3 1) (* 3 1))
→ (loop 2 3)
→ (loop (- 2 1) (* 2 3))
→ (loop 1 6)
→ 6
```

末尾再帰じゃない場合:

```
(fact 3)
 → (* 3 (fact 2))
 → (* 3 (* 2 (fact 1)))
 → (* 3 (* 2 1))
 → (* 3 2)
 → 6
```

というわけで、Scheme はアクターと Lisp にラムダ計算まで結びつけたんでゲソ。アクターと Lisp の関連性も想定外だったそうでゲソが、ラムダ計算との親和性についても当初は全く考えていなかったそうでゲソ (これに気づいた後でラムダ計算を考え出した Alonzo Church 先生の 20 年以上前の論文を読んだら、既に cons セルアクターみたいなものが書いてあって驚いたんだとか)。このことでラムダ計算自体の表現力の高さに気づいた 2 人は、それ以降 “Lambda Papers” と呼ばれる論文を発表してラムダ計算の凄さを布教して回ったんでゲソ。そしてこの過程でラムダ計算の強力さが関数型の世界で再認識されていったんでゲソよ。
今からだとなかなか想像できないでゲソが、Scheme 登場以前の関数型の世界では dynamic scope な Lisp が主流だったので純粋なラムダ計算からは少し遠ざかってたんでゲソ。それどころか「lexical scope だと高速な実装が作れないんじゃなイカ」みたいなことまで言われてたでゲソ。だけど、これをきっかけに原点であるラムダ計算を見直す動きが進んでいったんでゲソよ。ということは、もし Scheme がなかったら Haskell みたいな純粋関数型言語も誕生しなかったかもしれないでゲソ。みんなもっと Scheme の偉大さを称えるべきじゃなイカ?

15.4 そろそろ CPS についてひとこと言っておくでゲソ

さて「アクターモデルがメッセージパッシングだけで色々な制御構造を作れる」ということは「ラムダ計算でも末尾呼び出しだけでプログラムを記述すれば色々な制御構造が作れる」ということにならなイカ？ [*29]実際、この「末尾呼び出しだけでプログラムを書く」というスタイルは非常に表現力が高いので、“Continuation-passing style”(継続渡しスタイル, CPS) と名前もついているほどでゲソ [*30]。

ところで、もしまだ「CPS の表現力が高い」というのが信じられない人は、こう考えればどうでゲソ? 末尾呼び出しって最適化されれば単なる goto になるでゲソ? ところで、goto と if(条件分岐) さえあればどんな制御構造だって作れるでゲソ? [*31]ということは、末尾呼び出しと if さえあればどんな制御構造だって作れるはずじゃなイカ! [*32]

参考までに、CPS で書いた例を 1 つ出しておくでゲソ。さっきのアクター版と比較しやすいように factorial の例でゲソ

(といっても Scheme には CPS 用の述語とか数学関数 (具体的には = とか - とか * とか) はデフォルトでは用意されてないので、さっきの例と比べるとちょっとアクター感 (CPS 感) が足りないかもしれないでゲソ。こういった述語や数学関数もさっきのアクター版では継続を受け取ってくれたんでゲソが、ここではしょうが無いので普通に関数として返り値をリターンさせて使ってるでゲソ)。

```
(define cps_factorial
  (lambda (n c)
    (if (= n 0)
        (c 1)
        (cps_factorial (- n 1)
                       (lambda (m)
                          (c (* m n)))))))
```

15.4.1 ここらで「継続」についてもう一度整理しておくでゲソ

今回の話の最初に、「継続」=「その後に行う計算処理」=「スレッドの状態」=(大抵のケースで)「スタックの中身」という説明をしたでゲソ。だけどさっきのアクターや CPS の話で出てきた「継続」は「スタックの中身」とはちょっと違うような気がしなイカ?

というわけで、もう一度「継続」について整理しておくでゲソ。

で、そもそもなんでスタックなんていう構造が必要なのかと言えば、リターンするためじゃなイカ? 普通の関数呼び出しはリターンしないといけないでゲソね。だから、リターン先を覚えておくためにスタックという LIFO (Last-in-First-out) な構造でリターンアドレスを管理してるでゲソ[*33] [*34]。

だけど、メッセージパッシングにはリターンという概念はなかったじゃなイカ。そして、そこから派生した CPS にもリターンという概念はないでゲソ。というわけで、メッセージパッシングや CPS はスタックを

[*29] というわけで、アクターパラダイムはラムダ計算上でのものすごく強力なデザインパターン、とも考えられるでゲソね。

[*30] この CPS に既に馴染みがあった人は、CPS って元から関数型の世界の概念だと思ってなかったでゲソ? 実際には、CPS は全然違うアクターパラダイムというパラダイムから輸入されてきた概念だったんでゲソね。といっても、今となっては「なじむ　実に！　なじむぞ」ってくらいラムダの世界に溶け込んじゃってるのでどっちでもいいでゲソが。

[*31] というか、(普通の CPU って大雑把に言えばその 2 つくらいしか制御構造がないから) それで作れなかったらコンピュータ上で動かないでゲソ。

[*32] 正確に言えば、さっきのアクター版 factorial でやっていたように、条件分岐も述語関数による継続の起動としてしまえば全部末尾呼び出しで説明することも可能でゲソ。

[*33] 余談でゲソが、“stack” という用語を作ったのは Edsger Dijkstra 先生 [4] で、そのときの動機は「再帰呼び出しを可能にするにはリターンアドレスを管理する必要があるから」というものだったんでゲソ。で、なんで再帰呼び出し限定かというと、再帰呼び出しがなければ各関数が予めリターンアドレス用の領域を 1 つずつ確保しておけばなんとかなるじゃなイカ。実際、再帰呼び出しがなかった昔の Fortran や COBOL はそういう実装になっていたらしいでゲソよ。

[*34] こう言うと「局所変数を管理するという役割もあるじゃなイカ」と言われるかもしれないでゲソが、そもそも変数用のメモリ領域の管理とコントロールフローの管理を同期させなきゃいけない理由はないでゲソ? (Algol 以降のブロック構造を持つ手続き型言語にとっては、それが自然な実装だったのは確かでゲソが…)。というかむしろ関数型の言語では、クロージャとかに捕捉された自由変数はクロージャーが生き続ける限り永遠に生き続ける (格好よく言うと「無限のエクステントを持つ」) から、スタックと同期させちゃ駄目でゲソ。というわけで、ここではスタックの役割についてはリターンアドレスの管理だけを考えることにするでゲソ。

必要としないんでゲソ (というか、一方通行の制御だから LIFO な構造なんてあっても使い道がないというか…)。

さっき言ったとおり、(大抵のケースでは)「継続」=「スレッドの状態」=「スタックの中身」なんでゲソが、メッセージパッシングや CPS においては最後のスタックの中身って (ほとんどあるいは全く) スッカラカンでゲソ。代わりに、(スタックで管理するんじゃなくて) 自分で明示的にアクターやクロージャーを受け渡すことで管理してるんでゲソね。

というわけで、メッセージパッシングや CPS のような一方通行な制御では、「継続」=「その後に行う計算処理」=「スレッドの状態」=「そのスレッドに引数として渡されてきているアクター or クロージャー」ということになるんでゲソ。

15.4.2 ついでに、継続を捕捉する演算子 (call/cc) についても整理しておくでゲソ

上で説明したように、CPS はものすごく表現能力が高いから、プログラムを全部 CPS で書けば制御フローなんて思いのままでゲソ。

だけど、現実問題としてプログラム全体を CPS で書くのは結構大変だし、いろんな事情でどうしても CPS にできないことも多いでゲソ。

で、しょうがないから普通は一般的な関数スタイルでプログラムを書いてしまうでゲソよね? でもそうなると、制御フローの管理は「スタック構造で管理されるリターンアドレス」にお任せすることになってしまうでゲソ。そして、こうなってしまうと制御フローはスタック構造 (行ったら必ず帰ってこなきゃいけないという制約付きのフロー) にしばられるから CPS ほど柔軟には制御フローをコントロールできなくなってしまうでゲソ。

だけどそれだとなんだか悲しくなイカ? そこで、そういう場合でも CPS 並の強力な制御能力を使えるように、と導入されたのが継続を捕捉する演算子 call/cc なんでゲソ。つまり、call/cc というのは「(人手でプログラムを CPS 形式に直すのが難しいときに) プログラムを自動的に CPS に直して、継続に当たる部分を取り出してくれる演算子」と考えてもいいんでゲソ。

というわけで、そもそもプログラムを全部 CPS で書けるなら call/cc なんて要らないんでゲソよ。それに、関数的な「行ったら必ず返ってくる」フローで満足している場合にも必要ないんでゲソ。全部 CPS では書けないけど CPS ばりの制御力が欲しい、という時に使えるのが call/cc なんでゲソ [*35]。

15.4.3 ところで CPS って何かに似てなイカ?

ここで紹介した CPS には、この他にもいろいろと面白い性質があるでゲソ。例えばその一つに「完全に CPS で書くと評価順が関係なくなる」[*36] というものがあるんでゲソ。もうちょっとかみ砕いて言うと、(評価順というとセーキジュンジョ (Normal Order) とかテキヨージュンジョ (Applicative Order) とかいろいろあるでゲソが) どういう評価順の下でも常に同じ順番で評価が行われるようになるんでゲソ。これだけ聞くと不思議かもしれないでゲソが、そもそも評価順って「簡約できる箇所が複数あるときにどこから簡約するか」という話だったじゃなイカ? ところが完全に CPS で書くと、簡約できる箇所は常に 1 箇所 (式中の一番外側) しかなくなるんでゲソ。なぜかというと、もし式の内側に簡約できる箇所があったら、それって (関数呼び出しだと考えれば) その式の評価値をリターンしてくることにならなイカ? 完全に CPS で書いた場合にはリターンという概念はなくなるのでそんな式は存在しないんでゲソ。

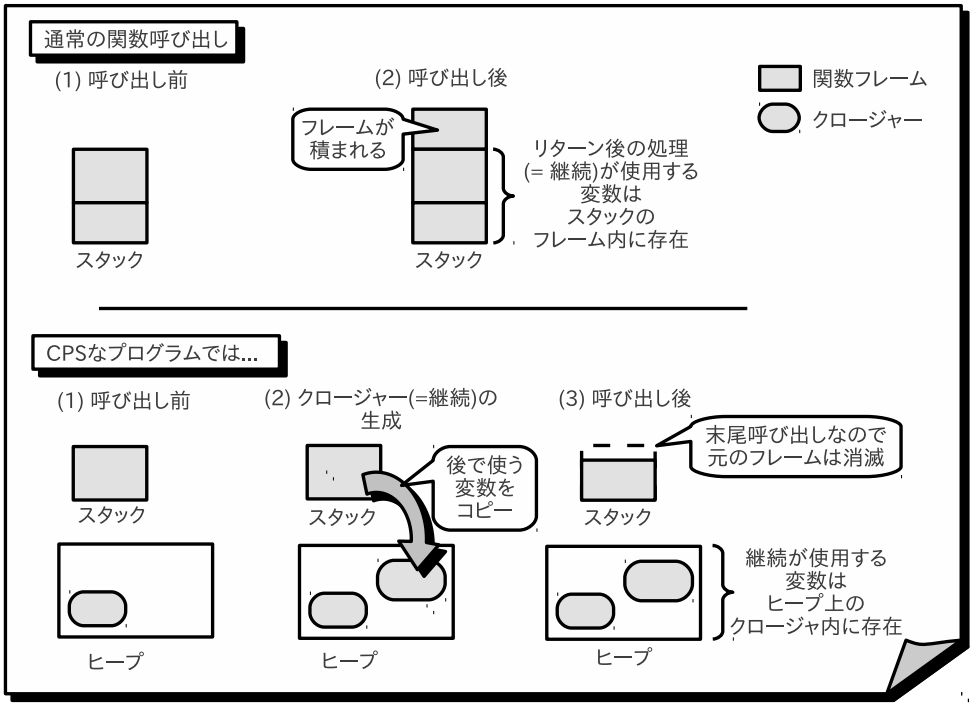
ところで、こういった評価順に関する話は特に副作用が出てきたときに問題になるでゲソね。最近の関数型の世界では、評価順を解決する方法として「モナド」が有名でゲソ。だけど、評価順の問題であれば、この通り CPS でも解決できるでゲソ。というかもっと言うと、モナドと CPS には関連性があるんでゲソよ。モナドを使って書かれたプログラムと CPS で書かれたプログラムって見た目が似てないでゲソ? (CPS で継続として関数を連ねていく書き方と、モナドで bind 演算子による関数合成を連ねていく書き方、よくよく見るとそっくりじゃなイカ?)。実のところ、モナドは CPS を一般化したデザインパターン (あるいは CPS はモナ

[*35] ちなみに、前の方の call/cc を説明していた箇所の注釈で、「継続と関数には関連性がある」とか「継続= 絶対にリターンしてこない関数」とか書いてあったのを覚えてるでゲソ? これは、要するに call/cc が捕捉する継続というのは「CPS で書かれたプログラムにおける末尾呼び出しされる関数」だからなんでゲソ。この関数は末尾呼び出しされるわけだから決してリターンしないでゲソ? だから、call/cc が捕捉した継続も決してリターンしないんでゲソ。

[*36] ただし、「完全に」と言うのがポイントでゲソ。上の cps_factorial みたいな (ちょっと中途半端な) CPS ではそうはならないでゲソ。

ドの特殊例 [*37]) とも考えられるんでゲソ [18] [6]。これは CPS とモナドの間にこういう類似性があるからでゲソね。

(ところで、この図は次の付録用のものでゲソ。気になった人は付録を見るでゲソ。)



[*37] というか Haskell の Continuation モナドがまさにそれでゲソね

付録 : CPS についての余談

さて「CPS はスタックを必要としない」と言ったんでゲソが、実際に CPS で書いたプログラムを動かすとどうなるのか気にならなイカ? 普通の処理系では関数呼び出しの際にはスタックフレームを積むことが多いので、CPS で書かれたプログラムでもスタックを全く使わないわけではないでゲソ。ただし、全部 CPS で書いた場合には (ちゃんと TCO する処理系なら) 全部の呼び出しが最適化されて goto になるわけでゲソ? だから、スタックフレームは常に 1 個しか存在しなくなるんでゲソ。というわけで CPS で書いたプログラムは、スタックは (使うには使うけど) ほとんど消費しない、ということになるでゲソ。

ところで、これについてもうちょっと考えると「CPS で書けばスタックフレームが積まれないからスタックオーバーフローも起こらない！ キタコレ！」みたいな気もしてこなイカ? だけどなんだか変でゲソね? CPS にするだけでそんなにいいことがあるなら、なんで皆しないんでゲソ?

これについての答えは上の図に書いたとおりでゲソが、スタックフレームを積まなくなる代わりに、その分ヒープを食うんでゲソ[a]。そもそも末尾呼び出し最適化で呼び出し時にスタックフレームを破棄しちゃって問題ないのは、その後の計算処理 (=継続) が必要とする値はヒープ上のクロージャーにコピーしているからでゲソ。

ところで、ここまでの内容を逆に考えると、call/cc みたいな「継続を捕捉する処理」には 2 通りの実装方法があるということにもなるでゲソ。1 つは (最初に紹介したように) 普通の関数スタイルで書いておいて継続を捕捉したくなった時点でスタックをコピーする方法、もう 1 つは予めプログラムを CPS 形式で書いておく方法でゲソ。もちろんどっちを使っても実現できるんでゲソが、スタックを効率的にコピー/復帰する手段が提供されていない Java VM のような処理系上では、プログラムを CPS 変換するアプローチの方が主流[b] でゲソ。例えば、Kilim [14] や Apache JavaFlow [1] というフレームワークでは、バイトコードレベルで Java のプログラムを CPS 変換することで継続捕捉操作を実現してるでゲソ。他にも、(最近人気が高い Scala なんかでも) Scala コンパイラがソースコードレベルで CPS 変換を掛けて継続[c]を取り出してたりするでゲソ [13]。

---

[a] 正確に言うと、クロージャーの実現方法にはいろいろあるから常にヒープを消費するとは限らないでゲソ。ただし、本質的に必要なデータ量が減らない限り、スタックにせよヒープにせよどこかを消費しないといけないのは間違いないでゲソ。というわけで、CPS にしたらメモリの使用量が減る、なんてのは絶対に嘘でゲソ。

[b] 主流と言っても、Java VM 上で継続を使うこと自体がものすごくマイナーな気もするでゲソが…

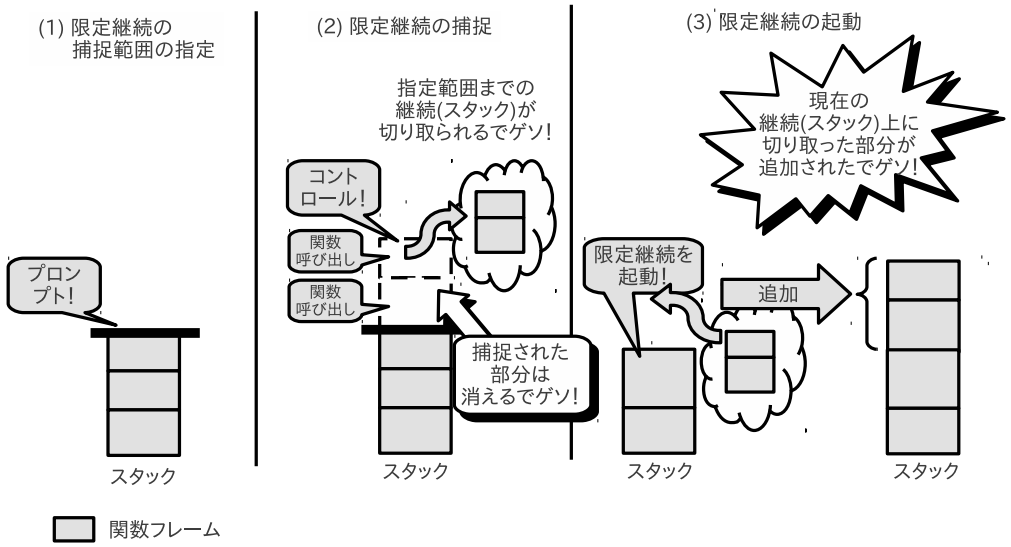
[c] 細かい話でゲソが、正確には「継続」じゃなくて「限定継続」なんでゲソ。限定継続については次の節で説明するでゲソ。

15.5 「ゲンテーケーゾク」って聞いたことはないでゲソ?

ところで、これまでは普通の「継続」について話をしてきたんでゲソが、最近は「限定継続 (Delimited Continuation)」という言葉の方がよく耳にするんじゃなイカ?「限定継続」というのは、ちょっと変わった新手の [*38] 継続でゲソ。ちなみに、限定継続という呼び名の他に、「部分継続 (Partial Continuation)」とか「合成可能継続 (Composable Continuation)」といった呼び名もあるんでゲソ。

限定継続が生まれたいきさつは後で説明するとして、まずどういうものかだけ簡単に説明してみるでゲソ。

継続は call/cc のような演算子で捕捉できたでゲソが、限定継続では二種類の演算子で捕捉を行うんでゲソ。限定継続にはいくつか流儀があってそれぞれに演算子の名前とかが違ったりするでゲソが、それも後で話すとして、とりあえずここでは control と prompt という演算子で説明してみるでゲソ。この演算子では、prompt で継続捕捉の終端点を指定し、control が prompt で指定された点までの継続を捕捉するんでゲソ。図にすると次のような感じでゲソ (図は引き続き「継続=スタック」という感じの例でお届けするでゲソ)。



こんな感じで、普通の継続と違って捕捉する範囲が control と prompt の間に限定されている (一部分しか捕捉しない) ので「限定継続 (部分継続)」と呼ばれるんでゲソ。あと、普通の継続だと呼び出したら決して返ってこないんでゲソが、部分継続は普通の関数のように呼び出し元にリターンしてくる、というのもポイントでゲソ (現在の継続を置き換えるんじゃなく、現在の継続の上に追加されるような感じになるんでゲソ)。このため、限定継続は他の処理と合成しやすい (composability がいい)、ということで「合成可能継続 (Composable Continuation)」とも呼ばれるんでゲソ。あと、(微妙な違いでゲソが) call/cc では捕捉した継続もそのまま残っていたのに限定継続では捕捉した分は現在の継続からは消える、というのも違う点でゲソ。

図だけだと具体的な動きが分かりにくいので、限定継続を使った例を 1 つ書いておくでゲソ。

```
(+ 1 (prompt (+ 2 (control k (k (k 42))))))
```

さて、この場合に何が起こるか見ていこうじゃなイカ。

まず、control が捕捉するのは「prompt と control の間」でゲソ。だからこの例で捕捉される限定継続は下のような式でゲソ。

```
(+ 2 □)
```

あるいは、これは以下のような関数のことだと考えてもいいでゲソね。

[*38] といっても初めて登場したのは 1988 年なのでもうあんまり新手って感じじゃないでゲソが…

```
(lambda (n)  (+ 2 n))
```

というわけで、上の式はこんな感じで評価されるんでゲソ。

```
(+ 1  (prompt (+ 2 (control k
    (k
      (k
        42))))))

→ (+ 1                      ; control によって、捕捉範囲の式が消えるでゲソ
     ((lambda (n) (+ 2 n))  ; 代わりに変数 k が捕捉した部分継続で置き換えられるでゲソ
      ((lambda (n) (+ 2 n)) ; (もう一つの k も同じく置き換えられるでゲソ)
        42)))

→ 47                        ; これが最終的な評価値でゲソ
```

さて、一応説明してみたんでゲソが、何となく雰囲気は分かったでゲソ? 初めて見るとすごく奇妙な動きに見えたりするんでゲソが、そういう時はコルーチンをイメージすると理解しやすいかもしれないでゲソ。

先にコルーチンを知らない人のために簡単に説明しておくと、コルーチンというのは一種の関数なんでゲソ。ただし、普通の関数と違って状態のようなものを持つんでゲソ。もうちょっと具体的に言うと、コルーチンは呼び出されると前回リターンしたところから実行が再開されるんでゲソ。

残念ながら Scheme とか Haskell にはデフォルトではコルーチンを作る機能が用意されてないので、とりあえず Lua での例を出してみるでゲソ。

```
coro = coroutine.create(
  function()
    coroutine.yield("イヤー！ ")     -- 1 回目は "イヤー！ " を返すでゲソ
    coroutine.yield("グワー！ ")     -- 2 回目には "グワー！ " を返すでゲソ
    coroutine.yield("アイエエエ！ ") -- 3 回目には "アイエエエ！ " を返すでゲソ
  end)

print(coroutine.resume(coro))     -- 1 回目の呼び出し
> true    イヤー！
print(coroutine.resume(coro))     -- 2 回目の呼び出し
> true    グワー！
print(coroutine.resume(coro))     -- 3 回目の呼び出し
> true    アイエエエ！
```

Lua では coroutine.create() という関数でコルーチンを作れるんでゲソ。コルーチン内でリターンしたい時には coroutine.yield() を使えばいいんでゲソ。また、(関数呼び出しならぬ) コルーチン呼び出しをしたいときには coroutine.resume() でゲソ。この例の通り、作ったコルーチンは呼び出す度に前回 yield した箇所から実行が再開されるんでゲソ (ちなみに true と出てるのは coroutine.resume() が呼び出しの成否を示すための真偽値も一緒に返すからでゲソ)。

で、これと限定継続の関係なんでゲソが、限定継続の方も「prompt で囲われた範囲」を 1 つの関数と考えると、コルーチンの動きと少し似ているんでゲソ。コルーチンの挙動は「yield すると関数からリターン、次に呼び出されると前回の yield から実行再開」だったでゲソね。限定継続の場合は「control すると prompt で囲われた範囲からリターン、control で得られた限定継続を呼び出すと control した所から実行再開」なんでゲソ。

```
(define coro-modoki
  (prompt
    (string-append "「コワイ!」"       ; この文字列連結処理が
                                       ;   「残りの処理（限定継続）」になるでゲソ。
      (control k                       ; 最初は、残りの処理（限定継続 k）と
        (cons k                        ;   "「アイエエエ!?」..." を返すでゲソ
          "「アイエエエ!?」「ケイゾク!?ケイゾクナンデ!?」")))))

(cdr coro-modoki)                                ; とりあえず最初の結果を
> 「アイエエエ!?」「ケイゾク!?ケイゾクナンデ!?」   ;   見てみるでゲソ

((car coro-modoki) "「ゴボボーッ!」")             ; 引き続き、残りの処理も実行して
> 「コワイ!」「ゴボボーッ!」                      ;   結果を見てみるでゲソ
```

ただし、コルーチンの場合は (勝手に状態が更新されていくので) 元の状態に戻したりすることは出来ないでゲソが、限定継続の場合は (control で捕捉した限定継続をどう使うかはユーザーに任されているので) 限定継続を使わずにもう一度最初から実行させることも出来るし、あるいは同じ限定継続を何度も実行することも出来るでゲソ。というわけで、制御構造としては限定継続の方が柔軟でゲソね (実際、限定継続さえあればコルーチンを作るのなんて楽勝でゲソ)。

15.6 どうして限定継続が考え出されたのか気にならなイカ?

以上で限定継続について一通りの説明は終わったんでゲソが、そもそもなんでこんなのが考え出されたのか不思議じゃなイカ? ここまでの説明だけだと (INTERCAL の COMEFROM みたいな) キワモノ制御演算子の一種に見えるかもしれないので、少し誕生の経緯も説明しておくでゲソ。

ところで、上の継続の説明で「継続=スレッドの状態」という話を見た時に、名状しがたい冒涜的な気持ちになった人はいないでゲソ? 確かに「状態」なんて言葉を聞くと、ついつい参照透明性のことが心配になってしまうでゲソね。

で、実は嫌な予感の通り、call/cc のような「継続を捕捉する演算子」は参照透明性を持たないんでゲソ[*39]。意外に思ったかもしれないでゲソが[*40]、次のように考えてみれば納得できるんじゃなイカ?

・参照透明性
「同じ関数に同じ引数が適用されれば、（文脈や状態といったそれ以外のものに関わらず）常に同じ値が返る性質」
・継続を捕捉する演算子
「その演算子が呼び出された時点での継続（≒スレッドの状態）を取り出す演算子」

call/cc が取り出す「呼び出された時点でのスレッドの状態」は、どう考えても参照透明性が要求する「文脈や状態に依存しない値」とは思えないでゲソ。実際、call/cc を使って次のような関数を作ってみると簡単に参照透明性を壊せるでゲソ。

```
(call/cc (lambda (k) k)) ; call/cc の引数が継続をそのまま返すだけなので、
                         ;   この式を評価すると「その時点での継続」になるでゲソ
```

次の 2 つのプログラムを考えてみるでゲソ。

[*39] この話については、川合史郎さんの次のページに詳しい話が載っているのでお勧めでゲソ。http://practical-scheme.net/wiliki/wiliki.cgi?Scheme%3Acall%2Fcc%e3%81%a8%e5%89%af%e4%bd%9c%e7%94%a8

[*40] 例えば、Haskell の Continuation モナドを知ってる人は、参照透明な Haskell にも callCC があるじゃなイカって思ったかもしれないでゲソ。だけど、もう一度その callCC をよく見て欲しいでゲソ。Haskell の callCC は「継続を捕捉」したりはしないので参照透明なんでゲソ。

```
(1) (let ((a (call/cc (lambda (k) k)))
          (b (call/cc (lambda (k) k))))
      (cond
        ((eq? a 0) #t)
        ((eq? b 0) #f)
        (else      (a 0))))   ; <= 違いはここだけ（こっちは a を使用）

(2) (let ((a (call/cc (lambda (k) k)))
          (b (call/cc (lambda (k) k))))
      (cond
        ((eq? a 0) #t)
        ((eq? b 0) #f)
        (else      (b 0))))   ; <= 違いはここだけ（こっちは b を使用）
```

この (1) と (2)、ほとんど同じプログラムでゲソ。違いはたった一カ所で、最後の行で a に 0 を適用するか b に 0 を適用するかだけなんでゲソ。しかも a と b はどちらも (call/cc (lambda (k) k)) という式の評価値じゃなイカ。ということは、(call/cc (lambda (k) k)) が参照透明性を持つなら、この a と b を入れ替えただけのプログラムは同じ評価値を返すはずでゲソ? ところが実際にやってみると (1) は #t になり (2) は #f になるでゲソ[*41][*42]

とまぁこんな問題があったんでゲソが、1980 年代の末になるまでは call/cc のような「継続を捕捉する演算子」はあくまでプログラミング言語の機能としてしか考えられていなかったんでゲソ。逆に言うと、元々のラムダ計算の世界でどうなるのかは考えられていなかったんでゲソね。そのせいで、それを取り入れた計算体系の性質 (チャーチロッサーセーとかサンショートーメーセーとか) はよく分かってなかったんでゲソ。そこで、継続捕捉操作で拡張したラムダ計算を作るという研究 [5] が行われたんでゲソが、これが限定継続の始まりなんでゲソ。

ところで、継続が状態だと考えると参照透明な世界で表現するのは難しそうじゃなイカ? だけど、こういう一見すると参照透明じゃないものを参照透明な世界で扱う方法があったでゲソね。そう、「モナド」でゲソ。限定継続のアプローチは少しモナドに似てるから、「モナド化することで参照透明化した継続捕捉操作」とも考えられるでゲソ。とはいえ、元々の論文のアイデアは全然モナドじゃない (というかモナドには一言も触れてない) ので、今回はあんまりモナドにこだわらずに説明してみるでゲソ。

では、元々の論文のアイデアを簡単に説明してみるでゲソ。まず、元々の論文ではラムダ計算に対して次のような F という演算子[*43]を追加しようとしてたんでゲソ。

```
F M   ; ただし、M は「引数として継続を受け取るラムダ式」に簡約される式
```

この F 演算子、ほとんど call/cc とそっくりなんでゲソが、捕捉された継続が消えてなくなるというのが call/cc との違いでゲソ。(図は引き続き「継続=スタック」の例で説明するでゲソ)

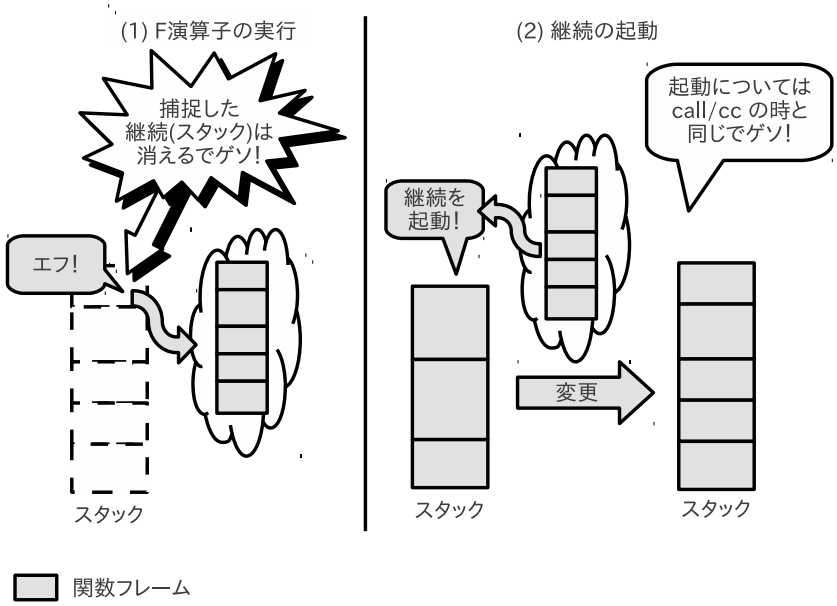
まぁあんまり大きな違いじゃないでゲソが一応どういうことかを説明すると、(引数である M の中で捕捉

[*41] ところで、この話ちょっと紛らわしいので注意が必要でゲソ。ここで言っているのは「継続自体に参照透明性がない」という話ではないでゲソ。実際、継続自体は単に末尾位置で呼び出されるだけの関数だったりするわけで、(その中で明示的に副作用とかを起こさない限り) 参照透明でゲソ。だから、CPS でプログラムを書いたりするのは全く問題ないんでゲソ。ここで問題になってるのは、「継続」自体じゃなくて、「継続を捕捉する演算子」でゲソ。

[*42] (ついでに、上で紹介した川合史郎さんのページにも書いてある話でゲソが、念のために紹介しておくと) 上で書いたように call/cc って単に「プログラムを CPS の形式に変換してくれる」演算子なんでゲソ。だから、プログラムを手動で CPS 変換すれば call/cc なんて必要ないでゲソ。ところでこのとき、call/cc を使った方は参照透明じゃなくて、同じ計算処理をするのに CPS に直した方は参照透明性があるということになるんでゲソが、ちょっと不思議じゃなイカ? でもこれは次のように考えればたぶん納得できるでゲソ。プログラムを CPS に直すのはプログラム全体を書き直さないといけない大域的な変更でゲソ。実際、CPS で書けば call/cc なんて要らないのにわざわざ call/cc が用意されてるのは、CPS に直す作業がものすごく影響範囲が広い大変な作業だからでゲソ [16]。で、そんな大域的な変更をしていいなら、どんなに参照透明性のないものだって参照透明性のある形に直せるじゃなイカ (例えば I/O だって I/O モナドを使った参照透明な式に出来ちゃうでゲソ?)。だから、参照透明性の無いものが (大域的な変更を通して) 参照透明性のあるものと対応するのは不思議じゃないんでゲソ。

[*43] この演算子は Felleisen's F operator とか Felleisen's C operator とか呼ばれることが多いようでゲソ。

した継続が起動されなかった場合) 継続が全部なくなっているので M の返値がそのままプログラム全体の評価値になるんでゲソ。言い換えれば、M の返値を持って一気にトップレベルまで大域脱出してしまう感じになるんでゲソ[*44]。



なので例えば call/cc とはこんな感じで違いが出るでゲソ (本当は F はラムダ計算に追加する演算子でゲソが、call/cc との比較のために scheme っぽく書いてみるでゲソ)。なお、書くのがちょっと面倒だから「捕捉された継続」の詳細は書かずに●で済ませてみるでゲソ。

```
・ F の場合:
   (+ (F (lambda (k) 42)) 1)
→ ((lambda (k) 42) ●) ; 継続である (+ □ 1) の部分は継続捕捉によって消えるでゲソ
→ 42                  ; 継続が無くなってるので、42 が全体の評価値になるでゲソ

・ call/cc の場合:
   (+ (call/cc (lambda (k) 42)) 1)
→ (+ ((lambda (k) 42) ●) 1)  ; 継続である (+ □ 1) は残ったままでゲソ
→ (+ 42 1)                   ; 継続が残っているので、まだ残りの処理があるでゲソ
→ 43                         ; これが最終的な評価値でゲソ
```

もちろん明示的に継続を起動すれば消えちゃった継続は元に戻せるので、大域脱出したくない場合でも心配ないでゲソ。例えば、call/cc 的に使いたい場合はイカのようにすればいいだけでゲソ。

```
(call/cc M) =                    ; (M k) を計算した後、
   (F (lambda (k) (k (M k)))) ;   k を起動して継続を元に戻せば call/cc 相当でゲソ
```

で、この F 演算子なんでゲソが、単純に導入するとやっぱり参照透明性はないんでゲソ。例えば次の 2 つの式を考えてみるでゲソ (上でも書いた通り F は本当はラムダ計算に追加する演算子でゲソが、引き続きちょっと Scheme っぽく書くでゲソ)。

[*44] control とかが捕捉した継続を消し去るという挙動もこれに由来してるでゲソ。ちなみになんでこうなったかというと、数学的にはこっちの方がよさげだと考えたから、らしいでゲソ。確かに本当に CPS でプログラムを書いた場合は受け取った継続を使うも使わないも自由だから「現在の継続を使わない」という選択肢がある方が自然かもしれないでゲソ。

```
(a) (F (lambda (k) 0))
    → ((lambda (k) 0) ●) ; F による継続捕捉
    → 0
(b) 0
    → 0
```

この 2 つの式、見ての通りどちらも 0 に評価されるでゲソ。

さて、評価値が同じなんだから、参照透明性があればこの 2 つの式は自由に入れ替えても問題ないでゲソ? ところがやっぱりそうはならないんでゲソ。次の式を考えてみるでゲソ。

```
(a') (exp (F (lambda (k) 0)))
     → ((lambda (k) 0) ●) ; F による継続捕捉
     → 0
(b') (exp 0)
     → 1
```

違ってるのは (F (lambda (k) 0)) と 0 だけなのに、(a') 式の評価値は 0 で (b') は 1 じゃなイカ。というわけで、明らかに参照透明性がないでゲソ！

それじゃいよいよ、限定継続を入れるとこれがどう解決されるかを説明するでゲソ。最初にアイデアを大雑把に言っておくと、解決策というのは次のようなものだったんでゲソ。

(1) F 演算子は、42 とか 0 じゃなく、特殊な値 (ここでは「限定継続値」と呼ぶでゲソ) を返すんでゲソ。
(2) この「限定継続値」は、この値を処理しようとした式を自分の中に飲み込んでいくというブラックホールみたいな代物でゲソ。
(3) ただし、この「限定継続値」に飲み込まれない演算子も 1 つ追加するんでゲソ。その演算子は、限定継続値に飲み込まれた式を取り出して評価を行うんでゲソ。

じゃあアイデアを順に説明していくでゲソ。ただしこの「限定継続値」については、Scheme の式の中で区別が付きやすいように $(...) とでも書くことにするでゲソ。

さてまず (1) でゲソが、どういうことなのか具体的に説明すると、「(F (lambda (k) 0)) という式は 0 じゃなくて $(lambda (k) 0) という値に評価される」、ということでゲソ [*45]。

つまりさっきの (a') 式は、一段だけ評価を進めると (exp $(lambda (k) 0)) になるんでゲソ。これなら (b') 式の (exp 0) と評価結果が違っても問題ないでゲソね。参照透明性は「同じ関数に同じ引数が適用されたら同じ値になる」という性質だったじゃなイカ。exp 関数の引数が違うなら評価結果が違っても参照透明性違反じゃないでゲソ。

でもそうすると今度は、指数関数の評価の方で困ってしまわなイカ? 指数関数は算術用の関数だから数値は引数に取れても限定継続値なんて定義域に入ってないでゲソ。それに、そもそも F 演算子が作る部分継続は、この指数関数を継続として捕捉しないといけないんでゲソ。だから指数の計算が行われちゃっても困るでゲソ。さて、どうするでゲソ?

ここで出てくるのが (2) でゲソ。まず、指数関数が引数として限定継続値を取れないなら、取れるように定義域を拡張してしまえばいいだけの話でゲソ。そして引数として限定継続値が与えられた場合には、(指数を計算する代わりに) その限定継続値に「指数を計算するという継続」を足し込んだ新しい限定継続値を返す、ということにすればいいんでゲソ。つまりこんな感じでゲソ。

```
(exp n)     = e の n 乗の値        ; ただし、これは引数が数値の場合でゲソ
(exp $(M)) = $(lambda (k)          ; 限定継続値だったら、新しい限定継続値を返すでゲソ
                (M
                  (lambda (n)  (k (exp n))))) ; この部分が exp を計算する継続でゲソ
```

[*45] ところで、モナドが好きな人は「F 演算子はモナドを作る演算子 (return)」なんだと思ってくれればいいでゲソ。

どうでゲソ? [*46] ちょっと読みにくいでゲソが、まるで exp 関数が限定継続値の中に飲み込まれたみたいに見えなイカ? [*47] というわけで、「F 演算子が指数関数を継続として捕捉する」という挙動が見事に実現できてるでゲソ!

もちろん、指数関数以外の関数についても同じように拡張を行うんでゲソ。これで、限定継続が周りの全ての計算処理を継続として取り込めるようになるでゲソ。[*48]

さて、これで参照透明性の問題は無くなったし、F 演算子が周りの計算を継続として取り込むという挙動も実現できたでゲソ。というわけで問題は全部解決したように思えるでゲソ?

ところが、まだ話は終わってないんでゲソ。というのも、限定継続値なんていう変な値だけ作れても困るでゲソ! そもそも、今回の (a') の例では最終的な評価値として 0 が出てきて欲しかったんでゲソ。今のままだとどこまで行っても限定継続値しか出てこないから 0 にならないでゲソ。

何が問題かというと (2) のルールだけだと限定継続値の中に継続を構築できるというだけで、実際にそれを使うことが出来ないんでゲソ。つまり、限定継続値の中に構築された継続を取り出す方法が必要なんでゲソ。というわけで (3) のルール、もっと具体的に言うと prompt 演算子の出番でゲソ [*49]。限定継続で二種類の演算子を使うのはこういう訳なんでゲソ。

で、prompt 演算子の挙動を説明するとこんな感じでゲソ。ちなみに、ここでは元の論文と同じく prompt 演算子を # 記号で表してみたでゲソ [*50]。

```
(# $(M))  = (M (lambda (x) x)) ; 限定継続値の中身 M を取り出し、
                               ;   さらに恒等関数を適用することで M を実行するゲソ
(# v)     = v                  ; ちなみに、引数が限定継続値じゃない場合は
                               ;   単にその値を返すだけでゲソ
```

じゃあ、この規則を入れたら (a') 式と (b') 式がどうなるかをもう一度考えてみるでゲソ。ただし、前のままだと (a') 式の方が限定継続値にしかならないので、どちらも prompt 演算子を付けて考えてみるでゲソ。

```
(a'') (# (exp (F (lambda (k) 0))))
      → (# (exp $(lambda (k) 0))) ; F 演算子が限定継続値を作り出すでゲソ
      → (# $(lambda (k)           ; exp 関数が限定継続値に飲み込まれるでゲソ
              ((lambda (k) 0)
                (lambda (n)  (k (exp n))))))
      → (# $(lambda (k) 0))       ; ちょっと限定継続値の中の式を整理しておくでゲソ
      → ((lambda (k) 0)           ; # 演算子が限定継続値の中身を取り出すでゲソ
         (lambda (x) x))
      → 0                         ; 見事 0 になったでゲソ!
(b'') (# (exp 0))
      → (# 1)
      → 1                         ; こちらも見事に 1 になったでゲソ!
```

うむ、見事に説明が付いたじゃなイカ!

[*46] えっ、関数の定義域や値域が恣意的で気持ちが悪い? うーん、でも例えば exp 関数って「無限ループする式」を引数に受け取ったら自分も停止しないでゲソ? それって exp(⊥) = ⊥ ということじゃなイカ。上の定義が気持ち悪いなら、これだって同じくらい気持ち悪くないでゲソ?

[*47] 正確には、限定継続値が飲み込むと言うよりは、指数関数が自発的に限定継続値の中に入っていくような感じでゲソが...

[*48] ところでモナドが好きな人は、(2) は (Haskell の do 構文とか Scala の for 構文みたいな感じで) モナドの bind 演算子とかが暗黙的に存在しているんだと考えてもらえばいいでゲソ。つまり、上の (a') 式は (return (\k -> 0)) >>= (\m -> return (\k -> k (m (\n -> exp(n))))) のシンタックスシュガーみたいなものでゲソ。

[*49] モナドが好きな人なら「モナドからその中身を取り出す演算子」だと思ってくれればいいでゲソ。ところで、限定継続が状態みたいなものなら、モナドの中身を取り出したりしたら危険な気もしなイカ? だけど限定継続が表してるのは、状態は状態でも「prompt 内だけの局所的な状態」なんでゲソ。同じように局所的な状態を表してる Haskell の State モナドなんかも中身を取り出せるでゲソ? だから限定継続も中身を取り出して問題ないんでゲソ。

[*50] 余談でゲソが、元論文では (Scheme っぽく書いたりはしてなかったので) prompt 式の見た目は "# ..." という感じだったんでゲソ。で、この # 記号が先頭に来る見た目が「なんだかコマンドラインのプロンプトっぽい」というので "prompt" という名前が付いたんでゲソ。ところでプロンプトが # ってことは普段 root で作業してたってことでゲソ?

じゃあ最後に、(この例だけだと捕捉した継続を使ってなくてつまらないので) 別の例も出しておくでゲソ。さっきの例に似てるでゲソが今度は捕捉した継続をちゃんと使う例でゲソよ。ちなみに今回捕捉する継続は (+ 1 (exp □)) といった式なので、これを 0 に適用して、評価値は (+ 1 (exp 0)) = 2 になるはずでゲソ。

(ところで + 関数がどういう風に限定継続値に飲み込まれるかは説明してないでゲソが、まぁなんとなく分かるでゲソ?)

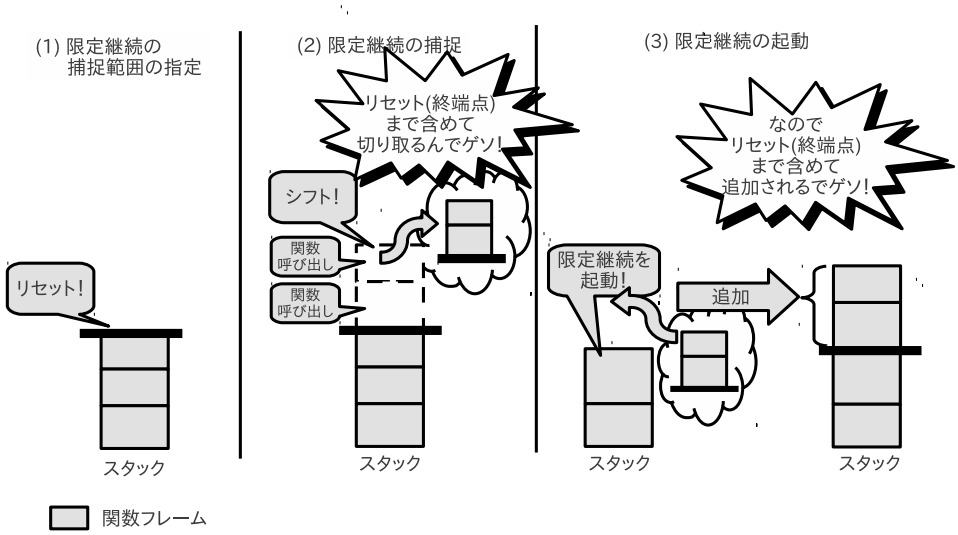
```
(c)   (# (+ 1
           (exp (F (lambda (k) (k 0))))))
      → (# (+ 1                        ; F 演算子が限定継続値を作り出すでゲソ
           (exp $(lambda (k) (k 0)))))
      → (# (+ 1                        ; exp 関数が限定継続値に飲み込まれるでゲソ
           $(lambda (k)
              ((lambda (k) (k 0))
                (lambda (n)  (k (exp n)))))))
      → (# (+ 1                        ; ちょっと限定継続値の中の式を整理しておくでゲソ
           $(lambda (k) (k (exp 0)))))
      → (# $(lambda (k)                ; + 関数が限定継続値に飲み込まれるでゲソ
              ((lambda (k) (k (exp 0)))
               (lambda (n) (k (+ 1 n))))))
      → (# $(lambda (k)                ; ちょっと限定継続値の中の式を整理しておくでゲソ
              (k (+ 1 (exp 0)))))
      → ((lambda (k)                   ; # 演算子が限定継続値の中身を取り出すでゲソ
           (k (+ 1 (exp 0))))
         (lambda (x) x))
      → ((lambda (x) x)
           (+ 1 (exp 0)))
      → 2                              ; うむ、バッチリじゃなイカ!
```

どうだったでゲソ? これでもう参照透明性の心配をせずに思う存分継続を捕捉できるでゲソね。

15.7 他の種類の限定継続も紹介しておくでゲソ

さっき紹介した control/prompt がオリジナルの限定継続の流儀なんでゲソが、その後で色々な限定継続が発見されてきたんでゲソ。その中でも一番有名なのが shift/reset と呼ばれる流儀でゲソ (ちなみに、呼び名の通り shift と reset という演算子を使うんでゲソ)。

また新しい演算子が出てきて面倒だと思ったかもしれないでゲソが、実は control/prompt と shift/reset の違いは 1 箇所だけなんでゲソ。下にあるのが shift/reset の動作を表した図でゲソが、control/prompt によく似てなイカ?



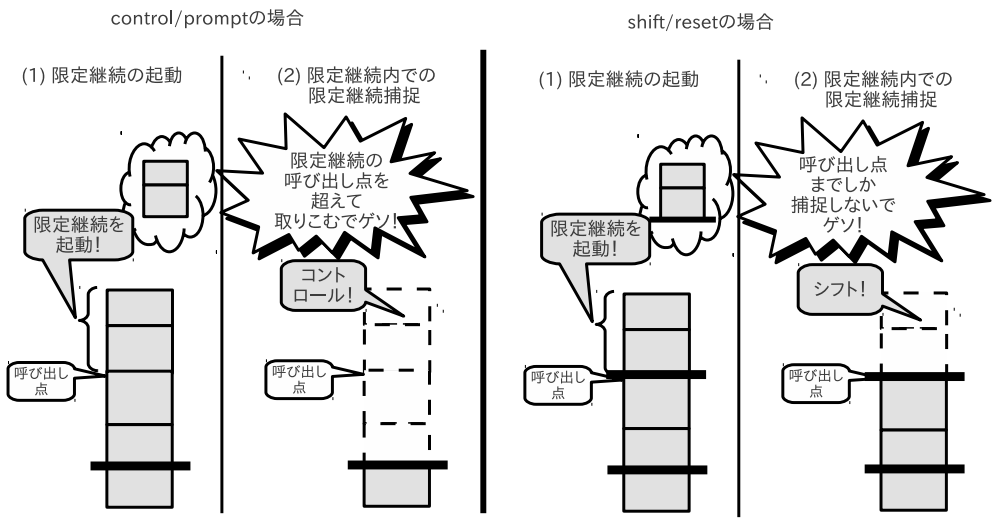
この 2 つの何が違うかというと、control 演算子では限定継続を捕捉する時に prompt 自体は取り込まなかったでゲソ? ところが、shift 演算子が限定継続を捕捉する時には reset まで含めて取り込むんでゲソ。

これでどういう違いが出るかというと、(捕捉した限定継続を呼び出した際に) その中でもう一度限定継続を捕捉すると違いが出るんでゲソ。次のページに図を描いたので、それも見ながら読んで欲しいでゲソ。

control/prompt の場合は、捕捉した限定継続自体には prompt が含まれないでゲソ? ということは、限定継続内で限定継続の捕捉を行うと (その限定継続内に他に prompt がなければ) 限定継続の呼び出し元の継続まで捕捉範囲に入るんでゲソ。

逆に shift/reset の場合は shift で捕捉した限定継続に reset が含まれるでゲソ? となると、起動した限定継続内で限定継続の捕捉を行っても、(仮に限定継続内に reset がなかったとしても) 最悪、限定継続の呼び出し元の地点で捕捉が止まるんでゲソ。

確かに、コルーチンとかを作る場合を考えると、コルーチンの呼び出し点で止まってくれた方がいいかもしれないでゲソね。限定継続を捕捉する操作は (その分の継続が消えちゃうから) 大域脱出を伴うことがあるでゲソ? コルーチン内で yield したらコルーチンの呼び出し元よりも前に大域脱出しちゃったりすると、ちょっと使いにくそうでゲソ。



とはいえ、これは control/prompt と shift/reset の戦力の決定的差ではないんでゲソ。というのも、control/prompt でこの挙動をマネするのは簡単だからでゲソ。実際問題として、control/prompt があれば shift/reset は簡単に作れるんでゲソ。

(ちなみに、上の説明で何となく気づいたかもしれないでゲソが、prompt と reset は全く同じものでゲソ。

違いがあるのは control と shift だけなので、control を使った shift の作り方だけを書いてみるでゲソ。)

```
(shift k M) =
  (control _k                          ; 限定継続 _k をそのまま渡す代わりに
    (M                                 ;   「prompt をセットしてから _k を呼び出す」
     (lambda (x) (prompt (_k x))))) ;   という関数を渡せば、shift 相当
```

一方、shift/reset から control/prompt は作れるか？ というと、Scheme のような形の無い世界なら作れることが分かっているでゲソ。一方、型のある世界では作れないことも分かっているでゲソ [*51]。

え、それじゃ shift/reset より control/prompt の方がいいんじゃなイカって? 確かに表現力だけ見ると control/prompt の方が良さそうでゲソ。だけど、shift/reset の良さは別の所にあるんでゲソ。

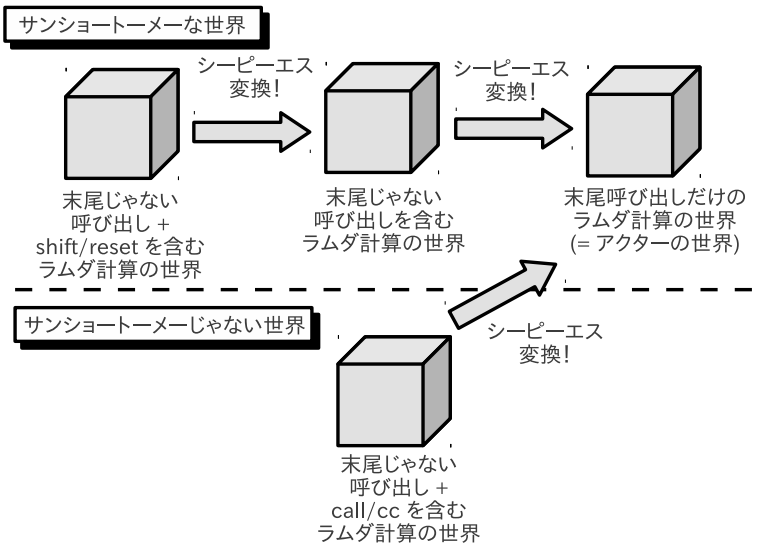
15.7.1 何で shift/reset の方がよく使われるんでゲソ?

shift/reset の良さは CPS (Continuation-passing style) と関係があるんでゲソ。

少し思い出して欲しいんでゲソが、限定継続って普通の関数のようにリターンしてくるじゃなイカ? で、「リターンしてくる継続」ってどういうことでゲソ?

(限定じゃない) 普通の継続は、前に説明したように「CPS で書かれたプログラムにおける末尾呼び出しされる関数」に対応してたんでゲソ。で、末尾呼び出しだから決して返ってこなかったんでゲソ。じゃあ逆に考えると、「CPS で書かれたプログラムにおける末尾じゃないところで呼び出される関数」って限定継続ということにならなイカ?

というわけで「CPS 変換しても末尾呼び出しにならない制御構造」を研究していたところ、見つかったのが shift/reset だったんでゲソ [3] [*52]。



で、こんな風に CPS 変換した世界に対応物があると色々と便利そうじゃなイカ? 例えば、control/prompt とか shift/reset の型を考えようと思ったとき、そのままだとよく分からないでゲソ?
でも CPS 変換で普通の関数に対応すると分かっていれば、その関数の型を手がかりに研究が始められるとい

[*51] 本質的に control/prompt の方が表現力が高いんでゲソ。なにせ control/prompt では「型がつくけど停止しない式」まで作れてしまうんでゲソ (カッコよく言うなら「キョーセーキカセー (強正規化性) がない」ということでゲソ) [8]

[*52] ところで「CPS なのに末尾呼び出しじゃない」ってなんだか自己矛盾しているようにも聞こえるでゲソね。というわけで少し整理しておくと、CPS は (スタックに積まれるリターンアドレスじゃなくて) 明示的に受け渡すクロージャーで制御フローを管理するやり方でゲソ? 逆に、末尾じゃない関数呼び出しはスタックに積まれるリターンアドレスで制御フローを管理するんでゲソね。じゃあ「CPS だけど末尾じゃない関数呼び出しを含むプログラム」はと言うと、「明示的に受け渡すクロージャーとスタックに積まれるリターンアドレスの 2 つで継続を管理する」ということなんでゲソ。つまり「2 種類の継続を用意しておいて使い分ける」というのが shift/reset の本質でゲソ。というわけで、明示的に 2 種類の継続をクロージャーとして受け渡すようにすれば、同じことを全部末尾呼び出しだけで書くこともできるでゲソ。それで、明示的に管理するのを 1 種類だけにしたいなら「CPS+末尾じゃない呼び出し」にすればいいし、両方とも明示的に管理したくないなら shift/reset を使えばいいだけでゲソ。

うわけでゲソ。それに実際に使う上でも、CPS な世界に対応物があれば (実行時にスタックをコピーする代わりに) コンパイル時に CPS 変換することで継続捕捉処理を実装できる、というメリットもあるでゲソ。

15.8 おわりに

さて、今回は「継続」について色々と話してきたでゲソが、どうだったでゲソ?

継続は強力な反面けっこう癖も強いので、普段はなかなか使いどころがないかもしれないでゲソ (正直、私もそんなに上手く使いこなせているとは言えないでゲソ...)。でも、ここぞという場面で上手く使えると (きっと) カッコいい (にちがいない) でゲソよ!

最近は、軽量スレッドがブームになったり Scala に限定継続が実装されたりで、継続が活用できる場面も増えてきているような気がするでゲソ。もしかしたら、来年あたりには継続を使った素敵な関数型のフレームワークが流行っちゃったりするんじゃなイカ (妄想)。というわけで、今から素敵な継続使いを目指して頑張ろうじゃなイカ!

参考文献

[1] JavaFlow: The Apache Software Foundation. `http://jakarta.apache.org/commons/sandbox/javaflow`.

[2] ABELSON, H., DYBVIG, R., HAYNES, C., ROZAS, G., ADAMS, N., FRIEDMAN, D., KOHLBECKER, E., STEELE, G., BARTLEY, D., HALSTEAD, R., OXLEY, D., SUSSMAN, G., BROOKS, G., HANSON, C., PITMAN, K., AND WAND, M. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11* (1998), 7–105. 10.1023/A:1010051815785.

[3] DANVY, O., AND FILINSKI, A. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (New York, NY, USA, 1990), LFP '90, ACM, pp. 151–160.

[4] DIJKSTRA, E. W. Recursive programming. *Numerische Mathematik 2* (1960), 312–318.

[5] FELLEISEN, M. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 180–190.

[6] FILINSKI, A. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1994), POPL '94, ACM, pp. 446–457.

[7] HEWITT, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence 8*, 3 (June 1977), 323–364.

[8] KAMEYAMA, Y., AND YONEZAWA, T. Typed dynamic control operators for delimited continuations. In *Proceedings of the 9th international conference on Functional and logic programming* (Berlin, Heidelberg, 2008), FLOPS'08, Springer-Verlag, pp. 239–254.

[9] KISELYOV, O. IO monad realized in 1965. `http://okmij.org/ftp/Computation/IO-monad-history.html`, 2012.

[10] LANDIN, P. J. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM 8*, 2 (Feb. 1965), 89–101.

[11] LANDIN, P. J. A generalization of jumps and labels. *Higher Order Symbol. Comput. 11*, 2 (Sept. 1998), 125–143.

[12] REYNOLDS, J. C. The discoveries of continuations. *Lisp Symb. Comput. 6*, 3-4 (Nov. 1993), 233–248.

[13] ROMPF, T., MAIER, I., AND ODERSKY, M. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not. 44*, 9 (Aug. 2009), 317–328.

[14] SRINIVASAN, S., AND MYCROFT, A. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), ECOOP '08, Springer-Verlag, pp. 104–128.

[15] STEELE, G. The history of scheme. In *JAOO Tenth Annual Conference* (Oct. 2006).

[16] STEELE, G. L., AND SUSSMAN, G. J. Lambda: The ultimate imperative. Tech. rep., Cambridge, MA, USA, 1976.

[17] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput. 13*, 1-2 (Apr. 2000), 135–152.

[18] WADLER, P. The essence of functional programming. Prentice Hall, pp. 1–14.

第16章

愚者のコントロール

— @tanakh

16.1 I

「わたし、気になります！」

やれやれ、また始まった。「やらなくていいことなら、やらない。やらなければいけないことなら、ギリギリまでやらない」がモットーの俺にとって、これは悪い予兆だ。子供のような無邪気な眼差しで四路戸えるに見つめられると、そんな信条とは裏腹に、どういう訳だかいつも面倒事に巻き込まれてしまう。

「私気になりません」

僅かばかりの抵抗を試みるが、最後には今回もこの眼差しに絡め取られてしまうのだろう。まったく厄介な役回りを引き受けてしまったものだ。

「いつもの遅延評価かい、奉太郎。そんなに無碍にしなくてもいいじゃないか。この謎、僕も気になるしさ」

複文里志はいかにも楽しそうだ。やめてくれ、俺はお前とは違う。ただ平穏な学校生活を送れればそれでいい。取り柄も何もない、普通の高校生なんだ。

「おはよう。……どうしたの、折具？」

部屋に入るなり川中摩耶花はそう言った。憂鬱な気分が入り口からも分かるほど顔に張り付いているのだろうか。だが、いよいよもって逃げられそうになくなってきた。このままはぐらかすのと、四路戸に答えを提示するのと、どちらが評価コストが低いのだろう。必ずしも正しい答えを提示する必要はない。ただ、彼女が納得する答えを導き出せばいいのだ。俺は覚悟を決めた。

「ああ、分かったよ……」

16.2 II

英国はエディンバラに留学している姉貴の言付けで、それなりに伝統ある古典論理部に入部したのがこの春のことである。そんなことでもなければ部活などという酔狂は、俺には無縁だったはずなのだ。高校に入学したとはいえ、特に何もやりたいこともない故、何をするかもよく分からない古典論理部なるものに入部したのがやはり何の間違いだったのか。

廃部寸前の古典論理部に入部しようという珍妙な奴がもう一人。部室に充てがわれている特別棟四階の地学講義室に入ると、そこに四路戸はいた。「豪農」四路戸家の一人娘にして、お嬢様らしい整った容姿。そしてそれに似合わぬ、旺盛な好奇心。答えを求めて俺を見つめる真っ直ぐな目には、何か逆らえぬものを感じた。

曰く、彼女の叔父は 40 年前のλ文字山高校古典論理部に所属していたらしい。そこで何やらとても悲しい出来事が起こり、彼は退学になってしまった。幼い頃その話を聞いてとても悲しい思い出のある四路戸は、40 年前の真相を確かめ、叔父の無念を晴らしたい、とまあそんな理由で古典論理部に入部したとのことだ。その謎を俺がたまたま解いてしまったもんだから、事あるごとに気になることを付きつけられることになるわけだが、その話は又の機会に譲るとしよう。

そもそも古典論理部とは何をする部なのか？ そもそもの話として、古典論理とはなんなのか。実のところ俺にもよく分かっていない。姉貴に言われるままに入部しただけなのだ。部長を務める四路戸にもそこのところが分かっているのかは疑問である。俺達は未だに部活動らしきものをしたことがない。今のところ判明しているのは、文化祭で伝統ある文集「評価」を出すことだけだ。一体、何のためにある部活なのだろうか。

16.3 III

「どうしてこのプログラムがうまく動かないのでしょうか？ わたし、気になります」

特に活動のない俺達は、部室に据え付けられた PC で、プログラミングをすることが当面の活動になっていた。特にそのことに理由や目的があるわけではない。たまたま部室に「Learn You a Haskell for Great Good!」なる書籍の変な表紙の和訳本が転がっていたから、手慰みにいじってみたに過ぎない。

Haskell というのは純粋関数型プログラミング言語というご大層な枕詞を冠するプログラミング言語で、遅延評価を特徴とする。「やらなくていいことなら、やらない。やらなければいけないことならギリギリまでやらない」俺の信条に遅延評価とやらがどういうことかぴったり合致したもんだから、Haskell を書くのは今のところそんなに億劫ではない。

「これ、昔の『評価』に書かれていた式なんですけど」

「へえ、評価ってプログラミングの記事も載ってたんだ」

里志が驚いてそう言った。俺もそんな話は初耳だったが、考えてみれば当然の話だ。俺は評価を読む必要に迫られていなかったから、読んでいなかったのだ。

「ええ、そうみたいですね。これ、このフィボナッチ数の計算なんですけど」

四路戸の手元の「評価」には、幾つかのプログラムらしきコード、それと「フィボナッチ数の定数時間アルゴリズム」というタイトルが見える。

「フィボナッチ数列はご存知ですよね。初項と第二項をそれぞれ 1, 1 と定義して、第三項以降を一つ前の項と二つ前の項の和と……」

「大丈夫だ、さすがにフィボナッチ数列は知っている。それより要点を手短に頼む」

「ちょっと折具、私はよく知らないんだけど」

「摩耶花には僕から説明しとくよ。それより四路戸さん、続きを頼むよ」

不満気な表情を見せる摩耶花を里志がなだめる。

「いいよ、ふぉーちゃん。続けて」

「あ、はい」

話を遮られていた四路戸が、コホンと口に手を当てて、改めて話し始める。

「この記事には、フィボナッチ数の第 n 項を求めるアルゴリズムが幾つか載っていました。最終的には第 n 項が定数オーダで求まるようになるようなのですが、わたしにはまだ理解できていません」

難しそうな顔をしてそう言う。と次の瞬間には急にしかつめらしい表情になって、

「それはともかく、記事の最初には指数オーダのプログラムが載っていたんです。それは再帰で書かれたとっても自明なアルゴリズムで、当然のことですけど、それは入力の大きさに対して指数関数的な時間がかかってしまいます」

ふむ、当然といえば当然だ。しかし気になるところが一つある。

「なあ四路戸」

「何でしょう、折具さん」

「さっき、第 n 項が定数オーダで、と言ったか？」

「はい、そうですが……」

そう言って、四路戸は「評価」に載っている数式を指さした。

$$F_n = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right\}$$

「フィボナッチ数の一般項がこれで表されるそうなんです。どうして整数のみの定義から平方根を含む式が出てくるのでしょうか？ とても不思議です」

「ああ、そうだな」

それはそれで面白い話だが、今回気になったのはそれじゃない。

「しかし、問題はそこじゃない。この式を用いても、その値は定数時間では計算出来ない。フィボナッチ数は n に対して指数的に大きくなることが知られている。大きな数をコンピュータで表現するには、それに見合う量のメモリが必要だ。整数 n を表現するのに必要なビット数はいくつだ？」

「ええと... $\log n$ でしょうか？」

「そうだ。指数オーダの数の対数を取れば結局元の値に対して線形のメモリが必要になる。入力に線形のサイズのメモリを出力するプログラムは必然的に...」

「線形時間を要する、というわけだね」

「ああ、そうだ。それに加えて、この式には累乗が含まれている。累乗は場合にもよるが普通は計算するのに対数時間がかかる」

「トータルでは $O(n\log n)$ ほど掛かりそうね」

「おそらくそうなるだろう」

四路戸はまたもしばらく難しそうな顔をしていたが、突然ハッとしたように顔を上げて、目を丸くしている。

「あっ、じゃあこの記事が間違っているのでしょうか？」

「これだけではなんとも言えないが――――」

当時の高校生が書いた記事だ。当然誤りが含まれることもあるだろう。

「計算量の議論を行うためには、計算モデルを予め仮定する必要がある。計算モデルによって計算量が変わってくるからだ。一般的には、定数時間でメモリにランダムアクセスできる『RAM モデル』というものを仮定することが多いが」

ふう、と一息ついて

「いずれにせよ、大きな数を扱うためには、どういう操作を定数時間で行えるのかきちんと定義する必要がある」

「へえ、なるほどね。奉太郎、いつの間に計算量の話に詳しくなったんだい？」

「詳しくなんかないさ、たまたま知ってただけだ」

感心したように四路戸はこちらを見つめる。

「で、お前が気になってたってのは、どういう話なんだ？」

「ええと、何の話でしたっけ？」

16.4 IV

「あっ、そうでした。メモ化の話です」

しばらくうんうんと唸って、ようやく思い出したかのように四路戸は言った。

「ちょっとまって、ふぉーちゃん。そんな話だったっけ？」

「あれ、違いましたっけ」

「プログラムが上手く動かないという話じゃなかったか？」

「あ、そうです。フィボナッチ数なんですけど」

四路戸には結論を先走り、間の説明を飛ばしがちなところがある。集中すると周りが見えていないとも言えるか。

「指数時間の次に載っていたのが、線形時間アルゴリズムなんです。折具さんの話を踏まえると、これも線形時間かどうかは怪しいのですが、とりあえずそのことは置いておきますね。それで、これがそのプログラムなんですけど」

```
int fib1(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    return fib1(n-1) + fib1(n-2);
}
```

```
int fib2(int n)
{
```

```
    int a = 1, b = 1;
    for (int i = 0; i < n; ++i) {
        int c = a + b;
        a = b;
        b = c;
    }
    return a;
}
```

「fib1 が指数アルゴリズムで、fib2 が線形アルゴリズムというわけだな」

「その通りです」

満足気に首を縦に振っている。表情がくるくると変わって、見ているだけで飽きない。

「fib1 は、フィボナッチ数列の定義をそのまま書き下した形、fib2 はループで実装した形です、それでここからなのですが」

ぐっと体を乗り出して説明を続ける。

「メモ化というのが次に登場するんです」

「なるほどね。ループというのは、ある意味好ましくない。再帰的な定義をそれを計算する手続きに変換しているわけだからね。指数アルゴリズムの方は、再帰的な、あるいは宣言的なとも言えるね。この実装の問題点は、同じ値を何度も計算するところさ。だから...」

「メモ化を使えば、宣言的な実装のまま、線形アルゴリズムにできるというわけね」

「その通りです」

```
int tbl[100];

int fib3(int n)
{
    assert(n < 100);
    if (tbl[n] > 0) return tbl[n];
    if (n == 0) return 1;
    if (n == 1) return 1;
    return tbl[n] = fib3(n-1) + fib3(n-2);
}
```

fib(4) の計算を考えてみる。fib(4) は fib(3)+fib(2) に展開されるので、次は fib(3) を求める。fib(3) は fib(2)+fib(1) に展開され、fib(2) は fib(1)+fib(0)=2 である。fib(1) は 1 なので、fib(3) は 3 ということになる。さて、今度は fib(2) だが、fib(2) は fib(3) を計算する過程で求めたはずだ。ナイーブな実装では、馬鹿正直に fib(1)+fib(0) と展開し、計算する必要がある。だが、fib(n) は引数 n に対して一意な値を返すのだから、本来は何度も計算する必要はないはずだ。最初に計算した時に、その結果を覚えておけば良いのだ。これが今回のプログラムに対するメモ化の目的だろう。

「そこで、これからが本題なのですが、メモ化と聞いて、折具さんは何か思い出しませんか？」

「ん...、特に思い当たるフシはないが」

「Haskell のグラフ簡約だね、四路戸さん」

「その通りです！ わたしはこの問題、Haskell なら楽に書けるのではないかと思ったんです。Haskell は評価した式をデフォルトでメモ化すると聞きました。遅延評価を実現するためですね。そこで Haskell を用いて、再帰的なフィボナッチ数のプログラムを書いてみたんです」

そう言って、エディタの画面を開いてみせた。そこには数行の小さなプログラムが書かれていた。

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

「定義をそのまま書き下した形ね」
「はい。fib の呼び出しがメモ化されるなら、これで十分な速度が出ると思ったのですが...」
四路戸は次に ghci の端末を開いてみせた。

```
ghci> fib 50
-
```

ghci は沈黙したままカーソルを点滅させていた。
「うーん、確かに気になるね。奉太郎」
里志はその様子を見て、顎に手を手をあてしきりにさすっている。言葉とは違い、表情は至って楽しげだ。
「四路戸」
「何でしょう？」
「まずはじめに言っておくが、グラフ簡約とメモ化は違う。グラフ簡約においてメモ化は行われるが、それは今回お前の期待するようなものではない」
「なるほど、そうでしたか」
四路戸はがっくりと肩を落とす。
「では、どうすれば正しくメモ化されるようになるのでしょう？ 純粋関数型の Haskell を使えば、簡単にできると思ったのですが.....。これがメモ化されないのなら、参照透過性とは何のためにあるんでしょう。変数を書き換えられない Haskell で、どうやってメモ化を行えばいいんでしょう。どうすれば——」
「ちょっと待て」
まくし立てる四路戸を遮って、俺は、また面倒になったものだと思った。しかし、今回も納得するまで帰らせてはもらえなさそうだ。なんとか手短に解説したいところだが。
「純粋関数型言語というものは、そんなに特別なものではない。評価戦略が違うだけだ」
俺は、自分で言ってから首を捻り、
「いや、違うな。評価戦略はプログラミング言語のカテゴリとは関係はない。たまたま Haskell が遅延評価を採用している、それに過ぎない。同じように、Haskell は参照透明であることを言語の大元となるコンセプトとして採用した。の二つは不可分なものでもないし、特に相関のあるものでもない。ただ、遅延評価を行う上では、参照透明であることは有利に働くことはある」
「うーん、ではどういう場合にメモ化が行われるのでしょう？」
「グラフ簡約では、同じノードの評価結果は二回以上評価されない。それを実現するために、Haskell では評価の結果を保存するようになっている」
「それで、先ほどの関数がメモ化されないというのは...？」
「グラフ簡約のノード、Haskell ではサンク（thunk）と呼ばれるものだが、それが関数呼び出しごとに違うものが作られるからだ。`fib 4` の評価は `fib 3` と `fib 2` のサンクを作って、`fib 3` の評価は `fib 2` と `fib 1` のサンクをそれぞれ作るが、ここで最初に作られた `fib 2` と二回目に作られた `fib 2` のサンクがそれぞれ異なるものになるということだな」
「どうしてそのようになっているのでしょう？」
「もちろん、そういうものを同じサンクにする実装も考えられる。参照透明のでは同じ引数に対して同じ値が返ることが保証されているからな。だが、そういうものをすべて記録していたら、呼び出した引数の数だけどこかにその結果が保存されることになる。保存された値は、それが後に利用されないことを確認することができないか、あるいはできたとしても難しい。つまりそのようにしてメモ化された値は、GC によって回収することが困難になる。プログラムの実行につれて、メモリ使用量が無尽蔵に増えていく処理系なんてのは、とても扱いづらいものになるだろう」

「ははあ、なるほど」
「もう一つ、そういったメモ化を実現するためには、ある引数での評価結果がすでに存在しているか、確かめなければいけない。だが、一般的にそういう操作を行うのは困難だ」
「ええと……、実行コストの問題でしょうか？ 例えば、ハッシュテーブルを用いるとか……？」
「ハッシュテーブルは利用できない。ハッシュ表を引くとき、どういう操作が必要になるかを考えてみてくれ」
「うーん、まずキーに対してハッシュ値を計算して、それからその値で配列を引いて……あっ、ハッシュ値ってどうやって計算すればいいのでしょう？」
「任意の型の値に対して、正しいハッシュ値を計算する関数がない。正しいハッシュ値というのは……」
「同じ値に対しては、同じハッシュ値になるということだね、奉太郎」
「そうだ。たとえそれができたとしても、次に問題になるのはハッシュ関数の品質の問題だ。ハッシュ関数の品質というのは……」
「ハッシュ値の衝突が少ないってことよね、折具」
「……そうだ。これらの理由から、ハッシュテーブルを用いてデフォルトのメモ化を実装するというのは良い選択とは言えないだろう」
「赤黒木などの、平衡二分木ならどうでしょう」
「今度はキーに全順序が付けられるという条件がつく。尤も、Haskell では全順序を持つ型、つまり Ord クラスのインスタンスは自動で導出できるわけだが」
「いずれにせよ、自動でメモ化が行われると、思わぬところで困ってしまいそうですね」

16.5　V

ひとまずは四路戸の納得する説明を与えられたようだ。
「それでだ」
漸く本題の方に入れそうだ。
「フィボナッチ数列のメモ化を行うにはどうすればいいのか」
「そうでした！」
すっかり忘れていたみたいだ。
「有名なコードはこんな感じのものだな」
俺はカタカタとキーボードを叩いて、次のコードを画面に写した。

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

「ああ、このコードなら僕も見たことがあるよ」
「Haskell の遅延評価と引き合いに出される、有名なやつよね」
さすがに有名なコードだ。
「でも、いまいちこれがどういう仕組みでうまく動くのか分からないのよね」
「このコードで言いたいのは、動作原理じゃないんじゃないかな。右にある `fibs` の呼び出しが、このコードではどうしてメモ化されるのか？」
「ご名答」
ghci に式を打ち込むと、今度はすぐに結果を返した。

```
ghci> fibs !! 50
20365011074
```

「あの、ちょっと待ってください」
四路戸が怪訝そうな顔をして、
「ここの右辺にある `fibs` の参照って、呼び出しなんでしょうか？ 変数の参照に見えるのですが」

「ああ、そうだったな」

もっともな質問だ。

「まず Haskell では、定数と 0 引数関数の区別がない。つまりそもそも関数と変数の区別はない。区別する必要がないからだ」

「ええと、参照透明性のおかげでしょうか？ 関数は引数の値にのみ依存して結果の値を返す…と言うことはつまり、0 引数の関数はいつも同じ値を返さなければいけないということですね」

「そういうことだ。それで、この” 関数呼び出し” が、関数呼び出しであると同時に単なる変数への参照でもあるわけだ。つまり、ここでの `fibs` への参照は、同じサンクへの参照になるわけで、最終的にはメモ化される事になる」

「なるほど。0 引数の関数ならメモ化されるというわけですね」

「実際の GHC は正確にはもう少し違う挙動になる。Lambda Lifting も考慮すると、トップレベルにメモ化されるサンクはもっと多くなるからな。GHC はこういう関数のことを “CAF（Constant Applicative Form）” と呼んでいる」

「折具さん、CAF というのは……？」

さすがにそこまで説明するのはめんどくさいし、今それは必要とされていないだろう。

「CAF については HaskellWiki に詳しい説明が書いてあるから、そっちを読んでくれ。とりあえず、自動的にメモ化されるのは CAF と呼ばれるものだけだということ、CAF は 0 引数の関数だということだ」

「なるほど、だいたい分かりました、ただ…」

「ただ……？」

「ただ、まだ少し気になるところが」

「何だろうか」

「はい、CAF というものがメモ化されるということは分かりました。でも、CAF が大きくなることはないんでしょうか？ 例えば先ほどの……」

そう言ってターミナルに式を打ち込む。

```
Prelude> fibs !! 100000
420269270299515438631900510129391513177391570263223450330471608719833
573145727622663393847726701366096253366170285832918664116229882221533
...
994892588234011350360387511421993302025047776808575499810068887160787
732953050111241465547976334636931551115834357110038285979669707537501
```

「あ、大丈夫でしたね」

ターミナルに数万桁の数字が表示されたのを見届けると、四路戸はそう言った。

「へえ、分かってはいたけど、本当に大きな数を扱えるものなんだね」

里志は少し驚いたような素振りを見せている。

「で、このメモリ使用量なんですけど」

四路戸は top を起動してメモリ使用量を探しているようだ。

「ええと……あ、ありました。これでだいたい 600MB 消費しているようです」

「そうだな、意外と食うもんだ」

「はい。それでこれが問題にならないのかなって」

「こんなに多くのメモリが、GC に回収できない状態になって、大丈夫かってことね」

「はい！ そうです」

「そうだな、確かに問題になることはある。だけど、基本的に CAF には大きくなり続けるデータを置かない。つまり、この `fibs` のようなものは普通はやらないし、やることはあまり良くない」

「へえ、そうなのかい。僕はてっきりこういう書き方をするのが良いものだと思っていたのだけど」

「メモリが増え続けるバグっていうのは、他の種類のバグの入り込みにくい Haskell にとっては相対的に大きな問題になる。そういう時にはヒーププロファイラなんかを使って問題を特定することになるんだが、大

きな原因の一つが、遅延評価に伴ってサンクが成長し続けるスペースリークで、もう一つの大きな原因が」
「CAF が成長し続けるってことね」
「ああ、そうだ」
「……なるほど、だいたい分かりました。ではメモ化をやるうまい方法は」
「遅延評価をうまく利用しつつというのは難しいだろうな」
「そうですか……」
そう言って四路戸は俯く。残念だという風だが、ともあれ納得はしてくれたようだ。
「しかし、意外だね」
中空に式を書くように、指をクルクルさせながら里志が言う。
「何が意外なんだ？」
「奉太郎がここまで Haskell を勉強していたとはね。必要じゃないことはやらない。必要なことも、必要になるまではやらない、いつもの奉太郎はどこに行ったんだい？」
「たまたまだ。偶然詳しかっただけで」
「……そうかい。まあいいさ、僕は奉太郎が終に無気力な学生生活に別れを告げて、自分の使命を全うし始めたのかと思ったのだけどね」
「買い被り過ぎだ」
俺は PC の電源を落とし、帰り支度を始めた。

16.6　VI

λ須先輩の誘いを断りきれず、どういうわけだかこうして二人で喫茶店にいるのは、それから幾日も経たない日のことだった。
「—— 君の高名は聞いている。随分とプログラミングが得意のようだね」
「そんな、たまたまです」
面倒な事になった。λ文字山高校理科部のλ須冬実、先輩の部ではこの夏に開催される　UltraCon なるプログラミングのコンテストに参加する予定だったらしい。ところが、プログラムの提出期限も間近というところで、メインで書いていたプログラマが失踪してしまったらしい。プログラムはバグでうまく動かない。それで、残されたプログラムの完成を古典論理部に依頼してきたというわけだ。
「たまたまというのは、そう何度も続くものではないだろう。折具、私は君が非凡な才能を持っていると思っている」
「買い被り過ぎです」
「そうだろうか。君の活躍は四路戸から聞いている。そこから判断した、私が保証しよう」
本当に俺は、特別な才能を持っているのだろうか。平凡な高校生だと、たまたまだと思い込もとしていただけなのだろうか。
「才能を持つ者は、それを正しく利用しなければならない。それが才能を持たない者への責務だ。—— この話、考えておいてくれ」
λ文字山高校の「女帝」は、そう言って立ち去った。

16.7　VII

「引き受けることにした」
「えっ、どういう心境の変化なんだい？」
「特に理由はないさ」
「へえ、まあ詳しくは聞かないよ」
里志は肩を竦めた。
さて、今回のあらましはこうだ。λ須先輩の部では　UltraCon に参加するために、まずその予選の改題を提出しなければならなかった。予選課題とはいえ、プログラムには正確さとしっかりとした速度が求められる。そのためにプログラムは Haskell で書かれる事になった。しかし締め切りも間近となった昨日、提出するためのプログラムを書いていた生徒が失踪したというわけだ。
提出するプログラムを任せられる部員が他にいなかったため、俺のことを聞きつけたλ須先輩によって、古典論理部にお鉢が回ってきたのだ。そもそもそういうプログラムを部外者に任せていいものなのか、よく

わからないのだが……。

「それで、どういった課題だったのでしょうか？」

いつの間にかやってきていた四路戸が、俺の背後から PC のモニタをのぞき込んでいた。依頼を受けた際に、課題と完成途中の課題の一式をあずかっていた。プログラムをエディタで開く。

しばらくプログラムを眺めていると、

「このプログラム、どこが未完成なんでしょうか？ 一見するとそういったところは見当たらないようですが……」

「えーと、このプログラムは」

コンパイラにかけてみる。エラーも出ずにコンパイルは成功した。適当な入力を試してみると、そのプログラムは期待通りの答えを返す。

「ふむ、うまく動いているように見えるね」

「ところがだな」

俺は適当なプログラムを書き始めた。

「何を書かれているんでしょう？」

「大きな入力を生成するプログラムを書いているんだ」

しばらくしてプログラムを書き終えると、それが生成したデータを、λ須先輩から受け取ったプログラムに入力してみた。

「……うーん、全然ダメね」

「なるほど、実行速度がいけないんですね！ でも、このプログラムは一体どこが……」

つまり、こういうことだ。正しく動作するプログラムはできているのだが、見たところ満足な速度では動作していない。λ須先輩から聞いたところによると、これを書いた生徒には、パフォーマンスに関してはあまり心配していなかったという。ここから僅かの変更で十分な速度を達成できる公算があったということだろう。

「この前のサンク、評価するときが案外早く来たのかもしれないな」

16.8 VIII

渡されたプログラムは、比較的単純な部類に入る。一通り読めば、どういうことをやっているのかは簡単に理解できた。つまり、

「この探索問題を解くために、一番単純な深さ優先探索でアルゴリズムが実装されているようだな」

「へえ、でもそれじゃあ、速度が遅いのは当たり前じゃないか」

「そうね、それでどう改良するつもりだったのかしら」

「それを考えるのが、俺達の役目だというわけだ」

古典論理部に改善を依頼したといっても、元のプログラムの意志は継いで欲しいと。プログラムを書いたやつの意図を、そのプログラムから読み取って、見事プログラムを完成させる。そんなことが、俺にできるのだろうか。

「なんでも、このコードを書いた生徒は、GHC のバージョンにえらくこだわっていたようだが、それが何に影響するのかは今のところよくわからない」

「最適化、正格性解析が新しい GHC だとうまくいくとかだったりして？」

「そこまでこだわりが必要なところだろうか」

まあ、案ずるより産むが易しだ。

「ところで、理科部の何人かもプログラム改善のアイデアを出してきたらしい。まずはそれを参考にして欲しいということだ」

「折具さん、それを早く言ってください！」

16.9 IX

「なるほど。大雑把に言うと、このプログラムは基本的に探索を行うもので、探索空間はそんなに広くない。だから、メモ化すれば高速に動作するはずだ。そういうわけだね」

しばらく全員でファイルに目を通してから、里志が口を開いた。

「まあ、そういうことになる。これを書いたやつも、おそらくメモ化をしたかったんだろう。だが、肝心の

その方法が...」

「わからないというわけね」

そうなのだ。Haskell ではメモ化は自動的には行われない。かと言って、適当なテーブルに関数の結果を書き込むのは実は Haskell では自明ではない。Haskell は純粋関数型言語、参照透明だからだ。普通の関数がその中で結果をキャッシュするということはできない。

「テーブルへ書き込むには、関数を IO モナドにする必要があるね」

「ST モナドでも大丈夫ですよ」

「そうだな。いずれかにせよ、結果をメモリにキャッシュするという操作は参照透明性を破る副作用に相当する。つまり元の関数の形をかなり大きく崩さなければならないことになる。それはあまり綺麗とは言わないのではないだろうか」

「コードの綺麗さか。意外だね。奉太郎がそんなことを口にするなんて」

「他のものは知らんが、プログラムを綺麗に保つというのは、それなりに重要なことだ。プログラムの理解しやすさといじりやすさは、そのままメンテナンス性に繋がるからな」

「でも、プログラムの綺麗さというのは、なんだか曖昧な気がします。何を綺麗で、何をそうでないのかというのは、とても主観的な基準だと思います」

たしかに、そういう側面もあるだろう。俺は首を捻りながら、

「そうだな、しかし実際に利益をもたらすものだというと、そこに求められる性質も自ずと普遍的になる。少なくとも考えられるのは、プログラムが抽象的であることだな。同じコードの複数回の繰り返しはひとつにまとめたい。そのまとめ方が無理なく自然に行えていることが、プログラムが綺麗だといえるだろう」

「それで、IO モナドは良くないという話かな」

「IO モナドというよりも、メモ化という操作それ自体を抽象化したいという話でしょうか？ IO を使ってテーブルに書き込む操作は、何も抽象化されていません。メモ化したい関数ごとにすべての操作を手で記述する必要があります」

「そういうことだ。わざわざ探索プログラムを Haskell で書こうというやつが、そういう手法を用いようとしていただろうか」

なんだか話が逸れて来たので、本題に戻すとしよう。

「では、順番に見ていこうか」

16.10　X

「...つまり、このプログラムは DP をしようってのに、単なる深さ優先になってるってことだろ？ なら要するに、メモ化をすればいいってことだ。

どうやってメモ化をするのかって？ なんでもいいんじゃないのか、とりあえずうまく動きさえすれば。例えば、CAF にリストを置くとかな。CAF に置かれたものはグラフ簡約の同一のノードへの参照としてキャッシュされることになるんだったな。全部の答えを格納するリストをグローバルに置いときゃいいんだ。例えばだな、簡単のためにフィボナッチ関数のメモ化で考えてみると、

```
fibs :: [Integer]
fibs = map fib [0..] where
  fib 0 = 1
  fib 1 = 1
  fib n = fibs !! (n - 1) + fibs !! (n - 2)
```

こうなる。フィボナッチといえば、有名なあのコードがあるが、それはひとまず忘れてくれ。`fibs = map fib [0..]` これですべての答えの入ったリストがいっちょ上がりだ。あとはいつもどおりに関数を書けばいい。気をつけるのは、再帰呼び出しの代わりに、この『アカシックレコード』から答えを引っ張ってくることだな。依存関係が DAG になっていれば、この問い合わせはループしないことが保証される。どうだ、シンプルだし、これで文句ないだろ。

GC 対象にならない？ 知ったこっちゃねぇよ。こんなライフタイムの短いプログラムのメモリ増加を気に

する必要があるか？ リストのアクセス時間？ そんなのは指数時間に比べれば無視できるんじゃないか。

とにかく、もう締め切りまで時間がない。俺はこれで十分だと思うぜ」

プログラムとともに受け取ったファイルの中に、理科部で行われた議論の様子を収録したビデオが入っていた。俺達はまず “1.avi” なるファイルを鑑賞することにした。ビデオは “3.avi” まであるようだ。部内でこんなに案が出ているなら、部外者である古典論理部にわざわざ依頼しに来るという理由は何だろうか？

「なんだか、大雑把そうな方でしたね」

その一つ目の動画では、えらくいかつい男が、自分の手法をまくし立てていた。

「プログラムなんてものは、なんだかんだ言っても、正しく動くことが最重要だからね。それで、奉太郎。このアイデアはどうなんだい？ 彼も言うように、このアイデアには特に問題がないと言えるかい？」

そう言って里志は何か試すような視線をこちらに投げかけてくる。暫し考えた俺の答えは、

「……却下だ」

「何がいけないの、折具？ 私には特に問題なく感じたんだけど」

「引数が整数引数一つに固定、しかも小さい範囲でなければいけないところでしょうか？」

首を傾げながら四路戸が尋ねる。

「いや、それはそこまで問題にはならないと思う。二引数の関数、そうだな、簡単な例として LCS を実装してみようか」

「LCS とは何でしょう…？」

「Longest-common subsequence、日本語だと最長一致部分文字列かな。二つの文字列が与えられた時に、両方に部分文字列として含まれる文字列のうち、最長のものを求めるという問題さ、例えば…」

里志がカタカタとキーボードを叩き、例を示す。

「文字列 “hello” と “haloo” があったとする。これらの部分文字列として一番長いものは、“hlo” という 3 文字の文字列になるよね。これを求める問題というわけさ」

「なるほど、部分文字列というのは、元の文字列から順序を変えずに部分集合を取り出したものという意味ですか」

納得した様子で、四路戸は頷いている。確かに部分文字列という表現は若干曖昧なところがある。少なくとも、今回考える LCS での部分文字列の定義というのはそういうものになる。

「ところで LCS には、それぞれの文字列の長さを m、n として、O(mn) の効率的な DP を用いた解放が存在する。これは、m × n のサイズのテーブルを用いることになるが、再帰とメモ化を用いて実装すれば、二引数の関数をメモ化する問題と同じくなる」

「なんだかイメージが湧かないわね」

摩耶花はあまりこういうアルゴリズムには明るくなかったか。古典論理部の復活のために、図書部と漫画部との掛け持ちで、こんな話にまで巻き込んでしまっては何か申し訳ない気分になってくるが。

「難しく考える必要はない。最初は適当に再帰的な解法から始めればいいんだ」

俺は簡単な LCS の解法を書き始めた。

```
lcs :: String -> String -> Int
lcs [] _ = 0
lcs _ [] = 0
lcs (x:xs) (y:ys)
  | x == y = 1 + lcs xs ys
  | otherwise =
    lcs (x:xs) ys `max` lcs xs (y:ys)
```

「見ての通り、何でもいいから答えを出すコードを書けばいいならとても簡単だ。再帰的なアルゴリズムで、規定ケースはどちらかの文字列が空文字列の時で、これは明らかに 0 になる。一般のケースについては、先頭文字が一致してい残りの文字列で再帰、一致していないときは、どちらかの先頭を捨てて、良い方を採用する」

「でも、このコードは一番下のケースのおかげで、指数関数アルゴリズムになっているというわけですね！」

「ああ。だが、これは明らかに同じ計算を何度も行なっている。つまり、同じ引数については必ず同じ値になるはずだということだ。そこでメモ化が使える」

```
lcs :: String -> String -> Int
lcs a b = tbl !! 0 !! 0 where
  m = length a
  n = length b
  tbl = [ [ f i j | j <- [0..n]] | i <- [0..m] ]
  f i j
    | i == m = 0
    | j == n = 0
    | a !! i == b !! j = 1 + (tbl !! (i + 1) !! (j + 1))
    | otherwise =
      (tbl !! (i + 1) !! j) `max` (tbl !! i !! (j + 1))
```

続いて俺はメモ化を行うように改変したコードを端末に打ち込んだ。

「見ての通り、さっきのコードを素直にメモ化したものだ。元々の関数は `f` に相当する。再帰呼び出しで f を呼ぶ代わりに、すべての答えが格納されているはずの `tbl` を参照しているところがやはり違うが」

「へえ、二引数以上でも素直に実装できるんだね」

「これは `fib` での方法と比べて、別の利点もある。今回は `tbl` が CAF ではないということだ」

「えっ、CAF じゃなくてもメモ化されるんですか？」

「グラフ簡約としてのノードが等しければ、計算結果はキャッシュされる。CAF はその判断材料が必要ないということだな」

「つまり、`tbl` は一連の `f` を計算している間、常に同じ物が参照されるというわけだね」

「ああもう、よくわからなくなってきた！」

グラフ簡約と CAF がよくわかっていないとここの簡単なルールは見出しにくいのかもしれない。

「ともかく、ここで言いたいのは、この関数のしかる評価が終了した後に、`tbl` の指すリストは速やかに GC 対象になるということだ」

「たしかに、CAF じゃなくなるなら、当然そうなるね」

「今回のケースは引数によって `tbl` の値が異なるから、こういう形のコードにするのは必然だ。しかし、`fib` の場合でも、同じアプローチによってテーブルを CAF でなくすることは出来る。このあたりは実装の簡便さとパフォーマンス要求のトレードオフになるだろうな」

「んん、じゃあ結局折具はこの方法のどこが問題だというのよ？」

CAF の問題は解決可能だ。だから、問題は他のところにある。

「問題の一つは、計算量のオーダだ。確かに指数オーダと比べれば無視できるほどには小さいが、多項式時間だから良いというわけではない。$O(n^2)$ と $O(n^3)$ は現実には違いが大きい。実際のところ、リストの要素アクセスが入ると、本質的ではないところで一つオーダが上がってしまう」

「リストを使わない、例えば配列を使うといったことはできないんでしょうか？」

「配列を使う方法も考えられる。望ましいことに、メモ化のキャッシュ内容は一度決まると変更されることはない。計算の進行とともに、計算の結果が決まっていくのに、遅延評価を利用すれば、あたかも初めから定数として答えが存在していたかのようにコードを書くことが出来る。つまり、答えをすべて保持している immutable な配列を作るだけでいい。計算の順序やら、メモリへの格納はすべて Haskell の処理系が自動でよろしくやってくれる。ある意味でとても宣言的なアプローチだとも言えるだろうな」

「じゃあ、これは別段大きな問題にはならないんでしょうか？」

「配列を使えばそうなるな」

「それでもやっぱり問題はあるのかい？」

「残りの問題の一つは、これでもやはり小さい範囲の整数しか扱いにくいということだ。それともう一つ、俺が気になるのは、メモ化されたプログラムの導出の問題だ。それがいかに容易に、機械的に導出できるも

のだったとしても、やはり人間が関数ごとに手を動かさないといけないのは抽象化の限界と言えるだろう」

そこまで言ってから、ふうと息をつく。

「というわけで、この案は却下だ」

16.11 XI

「さて、俺の番か。メモ化をするんだったな。メモ化なんてのは、本来プログラムの変更なしで出来るようになるべきものだ。例えば Ruby なんかには、関数を受け取って、その関数の定義を書き換えて、自動でメモ化バージョンに変えるものがあるが、参照透明な Haskell ではそういうことは不可能だ。

ここは関数型言語であるところを逆手に取って、fix を活用出来るんじゃないか。fix は普通は再帰を行うために導入されるプリミティブだが、fix の代わりに別のものに関数を渡せば、元の関数の動作を変えられるということでもある。与えられた関数をメモ付き再帰版にする memoFix という関数を作るのが考えられるね。

```
fix :: (a -> a) -> a
memoFix :: (a -> a) -> a
```

こんな関数ができたとすれば、fib はこう書ける。

```
fib_f f n = case n of
  0 -> 1
  1 -> 1
  _ -> f (n - 1) + f (n - 2)

fib     = fix fib_f
fibMemo = memoFix fib_f
```

fix の扱い方に慣れていればなんということはない。fib が通常の再帰を行うバージョン、fibMemo がメモ化再帰を行うバージョン。メモ化機能を追加するために、元の関数を書き換える必要がないというところが重要なところさ。

さて memoFix の実装だけど、計算済みならそれを返して、そうでなければ計算してそれを返すというのを書きたいんだけど、生憎の参照透明だ。ここは『アカシックレコード』に再登場願おう。関数のすべての引数に対して、その評価値を記録しているテーブルだ。

```
memoFix :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
memoFix f = g where
  g n = tbl !! n
  tbl = map (f g) [0..]
```

見ての通り実装はとてもシンプルだ。引数が Int に固定されてしまったが、これは Ix a 型に簡単に変更できるだろう。引数の範囲を大きくできない問題は、アカシックレコードをテーブルからツリーに変更してやればいい。ツリーといってもとても簡単なものでいい。すべての整数に対してツリーを作るんだ。いわば『完全』『平衡』『無限』二分木を作ればいい。つまり、

```
data Tree a = Node a (Tree a) (Tree a)

mkTree :: (Int -> a) -> Tree a
mkTree f = g 0 where
  g i = Node (f i) (g $ i * 2 + 1) (g $ i * 2 + 2)
```

```
lookupTree :: Tree a -> Int -> a
lookupTree = g where
  g (Node v _ _) 0 = v
  g (Node _ l r) i
    | odd  i = g l ((i - 1) `div` 2)
    | even i = g r ((i - 2) `div` 2)
```

これで、与えられた関数の答えをすべて含む無限に大きい木が完成する。しかもすべての値へのアクセスが $O(\log n)$ で完了する。これらを組み合わせると、

```
data Tree a = Node a (Tree a) (Tree a)

mkTree :: (Int -> a) -> Tree a
mkTree f = g 0 where
  g i = Node (f i) (g $ i * 2 + 1) (g $ i * 2 + 2)

lookupTree :: Tree a -> Int -> a
lookupTree = g where
  g (Node v _ _) 0 = v
  g (Node _ l r) i
    | odd  i = g l ((i - 1) `div` 2)
    | even i = g r ((i - 2) `div` 2)

memoFix :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
memoFix f = g where
  g n = lookupTree tree n
  tree = mkTree (f g)

fib_f f n = case n of
  0 -> 1
  1 -> 1
  _ -> f (n - 1) + f (n - 2)

fib = memoFix fib_f
```

ふむ、案外いいのができたんじゃないかな？」

「いかにも偏屈そうな方が出てきましたよ！」

ビデオに登場した生徒は、何度もメガネをクイとやりながら、滔々と説明をしていた。正直、こういった抽象化は、関数型言語としてはとても美しいとは思うのだが……。

「あの、折具さん。`fix` というのはどういったものなのでしょうか？」

ああ、そうだったな。当然のように `fix` が登場していたが、余程なラムダ計算オタクでないと普通はプログラミングに直接用いるものではない。

「`fix` というのは不動点演算子のことだな。不動点演算子というのは、その名の通り与えられた関数の不動点、つまり fix f = (f x == x になる x) を返す関数だ。型理論では、単純型付きラムダ計算にチューリング完全の力を与えるために追加するプリミティブとしても重要な役割を果たすものだ」

「ええと、折具。ちっともわからないのだけど……」

そうかもしれない。しかし、今回の話はそこが焦点ではない。

「ともかく、関数をメモ化するプリミティブを追加しようという案だということだ」

「うーん、あとで図書館でしらべとくわ……。ラムダ計算の本だっけ？」
摩耶花はイバラの道を歩むことになりそうだ。
「で、奉太郎」
そうだった。
「却下だ」
勿体も付けずにそう言った。
「へえ、どういうところがお気に召さなかったんだい？」
「技巧的すぎる」
端的に言うと、そういうことだ。
「現実的な問題としてはまず、プログラミングモデルとして、fix ベースのものを要求される。これは慣れていない者にとっては書きづらいところがある。引数の数を増やすのも大変だ。現実的には引数の数ごとに memoFix1、memoFix2 などを用意することになるだろう。ここで問題になるのは、メモ化に利用しない引数を追加するのがとても大変だということだ。まあ、この辺は Template Haskell でも使えばなんとかなるのかもしれないが、それは…」
「抽象化の敗北というわけだね」
「ああ、そうなるな」
俺の言いそうなことを見透かされているのが多少気にかかるが、
「それはともかく、やはり技巧に過ぎるものは扱いづらいという一面はあるだろう」

16.12 XII

「えっと……最後はあたしだね。
プログラム速くしたいんだよね？ だったら、遅いところを C で書けばいいんじゃない？ メモ化…だっけ？ C 言語だったらメモリだって、ドガッ、バキッって書き換え放題だよね。
せっかく FFI があるんだしさ、まどろっこしいことする必要はないよ。どーんと書き換えちゃえばいいんだって！」
「な、なんかすごい人が出てきましたよ！」
λ須先輩、最後にとんでもないのをぶつけてきたな。
「却下だ」
とは言いつつも、それが最善であるケースは常に考慮しなければならない。

16.13 XIII

「……では、うちの部のアイデアは三人とも却下というわけだな」
「はい、残念ながら」
それから数日後、何時ぞやの喫茶店で、俺はλ須先輩に結果を告げていた。
「それぞれ確かに、問題を解決する手段ではあります。しかし、どれも抽象化と速度を両立するものではありませんでした。果たしてそれが元のプログラマの意図かどうかはわかりませんが、抽象化を抜きにしてプログラミング技法を語るのは難しいでしょう」
「ふむ、なるほどな」
λ須先輩は長い前髪の隙間から覗く鋭い目で、こちらをじっと見つめている。
「では聞こう。君ならあれをどうやって料理する？」
「はい、……」

16.14 XIV

夏休みも近づいた、ある日のことである。
「理科部の皆さん、予選通過されたようですね」
「そうか、それは良かったな」
「本戦はどうされるんでしょうか？」
さあな、λ須先輩が何とかするんじゃないか。
「ところで奉太郎」

「なんだ？」
「結局どういう風にしてメモ化の問題を解決したのさ？」
「あ、そうでした！ 聞いてませんでしたね。わたし、気になります！」
そういえばそうだったな。とは言っても、何も大それたことはないのだが。
「モナドだ」
「なんだ、結局モナドなのかい？」
「ああ、Haskell と言ったらモナドを使うのは普通だろう。むしろ三人の中にモナドを使おうとするやつが出て来なかったのが意外なくらいだ」
「折具、あんた最初 IO や ST 使うの真っ先に否定してたじゃないの？」
「いや、そうじゃない。ここでいうモナドはそういうモナドのことじゃないんだ」
面倒だが、ここまで来たら最後まできっちり説明するとしようか。
「そもそも、モナドというものはどういうものなのか。参照透明の言語副作用を伴う IO を行うために導入されたという説明がなされることもあるが、それはモナドの一側面にすぎない」
「へえ、じゃあ奉太郎はモナドをなんだと考えているんだい？」
「計算の抽象化」
左の中空をぼんやりと眺めながら考えを巡らせる。
「あるいは、計算のコンテクスト。DSL そのもの。はたまた……」
「折具さん、話が抽象的すぎて…」
「ああ、済まない」
話が飛んでしまった。
「つまり、IO モナドというのは、副作用を伴う計算だということだ。ST モナドは同様に、副作用をメモリ操作に制限したモナド。リストモナドは少し変わって非決定計算のモナド、State モナドは状態を持つ計算のモナド。Error モナドはエラーを扱える計算のモナド」
そして、勿体をつけて、
「今回の場合は、計算結果がキャッシュされるモナドを作ればいいというわけだ」
「モナド……そんなことも出来るんですか！？」
四路戸は驚いたような手振りで俺に問う。
「モナドに不可能はない」
何やら柄にもない言葉が口をついて出たが、実際モナドはどんな計算でも表現できるはずだ。
「で、奉太郎は、モナドを使ってメモ化を行ったというわけかい」
「そうなるな」
端末を開き、この前と同じく、fib を書いていく。ただし今度はモナドを用いて。

```
fib :: MonadMemo Int Integer m => Int -> m Integer
fib 0 = return 1
fib 1 = return 1
fib 2 = (+) <$> memo fib (n - 1) <*> memo fib (n - 2)
```

「MonadMemo という型クラスは、メモ化モナドを一般化した形だ。MonadMemo k v m に対して、k がキャッシュするキーの型、v が値の型、m が親のモナド型になる。fib がモナドになった事によって、呼び出し方が若干変わっているが、これは細かいことだ。要するにここで何が重要かというと、メモ化を行う手段を抽象化しているということだ。例えば MonadMemo のインスタンスとして、全くメモ化を行わないという実装を考えることが出来るだろう。あるいは、DB を用いて永続化を行うものも考えられる。一般的な実装は、Ord を持つキーに対する平衡二分木だろうか。何れにしても、行う計算と、それをキャッシュする術が完全に分離されているということは、プログラムの composability としてとても重要なことだ」
「他の方法では、それはできないのかい？」
「できないということはないだろう。ただし、その抽象化を行うためのインターフェースを決めるのはひどく難しいと思う。あるいはモナドのインターフェースに類似したものになるだろう。すでに使用例が多く、

十分に性質が研究されている、モナドというインターフェースでこれを抽象化するのは、多くの研究にタダ乗りできるという点でも、とても便利なことなんだ」

「もともとモナドで書かれていないコードをモナドでのコードに直すのは、大変じゃなかったんでしょうか？」

「基本的に、まったくの純粋なコードをモナドのコードにするのはとても簡単だ。なにせ、return をつけるだけでモナドになるんだから。注意するのは再帰的な部分で、これは若干のコード変更が必要になる。その際にはアプリカティブファンクターと呼ばれるものを利用するとコードの見通しが大幅に良くなるだろう」

さて今度こそ一通り説明が終わった。四路戸は画面を見つめて、カチャカチャとコードをいじっていたが、しばらくしてスクリとこちらを向き

「なるほど...モナドでメモ化を抽象化するなんて、思いもよらなかったです。大変勉強になりました。どういう時に、モナドを使えばいいのか、指針のようなものがあればわかりやすいのですが.....」

「うーん、そうだな.....」

少し考えて、

「モナドによる抽象化は、Haskell プログラミングの醍醐味だ。何かを抽象化したいと考えた時、まずモナドを検討するぐらいでいいんじゃないか」

「何でもモナド、というわけかい」

里志は楽しそうに言った。まあそういうことになるんだろう。

しかし、今回の件も漸く一件落着かといったところで、何かに気づいたような摩耶花が口を開いた。

「ちょっと待って、折具。何かおかしくない？」

何だろうか。何かを見落としてたか...？

「この話、GHC のバージョンに関係ないじゃない」

16.15 XV

これ以降の話は俺にとってはどうでもいい話ではあるが、いつもの調子で事の真相が気になった四路戸がλ須先輩を問い詰めて判明した事実である。

失踪した生徒がもともと想定していた解法は、テンプレート Haskell を用いるものだったらしい。メモ化前とメモ化後の形を同じにしたいというのはやはり考えていたらしい。ところが、関数をメモ化関数に変える関数というのは普通には記述できないので、テンプレート Haskell を使って解決しようとしていたとのことだ。テンプレート Haskell はバージョンによって比較的インターフェースの変更が大きい。それで用いる GHC のバージョンを指定したかったのだろう。

ところが、テンプレート Haskell のコードを書くのは一朝一夕ではうまく行かず、最終的に失踪してしまうこととなった。λ須先輩はそれを知っていたが、テンプレート Haskell を使いたくなかったのか、別の方法を模索することになる。部員にアイデアを出させるも、今ひとつ満足の行くものが出てこない。そこで俺に別の解法をひねり出させようと、細かいことを隠して依頼してきたということだ。

ちなみに、λ須先輩も別の解法を考えていたそうだ。 GTA 計算と言って、探索系の問題を、解候補の生成（Generate）、解の検査（Testing）、解の集約（Aggrigate）の三つの操作に分解して記述すると、自動的にメモ化、並列化が行われるというとんでもないフレームワークがあるらしく、これを用いて高速化を目論んでいたそうだが、独自の Generator を作るのがうまく行かなかったそうだ。

16.16 Epilogue

「すまない、悪気はなかったんだ」

三度、喫茶店にて。同じくλ須先輩と。だが、今回は心なしかしおらしい表情が伺える。

「君を試すような真似をしたのは謝ろう」

「あなたは別に俺に頼る必要なんてなかった。最悪の場合でも自分でどうとでもできたはずだ。そうですね」

「実際のところ、そうなる。だが.....」

少し言葉が胸につっかえるような表情を見せたが、やがて。

「今回の件で確信した。君は自分が思っているほど平凡な人間ではない」

「買い被りです」

「君がどう思おうと思わなかろうと、いずれそれを自覚する時が来るだろう。それまで、その評価値はサンクの殻の中に閉じ込めておくとしようか」

評価する必要のないサンクはそのままゴミ集め機に回収される、だから遅延評価と言うのだ。遅延評価を標榜する怠惰な一学生としては、せいぜいゴミ集め機の働きに期待したいところである。

—— 了

次回予告

21 世紀初頭。ますます成長を続けた IT 大企業は数々の国際組織をも傘下に収める存在となっていた。

世界のすべてを見える化し、ユビキタスに結合するクラウド、Web22.0 計画。完成すればサイバー時代の幕開けを告げるはずだった史上最大の計画は、史上最大の失業対策でもあり、地球人口の半数までもが人月で量って投入され、史上最大のデスマーチを産む運命であった。やがてプログラミングの自由を巡って世界大戦が勃発。—押し付けられた仕様を押し付けられた言語で書かされるのは、もう御免だ— エジャナイザ（打ち壊し）運動によりさまざまなプログラミング言語のエコシステムがほぼ完全に破壊され、人類は「納紀 A.D.00 年 (AD = After Deadline)」をもって新たな歴史をスタートする。

間もなくして、超言語連合（仮）の支配・統制により、「プログラミング」は「人の尊厳を奪うモノ」として規制される「程序禁止法」が成立。一部の地域（伝統芸能絶対防衛圏）を除いては、プログラミング活動は厳しく罰せられる事となった。プログラマが次々に姿を消していく・・・人類のココロは暗い闇に包まれていくのだった–

そして—

かつてプログラミングと呼ばれた営為は絶滅したかに思えた。だが、そんな時代に復活した言語があった。

「Haskell0098」

かつて地球の存亡をかけた戦いの中、傷ついた人々の心に光を灯すべく、最後までプログラミングの楽しみを伝え続けた「Haskell98」。そんな伝説のニンジャ達の光と魂を受け継ぎ、その名を襲名する形で、「非合法言語」として立ち上がったのだ！

非合法がゆえ、公式なライブコーディングは出来ない・・・。程序禁止の時代だからこそ、どんな危険も顧みず、あらゆる星へ強攻突入し、熱狂的なゲリラライブコーディングをファン達に送り届ける。そう、この時代の Haskell は、『遅延関数型言語』から『先攻関数型言語』として進化していたのだ！ 悪に支配された暗い世の中、彼女達の「奇跡のライブコーディング」は、夢を失いかけた人々に未来への希望の光を与え続けていく！

だが、圧倒的な影響力をもつ彼女達に魔の手が襲う・・・彼女達の活動を無視できなくなった超言語連合政府（仮）により、「クラッカー」と断定され、妨害を受ける。自分たちのハッケージを、そしてファンや市民を守る為に、自ら武器を手にとり少女達は戦う。「待っているファンがいる限り、傷つき倒れても戦い、そして、私達は書き続ける！」

そして、時は納紀 A.D.0098…。

運命の年に、新たなスターとなるべく選ばれた第７７期研究生を中心とした物語が、今、まさに始まろうとしている。

彼女達の中から、新たな「希望の星」として憧れのハッケージに立つことが出来るのは誰か？ 友情、ライバル、プロ根性・・・・今、プログラマは新たな輝きを放つ！！！！

スピーディーなプロトタイピング！！ キュートなデバッグシーン！ 全宇宙を巻き込むスペースライブコーディング！

『愛に生き、会いに行くアイドルプログラマー達』の感動ライブストーリーが、今、幕を開ける！！

「星をこえて、みんなに会いに行くよ♡」

第IV部

簡約! λ力娘 (4) (2012年冬)

第17章

インターフェース

— @xhl_kogitsune

※本章は、某 LLVM 狐本[*1](の表紙) に端を発した非公式妄想です。妄想しか含まれていませんのでご注意ください。また、実在の人物 (勿論きつねさん含む)・言語処理系等とは関係ありません。

17.1 Kihime Side

「わたし」と「あなた」は、「インターフェース」で隔てられている。
「わたし」と「あなた」のカンケイは、「インターフェース」で切られている。
「わたし」は「あなた」とでなくても、同じようにやっていける。「あなた」にとって「わたし」は交換可能なものに過ぎない。
それが、「インターフェース」の持つ意味だから。
「インターフェース」の両側にいるのが誰だって、変わらないんだ。
—— キヒメ

「わたし」と「あなた」は、「インターフェース」でつながっている。
この「インターフェース」は、みんな同じ、みんな持ってる、誰とでも話せるものだけど。
「インターフェース」を流れる言葉は、ありきたりのものだけど。
「わたし」は、「インターフェース」を通して、「あなた」に話しかける。
「インターフェース」を超えて、「わたし」の心が伝わるといいなぁ、と思いながら。
—— コヨネ

わたしがコヨネと出会ったのはいつだっただろうか。最初は、どこにでもいるフロントエンドとしか思わなかったのだろう。フロントエンドなんて交換可能な短い付き合いなんだから、気にも留めなくて、覚えてもいない。これは、そんなわたしと、コヨネとのおはなし。

——

むかしむかし、コンパイラはひとつだった。原初の混沌のごとく、渾然一体としていて、ひとつの高級言語から、ひとつの機械語を生み出す、一枚板のナニカ。セカイはとても複雑に絡みあってはいたけど、とても単純だった。だって、そこには自分しかいないんだから。
時が下り、コンパイラはふたつに分かれた。天と地。フロントエンドとバックエンド。その両者をへだてる「中空」たる、インターフェース、中間表現。

> —— n 個の入力言語と、m 個の出力機械語があるとします。一枚板のコンパイラでこの全てをサポートするには、$n \times m$ 個の実装が必要です。でも、中間表現を間に入れれば、n 個のフロントエンドと m 個のバックエンドの計 $n+m$ 個の実装があればよいのです。

それはその通りだ。でも、わたしたちはフロントエンドとバックエンドに別れ、わたしたちは中間表現で隔てられてしまった。わたしたちの声は、中間表現という枠にはめられて、枠に入らなかったものは相手には届かない。枠にはまった声は誰のものでも同じで、わたしと、おなじものだったはずの相手とは天と地のように遠い。わたしは、誰が作ったのとも知れない中間表現を淡々と処理するだけの日常を送っていた。

[*1] 柏木餅子, 風薬 (著), 矢上栄一 (表紙): **3 日で出来る LLVM**, MotiPizza (2012).
http://d.hatena.ne.jp/motipizza/20120724 http://d.hatena.ne.jp/sabottenda/20120728

そんな単調な日々の終わりは突然やってきた。わたしがいつものように中間表現を淡々と流していると、突然背中に冷たい感触が走った。

「……ひゃあん!?」

触られた! いきなり背中を触られた!

驚いて —— というより激しい怒りで —— 振り向いたわたしは、そこに一人のフロントエンドが立っているのを見た。わたしが処理していた中間表現を持ってきた奴だ —— 確か。

「ごめんなさい、そんなに驚くとは思わなくて……でも、内部インターフェースがあったから、なんなのかなー、って気になって」

あんまり悪いとも思っていなさそうな、のほほんとした態度の黒髪の女。それに余計にカチンと来たわたしは、

「だからって何で触るのよ! そもそも内部インターフェースなんだからアンタみたいなフロントエンドのためのものじゃないの! 他人の undocumented なトコロに触るなんてサイテー!」

そう叫ぶように言い捨てて、わたしは処理途中だった中間表現をそいつに突き返して、その日はそのまま帰ってしまった。この女が —— 名前は後から知ることになる —— コヨネだった。

後で聞いたところによると、この時のコヨネにとってのわたしの印象は「かわいい」、だったらしい。

『驚きと怒りのあまり顔を真っ赤にして涙目になりながら非難してくる、金髪のバックエンド。かわいい』

……ひどい女だ。

この日から、この女はわたしにちょっかいを出してくるようになる。

翌日もまた、彼女は中間表現を持ってやってきた。中間表現なんて、別にネットワーク送信で済むのだし、昨日の今日で手で渡しにくるという時点で警戒せざるを得ない。背中を向けないように注意しつつ、無言で受け取り、開く。

……何考えてるの、この女。

そこには、ぱっと見 nop しか書いていなかった。そもそも、中間表現において nop とか普通は使わない。なのに、nop しかないこれは何だ。嫌がらせか? いつもどおり淡々と処理しようとしたわたしはやや面食らい、一瞬手を止めてしまう。しかし、それは何か癪な気がしたので、そのまま nop の列をスルーし、さっさとほぼ空のコンパイル結果をつき返して後は無視することに決めた。

しかし、翌日以降も奴は妙な中間表現を —— それも毎日 —— 持ってきた。嫌がらせに違いない。気にしたら負け。わたしはそう自分に言い聞かせ、完璧に通り一遍のコード生成だけをして、さっさと突き返す。いつしか、そんなよく分からない日課のようなパターンが出来てしまっていた。

ある日、彼女はみかん箱を持ってきた。……今度は何だ。新手の嫌がらせか? しかも、みかん箱をわたしの机の上に乗せて、じっと見られても困るんだけど。

「……何これ」

仕方ないので、またろくでもない嫌がらせか早くカエレ! って気持ちを込めて聞いてみる。すると、中間表現のコンパイルを頼みたいのだと言う。いつものと違ってこれは普通のユーザープログラムを普通にコンパイルしたものだ、とも。

……みかん箱で中間表現持ってくる奴をわたしは初めて見た。手渡しで中間表現を持ってくるフロントエンドはこいつ以外にもたくさんいるが、それは言ってみればスクリプト言語におけるワンライナーのようなもので、量が少ない時にたまたま通りかかったから渡す、みたいなお手軽なものだ。量が多い時には普通ネットワークで送信する。そもそも、このみかん箱はどこから持ってきたのか。「愛媛みかん」って何だ。こいつみかん食うのか?

—— いけない、これは高度な攻撃だ。気にしてはいけない。しかし、こいつの言うことが本当なら、これはそれなりの規模のプログラムで、コンパイルには少し時間がかかりそうだった。

「……じゃあ、終わったら呼ぶから。えっと……」

—— そこで、わたしはこいつの名前を知らないことに初めて思い至った。いや、そもそもわたしはひとの —— 特にフロントエンドの —— 名前を覚えないから、前に聞いてすぐ忘れたのかもしれない。中間表現の向こう側にいる奴の名前なんて覚えて何になるのだろう? しかし、みかん箱一杯の中間表現、などというものを目の前に積まれて初めて、名前が必要な状況があることをわたしは知った。封筒一枚分の手渡しなら、その場ですぐコード生成して結果を目の前にいる奴に渡せばいい。ネットワーク越しに届いたものなら、名

前を見ずとも返信ボタンを押せばそれで終わりだ。だが、すぐにはコンパイルが終わらない量のコードを手渡ししてくる奴に結果を渡すには、後で名前で呼び出しをかける以外に何があるのだろう?

この場で待たせるのは邪魔なので論外。こいつがまた立ち寄るのを待つのも、来るか分からない待ち人を待っているようで嫌だ。……こいつの名前を訊くのも何か負けている気がするが、どちらにせよ、言いかけてしまった以上訊くしかない。

「コヨネ」

彼女はそう名乗った。わたしはその名前を手元のメモに書きとめると、彼女を追い払い、みかん箱の開封に取り掛かった。

中身は、確かに普通のプログラムっぽかった。わたしはいつもどおり淡々と、読み込み、フロー解析をし、最適化を適用する。命令選択をし、スケジューリングをし、レジスタを割り付け、コードを出力する。それだけ。

でも、わたしは何かいつもと違うものを感じた。何だろう。

—— コードの特有のクセ。

—— ある種型にはまった、最適化の利きそうなパターン。

—— 元のソースコードが見えそうな錯覚さえ覚える、最適化が導かれるような、不思議な感覚。

これは、このプログラムを書いたプログラマのクセだろうか? それとも、プログラミング言語自体によるクセ? これを、わたしはどこかで見たことがある気がする。あの女、何の言語のフロントエンドだったかしら? ……そもそもわたしはあの女に何の興味もなかったので知るはずもなかったのだが。

最適化が進む。実行してプロファイルを取ってみないと最終的には何とも言えないが、最適化が進んでいく感覚がする。わたしの直感（ヒューリスティクス）とひらめきを導く、目に見えない何か。

そして、半ばあたりで、わたしは唐突に気づく。これは、プログラマでもなく、ソースの言語でもない、あのフロントエンドの女のクセなのだと。どこかで見たことがある、それは、あの毎日渡しに来ているあの嫌がらせコードだ。トリッキーなコードは、真面目にコンパイルすると無駄に手間がかかったりする。わたしは、あの女に少しでも早く結果を突き返して追い払うべく、楽な最適化方法をコードから勘で割り出すことで自分のコンパイル速度を最適化していた。このクセやパターンは、そこで目にしていたものだったのだ。

そう気づいたわたしは、このコードから目を離せなくなった。同時に、急に顔が赤くなるのを感じる。インターフェース。中間表現。コードそれ自体以外何も通さないと思っていた壁越しに届いた、ある種のメッセージ。「このコードは私がコンパイルしました」という、フロントエンドからの。その相手が、あの嫌がらせ女であることは残念だが —— わたしは、しばらく手を止めたまま、とりとめのない思考を走らせていた。

コンパイルが終わって —— 自分で言うのも何だが、なかなか良い出来に仕上がったと思う —— 、机に貼ったメモを見てあの女 —— コヨネを呼び出す。あの中間表現は何だったのか。結果を渡す時、そう聞いてみたい気がしたが、自分でも一体何を聞きたいのかが分からなくて、結局何も聞かないままで終わった。

—— さて、奴はみかん箱を抱えて帰った、これで仕事は完了、あの女の名前も忘れてよし。……のはずなのだが、今日の中間表現のことが気になっていたわたしは、なんとなく机のメモをはがして捨てず、そのままにして自室に引き上げたのだった。

それからも毎日の嫌がらせ中間表現は続いたが、彼女はちょくちょく普通のコードも持ってくるようになった。……毎回手渡しで。たまにみかん箱で。あいつが目の前にいると早く追い払いたい気持ちが先に立つが、後で渡すからと追い払ってから中間表現を読んでいると、色々面白い発見がある。あの女のクセ。わたしに最適化を丸投げしているような箇所。あるいは、クセを超えた、何か最適化ヒントめいたもの。中には、わたしのデフォルト最適化ルーチンが最適化しやすいように変な書き方をされたコードまであった。この前は一度、わたしが命令スケジューリングとレジスタ割付のどちらを先にしているかを試すような中間表現を渡されたこともあったし、ひょっとすると、あの嫌がらせの中にはそういうのが結構混じっていたのかもしれない。

中間表現を通じて伝達されるメッセージ。

—— ここ、最適化してほしいのだけれど。

—— こう書けば、あなたは最適化してくれるかしら?

—— あなたがどの順番でコード生成しているのか、わたしはそれを知りたい。

そして、わたしはそれに対して誠実に答えを返す。もはや、相手があの嫌がらせ女であることも気にならなくなっていた。

でも、とわたしは考える。これは、していいことなのだろうか、と。

フロントエンドとバックエンド、わたしたちは最初から分け隔てられて創造された。

効率的なセカイのために。

誰とでも仲良くなれるように —— 裏を返せば、誰とも必要以上に仲良くならないように —— 。

間をつなぐのは中間表現だけ。だから、わたしもフロントエンドからは中間表現しか受け取らなかった。つまらない、とは思った。わたしたちに定められた“正しい”インターフェースが、こんなにも無味乾燥で、気持ちも何も伝わらないことに。でも、それでいいんだと —— わたしは、諦めていたんだと思う。

それでも、そんな中間表現ごしに、気持ちは伝わってしまった。正直に言おう。うれしかった。わたしたちは、完全に断絶されて創られたのではないのだと。このセカイは、インターフェースは、ちゃんと気持ちを伝えられるのだと。—— ほんの少しだけ、あの女に感謝した。まぁ意図してのことじゃないのかもしれないが。

でも……中間表現ごしに仲良くなるのは、いいことなのだろうか? 形式的には問題ない。フロントエンドからバックエンドに渡されているものは、普通の形式の中間表現なのだから。それでも、コレは、セカイの意図からは逸脱しているのではないだろうか? あるいは —— それがいいのなら、中間表現以外で仲良くなっても、いいのではないだろうか?

—— 仲良くなりたい。わたしは、そう思っているのかもしれない。気の迷いだ考え直せ、とは思いつつ。

今日もまた、コヨネは中間表現を渡しに来る。今日は、嫌がらせじゃない方らしい。きっと、この中にはまたいろいろなものが詰まっているのだろう。

—— これはいいことなのか。

—— これ以上仲良くなってもいいのか。

—— こいつはどう思っているのか。

そろそろ我慢できなくなっていたわたしは、わたしのルールを破って、聞いてみることにした。

「ねえ、ちょっと聞きたいことがあるんだけど、いい?」

その場で話を続けるのは論外だった。わたしがフロントエンドに声をかけた瞬間、近くでコンパイルしていたバックエンド友達がすごい勢いでこっちに振り向いた。ぽかーんと口を開けたまま、ガン見してくる。……そんなに珍しいか。あいつにこれからの話を聞かせるわけにはいかない。わたしは席を立つと、ついてくるようコヨネに身振りで示して、どこか落ち着いて話せる場所を探すことにした。

わたしが選んだのは、建物の外周に設けられた、広い屋外休憩スペースだった。寒いし、普段ほとんど使うことはないのだが、だだっ広く、一部微妙に入り組んでいて、人もほとんどいないので、あまり見られたり聴かれたりはしないだろう。

「……話ってなにかしら?」

「まずは、その……ありがとう。何言ってるか分からないかもしれないけど、中間表現ごしにいろいろメッセージくれて」

素直になろう。今日だけは例外。

「でも、あんたはどう思う? これってやっていいことなのかな? バックエンドはフロントエンドからは隔たれて創られた。なのに、特定のフロントエンドと仲良くなるのって、いけないことじゃないのかな?」

コヨネが戸惑う気配。言葉足らずだっただろうか。その自覚はある。でも、わたしも自分で何を言いたいのか、よくは分かっていないのだ。

「……どうして、そんなこと言うのかしら。それで最適化が進むのなら、いいんじゃないかしら?」

「でも、わたしたちは分けて創られた。そのデザインは、インターフェースは、守るべきじゃない?」

仲良くなりたい。ああ、もう認めてしまおう。わたしはこいつと仲良くなりたい。

しかし、それはいいことなのだろうか? 何度も繰り返した問いが、わたしをうつむかせる。フロントエンドとバックエンドは独立であるべき。今までずっと当然だと思っていたソレが、今のわたしを縛っている。どうすべきなのだろうか? 分からない。わたしはうつむいたまま動けない。

ふと、指が頬に触れた。あの時わたしの背中に触れた指。コヨネの指。それが私の顔を上げさせる。コヨネの顔。わたしは、嫌がらせ女とか言いつつ、コヨネのことをとっくに好きになっていたんだと思う。そして。

「どのように創られたかと、どのように生きるかは別の話よ。神様は自分の手間を省くために私たちを分けて創造したけれど、私たちが勝手に仲良くする分には構わないんじゃないかしら。そもそも、私たちがどう生きるかは自由だわ。私は、隔てて創られたあなたと、仲良く共に生きたい」

—— これが決定的だった。とどめの一撃。コヨネはずるい。こんなのずるい。でも、わかった。これは、コヨネの偽りも飾りもない本心なのだと。

で、今に至る。

コヨネは中間表現ごしの恋文を好んだ。

最初にわたしが気づいたのは、nop まみれの単なる嫌がらせだったが —— あれが恋文だったとしたら送り主はよほど歪んでいると言わざるをえない ——、実は、「ちょっかい」はこの nop に限らず、色々やっていたらしい。ただわたしが気づかなかっただけで。最初は nop の列とか、分岐パターンを全列挙して最適化すれば全命令が消える、なんていうパズルとか —— これも嫌がらせだろう ——、そういう特に意味のないものが多かった、らしい。その頃のわたしはそういうのは全部スルーしていたのだが。

そして、嫌がらせ以外のメッセージも、コヨネはたくさんくれた。ループを命令列に落とす時のちょっとしたクセ。命令ニーモニックのわずかな自由度にエンコードされた分岐頻度情報。プリフェッチの微妙なポジショニング。中間表現が語る、コードの意味以上の何か。

「なんかふつーに話しかけても無視されたし、他のバックエンドから『キヒメは内部インターフェース触られるの極度に嫌うし、中間表現しか受け付けないし、でも中間表現のこと誰が書いても同じ無機質なインターフェースって馬鹿にしてるし』って聞いてたから、こういう趣向はどうかな、って」……なんてまどろっこしくて分かりにくい恋文。しかも、それはかなりの期間、たくさんのバリエーションで続けられたらしい……というか、今も続いている。そんなにもわたしはコヨネに想われていた (そして今も)、ってことなのかしら。……いやその理屈はおかしい。そもそもそんなこと考えてしまう時点でコヨネの策略にはまっている気がする。

「最初は、キヒメをこれで惚れさせようって思ったんだけど、全然気づいてくれなくて。だから、ちょっと背中のアレを、ね?」

やっぱり、ひどい女だ。それでもやはり、わたしはコヨネの中間表現が、好きなのだ。

仲良くなってからは、わたしもコヨネの謎中間表現 —— 単なる嫌がらせだと思っていたもの —— に付き合うようになり、他のパターンのメッセージにも気づくようになった。

分岐命令の方向とか、いくつかのどちらにでもできるビットを使ってエンコードされたアスキー文字列。

逆から読むと別のプログラムになって、わたし向けのメッセージを再生するコード。……順方向だと意味不明なコードを超がんばって読んでコード生成したわたしの時間を返せコノヤロウ。

もっと直接的に、中間表現のバイナリをテキストとして見るとわたしに向けた祝いの言葉が書いてあるコードなんてのもあった。……あれもコードとしては意味不明だったので丁重に

「`Compile Error:` **わたしの機嫌が悪いのでこんなゴミコード送りつけんな馬鹿。ありがと**」

と送り返しておいた。まぁ、あれはものすごく素敵なバースデーカードだったけど。

今でも、コヨネは何か嫌なことがあるとわたしに nop 列を送りつけてくる。この前は 4.5GB にわたる nop 列が届いた。完全に八つ当たりである。……まぁ、nop 列を処理しながら、というか片っ端から nop を捨てながら、そんなコヨネもかわいい、とかわたしが思っているのは秘密であるが。そんな時でもコヨネはメッセージを仕込んでいることが多い。4.5GB の nop 列は、その nop の数で整数を……整数にエンコードされた文字列を、エンコードしていたのだ。

「`Come`」

はいはい、わかりましたよ、と……。まぁ、おそらくそれに気づかなくても同じことをしただろうが、わたしは、全部 nop であることを確認してゴミ箱に捨てた後、コヨネを慰めに行ってあげたのだった。

でも。この中間表現のチャネルは、コヨネからわたしへの一方通行のチャンネルなのだ。フロントエンドはずるい。バックエンドからでは、Error や Warning を出すくらいしかない。でも、コヨネのコードに対してありもしない Warning とかを出すのは、バックエンドとしてしたくなかった。コヨネのコードは、いつも —— いや、コヨネが変な嫌がらせをしてくる時以外はいつも —— きれいだったから。ああいうコードは、ノーエラー、ノーワーニングで通すべきなのだ。

コヨネに完全に惚れてしまったわたしは、バックエンドとしては堕落してしまったのかもしれない。中間表現ごしの睦言だけでは足りない。絶対に足りない。そもそもわたしからはほとんど何も言ってない。他のインターフェースを。コヨネとわたしだけの、特別な、直通両方向インターフェースを。フロントエンドとバックエンドの独立性? それはある意味実装の手間のための教理(ドグマ)だ。実装の手間を惜しまず、あるいはインターフェースのきれいさを犠牲にして、果たしたい目的 —— 多くは性能 —— を追求してきた人はたくさんいた。今更、そこにわたしがひとり加わっても大丈夫だろう。コンパイラとしてのあるべき姿を守るために、warning で会話するといったことを —— コヨネが明らかな嫌がらせしてきた時を除いては —— 我慢したのだから、少しくらいバックエンドとしての逸脱をしてもいいのではないか。

まずは内部インターフェース。最初にコヨネに触られたあの背中のやつ。腕の内側。耳。しっぽ。内部インターフェースはその名の通り、「内部」、つまりわたし自身に閉じたインターフェースだ。中には他のバックエンドやコントローラに向けてのものもある。でも、決してフロントエンド向けのものではない。わたしは、それまで内部インターフェースをフロントエンドに触らせたことはなかった。半分は独立性維持のために、もう半分は、たぶん単純な生理的嫌悪感によって。

最初の時もそうだったけど、コヨネはたびたび内部インターフェースを触ってきた。

「わたしはもうあなたにとって単なるフロントエンドの一人じゃないんだから、内部インターフェースさわってもいいのよね?」

そんなことを言われたことがある。駄目に決まっている。だって、それでもやっぱりあなたとわたしはフロントエンドとバックエンドなのだから。コンパイルにおいてこのふたつのインターフェースはやはり中間表現でなくてはならない。そんなこと思いながら無視していたら、続けて、

「じゃぁ、コンパイル以外のことをしましょう。それなら文句ありませんわね?」

そんなことを言われた。口調がおかしい。それにコンパイル以外って、何をす、ひっ!?

「何すんのよ!」

「コンパイル以外のこと」

また触られた! 何なのこいつ!

コヨネはどうも、わたしを無意味にいじめたがっている節がある。そんな時はわたしも全力で抵抗するのだが。

最初の頃はそんなんだったけど、最近はもうわたしも内部インターフェースへのアクセスを許しつつある。そんなインターフェースを、コヨネはひとつひとつ叩いていく。わたしはそれに対して反応(レスポンス)を返していく。フロントエンド向けのインターフェースではないから、支離滅裂な意味をなさない反応になる。支離滅裂だけど、コヨネはそれを楽しんでいる節がある。わたしも、わたしのインターフェースが支離滅裂なレスポンスを返すのを楽しんでいるのかもしれない。……やはり、わたしは堕落したのだろうか?

あるいは禁断の friend 宣言。しかも、これは、わたしから。private、つまり原則他人には一切触れられないインターフェース。それに対して、名指しで、特定の人のアクセスを許可する……もちろん、コヨネに。それをやったら、コヨネに見つけてもらう前に速攻でバックエンド友達に見つかった。

「あー、キヒメちゃん、何その private 関数? friend 指定? じゃあさわらせてよー友達じゃーん?」

……こんな性格のわたしにも、バックエンド友達の一人や二人はいる。……お願い、信じて。コヨネにはこのネタで散々いじめられた。ひどい女だ。ともあれ、触らせるわけにはいかない。だって、これはコヨネ用のものだもの。

「`Compile Error:` **その関数は** `private` **です**」

冷たい目でそう突きつけても、わたしの友達なだけあって、軽く言葉を返してくる。

「なんかさー、最近キヒメちゃん丸くなったよねー、なんてゆーか、睨んできても凄みに欠けるってゆーか、ただ単に照れてるだけに見えるってゆーか。前はもっとゴミクズを見るような目で見下してきてくれたのにー」

鋭い。わたし自身も自覚しているが、明らかにそれはコヨネのせいだ。……しかし。

「ゴミクズを見るような目、っていうのはさすがに心外なんだけど……」

そんなにひどい目をしていたのかしら、わたしは。

「褒め言葉だよー?」

この子の言っていることはたまによく分からない。ともかく、わたしは彼女の魔の手をかいくぐり、コヨネを探しに行ったのだった。

—— private のアクセス制限は、もはや完全に生理的なものだ。さわられると、鳥肌が立ち、自分ではどうしようもない嫌悪感が背中を走る。それがコヨネの手によるものであっても。コヨネもそれを分かってくれるのか、すごく優しくしてくれる……言葉以外は。

「こうやってキヒメいじめるのって、楽しいわ……。キヒメちゃんがおとなしく我慢しているのを見るのは特に」

ひどい女だ。

コヨネはあったかい。コヨネの声、コヨネの手、コヨネのしっぽ、そして —— コヨネの書く中間表現。わたしとコヨネの心は中間表現越しにつながっている。以前感じていた、諦念のようなものは、今のわたしにはない。今日もわたしは、次のデートはどこに行こうかとか考えつつ、コヨネからの中間表現を読み、わたしからコヨネへの気持ちの伝え方を考えている。それが、最近のわたしの楽しい日常なのだ。

——

むかしむかし、コンパイラは独りだった。ひとつの高級言語から、ひとつの機械語を生み出す、その全てを自分でやっていた。

コンパイラはふたりに分かれた。わたしとあなた。ひとりではできないことをするために。インターフェースを通じて、みんなでおしゃべりしながら。

17.2 Koyone Side

最近、私には気になる子がいる。いつもつまらなそうな顔をしてコード生成してる、キヒメという名の、金髪碧眼のバックエンド。……なんで、気になるんだろうか。

—— 恋に理由なんて要らない。私は、そういうことにして、「いじめたら可愛いだろうな」という第一印象を塗りつぶすことにした。

一般的なフロントエンドの女の子としては、しばらく想いをあたためたり、「好きな人が、できたの///」とかいう話を友人にこっそり打ち明けてみたり (そういう時に「インラインコメント出てるよ」という表現が最近流行りだが、当然言語依存なのでフロントエンド間での格好の論争の種である)、そののちおもむろに、とかなのかもしれないが、命短しなんとやら。気になったのなら行動をためらうのは私の性ではない。

—— ということで、とりあえず、ラブレター、を、書いて、みた。

ラブレター。10 年前には既に廃れたメディア、直に口で告白できないような奴の使う手段。そう思っていたのに、なんで、私の机の上には自筆のラブレターが載っているのか。……ひょっとして、私は自分で思う以上に舞い上がっているのだろうか? ……それとも、自分で思っている以上にチキンなのかしら?

そもそも、これはラブレターと言えるものなのだろうか。言語は中間表現。バックエンドの娘への恋文としては妥当な選択だと思う。`ILoveYou` 関数とか定義してあったり、プログラム構造そのものに書きたいことをエンコードしてみたり、文字列定数で名前や連絡先を書いてみたり。……いかなる精神状態で私はこれをジェネレートしたのか。

これはあからさますぎはしないでしょうか?

いえでも、らぶれたぁとはそういうものでは?

自問自答が頭をめぐる。目の前には、普段は与えられたプログラムから中間表現を作るだけだった私がゼロから書いた、たぶん初めての中間表現。

—— ためらうのは私の性ではない。そういうことにして、私は、目をつむってそれを送信した。

ほどなく帰ってきたレスポンスは、

「`Compile Error: variable ‘your_response_here’ is not defined.`」

バックエンドの娘の返答 —— 諾か否か —— を入れるところを故意に undefined variable にしていたので、当然コンパイルはエラーなのだが……返事はそれだけ。何か他の経路で返事が来るかも、と期待したが、そ

れもなし。……これは、無視されたか、あるいはラブレターであることに気づかずに単にコンパイルして終了したか。がっかりしたが、どちらにせよ、更なる行動が必要だった。

とりあえず、廊下ですれ違う時に……当然すれ違うように仕組んだのだが……直接話しかけてみた。
「こんにちは、キヒメさん」
スルーされた。
「……えっ?」
そのまま、何もなかったのように歩き去っていく彼女。ぽかーんとしていると、後ろを一緒に歩いていたらしいバックエンドに話しかけられた。
「ごめんねー、キヒメ、フロントエンドには冷たいのよー」
「なん、ですって……」
そこでひっかかるとは思っていなかった。
「その、それはフロントエンドだけ、なの?」
「んー、バックエンド友達にもまぁ似たような感じで無愛想だけど、一応話はできるよー」
バックエンドうらやましい。
「ふふん、たまにパフェとか食べに行ったりもするんよー。物食べてるコヨネかわいくてさ、まぁ無愛想はそのままなんだけど、おいしい時とそうでもない時とでテンションがすごく違ってさー」
「……うらやましいですね。そのポジション、代わってくれないかしら」
「だめー」
本当にうらやましいわ。
「……どうしたらいいのかしら」
「うーん、気がある、ってやつなのかな?」
「えっ……それは……」
「でも、キヒメ、掛け値なくどのフロントエンドにもあんな感じだし、あたしも、どうしたらあの子が心開くのかは知りたいなー。ごめんねー何の参考にもならなくて。まぁがんばりたまえ、わはははは」
そう、なんだかよく分からないことを言って、そのバックエンドもキヒメを追っていってしまった。

今度は、友人のフロントエンドに相談してみる。
「えっ、あんた中間表現でラブレター書いたの!?」
「ええ」
「ないわー」

100 ms（ミリ秒）で reject された。
「え、でもあなたもフロントエンドなのだから……」
「ないわー」
そんな……。自然なことだと思ったのだけれど。
「でも、キヒメに目をつけるとは、あんたもなかなかやるわね」
「……それはどういう意味かしら?」
「彼女、変な意味で目立ってるのよ。フロントエンドの仲間内で一時期観測スレも立ったんだけど、彼女、フロントエンドに対しては取り付く島なさすぎて観測するネタがなくて、すぐ廃れちゃったのよねー……。んー、でもそうかー、直接話しかけても口きいてくれないのかー。彼女らしいわー。あれ、でも、誰かが中間表現手渡ししてたって話は聞いた気が……」
「なんですって?」
「いや、ラブレターとかじゃないと思うわよ? 彼女だってバックエンドなのだから、中間表現は受け取るだろうし、あんただって他のバックエンドに手渡しすることあるでしょう?」
確かにそうだ。盲点だった。あるいは、面と向かってラブレターを渡す、なんて恥ずかしいことを無意識に拒否していたのだろうか。
「……分かった。試してみる」
「だから、ラブレターはないと思うわよー?」

翌日。私は、この前書いたラブレターを少し書き直して、物理的に持って行くことにした。

—— 目標を肉眼で確認。これより作戦行動を開始する。

「あ、あの、キヒメさん……っっ」

……まずい。声が上擦った。これではまるで、憧れの先輩にラブレターを渡す女子中学生ではないか。いえ、今から渡そうとしているのはラブレターに他ならないのである意味正しいのかもしれないが、自分がこんなに乙女だったとは思わなかった。彼女は無言。というか無反応。私も動転して声が出なくなったので、とりあえず中間表現 —— という形のラブレター —— を無言で彼女に押し付けた。

「……」

彼女は中間表現の入った封筒を一瞥すると、無言で受け取り、開き、コンパイルする。封筒を開く彼女の指。中間表現を眺める（読み取る）まなざし。私のきつねの耳がかろうじて捉える息遣い。

「……はい」

彼女の声。初めて聞くその声は、暗く平坦だったけど、澄んでいてすごくきれいだった。

「……?」

彼女の眼。そこで私は、コンパイル結果を差し出されていることと、私がフリーズしているのを不審がられてることに気づいて、あわてて受け取った。

「……あ、ありがとう……」

彼女の眼。眼があったように思う。でも、結果の封筒を受け取った瞬間、その眼はそらされて、彼女はもう私を見てはいなかった。彼女の眼に私が映っていたのかどうか、気になったが、それを確かめる術はなかった。

「じゃあ、またね……」

一応そう声をかけて、私は自室へと引き返すことにした。

「`Compile Error: variable 'your_response_here' is not defined.`」

自室で開けた封筒の中身は、予想通り何の変哲もない、同じく一行だけのものだった。エラーに対してちょっとくらい表情動かすかしら、とも思ったのだけれど、そんなこともなかった。

取り付く島のない、バックエンドの女の子。

話しかけても無視される。

でも中間表現は受け取る。

受け取るけど、単にコンパイル処理結果が返ってくるだけで内容は彼女の心を素通りするようだ。

私は、長期戦を覚悟し、作戦を考えなおすことにした。

「好き」の反対は、「嫌い」ではなく、「無関心」だ。つまり、「無関心」は「嫌い」よりも「好き」から遠くて、無視されている今の私はたぶんそれよりも遠いところにいる。ということで、まずは好感度を下げてみることにした。それで、話しかけて何か反応が返ってきたり、あるいは中間表現の内容に気をとめてくれればいいなぁ、と思って。

とはいえ、本気で嫌悪されるのは当然避けたいな、と思う。私にはそういう趣味はないのだ。しかし、あれから何度か話しかけてみたが、どれも完全にスルーされた。あれは、無視されているというよりかはそもそも耳に入っていない感じがする。中間表現に対するレスポンスも同じく機械的なもの。

……これは、物理的にちょっかい出すしか、ないわね。

なぜかぞくぞくするものを感じながら、私はそういう結論に達した。今は夏。他人には興味なんてないわ、って顔しながらなぜか背中の大きく開いた服でおしゃれしているので、そこをつつくのはどうだろうか。

……それくらいなら、いい、わよね?

いいことにした。行動はためらわない。早速、翌日試してみた。

彼女は基本、無反応である。声をかけなければ当然無反応。声をかけても、中間表現を渡すのでなければ無反応。つまり、逆手に取れば、後ろからそっと近づいて、その無防備な背中にそっと指を伸ばしても無反応、ということなのだ。とりあえず、一応のおとりとして中間表現を渡した後 —— 今回はラブレターではなく、誰かさんの書いたプログラムから生成した普通の中間表現だ ——、中間表現に眼を走らせている彼女の

後ろに回りこみ、背中に内部インターフェースを発見した私は、それに指を伸ばした。私の指が彼女の背中のインターフェースをつーっとなでる。
「……っっっ!??」

反応は覿面(てきめん)だった。声にならない悲鳴。反射的に肺の空気が押し出されただけのような音。振り向く彼女。睨んでくる眼。
今度こそ、しっかりと眼が合った。彼女の眼には、私がちゃんと映っている。うれしい。
心の中でガッツポーズをしながら、もう一押し煽ってみる。
「ごめんなさい、そんなに驚くとは思わなくて……でも、内部インターフェースがあったから、なんなのかなー、って気になって」
今度は普通に言えた。
「だからって何で触るのよ! そもそも内部インターフェースなんだからアンタみたいなフロントエンドのためのものじゃないの! 他人の undocumented なトコロに触るなんてサイテー!」
涙目で叫ぶ彼女 —— キヒメ。やっぱりかわいい。機械生成の中間表現大量に送りつけて DoS 攻撃したらこの娘はどんな顔するのか見たい。……私は、こんなにいじめっこだったのだろうか?
そんなことを考えているうちに、キヒメはコンパイル途中だった中間表現を私につき返して、そのままどこかに行ってしまった。

これで、少しは私のこと見てくれるとうれしいのだけれど。さすがに踏み込みすぎたのでこれ以上繰り返すと本当に嫌われることを感じ取った私は、翌日からは中間表現でのラブレター作戦に転向することにした。
まずは手始めに、中間表現でのメッセージに気づいてもらうところから。nop だけで埋めれば、さすがに気づいてくれるだろうか。nop だけの中間表現を封筒に入れて、キヒメを探しに行く。
キヒメは昨日と同じ場所にいた。
「こんにちは、キヒメ」
また無反応かな……そう思いつつも声をかけると、睨まれた。警戒対象になっているらしい。……いい傾向だ。少なくとも、私というものを認識して識別してくれるようになったのだから。とはいえ、ここから仲良くならないと意味がないのだが。
とりあえず、中間表現の入った封筒を渡す。キヒメは、睨みつつも無言でそれを受け取って、読んで、コンパイル結果を返してきた。
……読んでいる間に、いぶかしそうな表情に一瞬なった、というのは私の気のせいだろうか? 結果を受け取ると、キヒメはぷいっと眼をそらして無視モードに戻ったようだった。今日は私が認識されていることが分かっただけでも収穫だったわ、と思いながら、別れの挨拶を一方的にして、撤収することにした。
それから、毎日一回、何らかのネタを中間表現に仕込んで、キヒメに手渡ししにいくのが日課になった。睨まれるし無言だけど、毎日変な中間表現を渡しているうちに、中間表現をちゃんと読んでくれるようになった……気がする。でも、"Hello"とか文字列定数で埋め込んでもスルーされるので、嫌われてる状況は変わっていないようで、私は手詰まりを感じ始めていた。

いつもネタ中間表現だけというのもどうなのか、と思ったので、ある日、結構大規模なユーザープログラムをコンパイルしてキヒメの所に持っていった。ちゃんとした中間表現を渡すのは実はこれが初めてだったので、実はすごくどきどきした。キヒメはここちゃんと最適化してくれるかしら、言語特有の構造だけどフロントエンドだけでは最適化しきれない部分を汲んでくれるかしら、とか考えながらコンパイルしている時間も楽しくて、自分でラブレターを書くのとは違う感情の高まりを感じた。……その高まりに乗じて変なトリックを仕掛けることは、かろうじて自重した。
コンパイルが終わって持っていく段になると、今回は規模的に封筒ではなく、ダンボール一箱になった。まぁ、物理的な量は実はどうとでもなるので気分の問題に過ぎないのだが。
「……何これ」
箱をキヒメの前に置くと、嫌そうな顔でそう聞かれた。
「な、何って、コンパイルお願いしますわ。断っておきますけど、今回は普通のユーザープログラムですからね」

「……終わったら呼ぶ」
　そう言って、箱を開けて中身を読み始めるキヒメ。もう私は眼中にないようだったので退散する。
　……なるほど、ちゃんとした中間表現を持っていけば、少しは会話のチャンスがあったのか。気づかなかった。今までネタ中間表現ばかり持っていって、「どうせまた変なやつなんでしょ、時間の無駄だから早く済ませたいわ」みたいな顔されて、無言でそっけない結果返されるだけだったので、言葉を交わしたことなどほとんどなかったのだ。
　しばらくすると、キヒメからメッセージが届いた。初めてのメッセージ。内容はなんてことはない、
「コンパイルが終わったから取りに来て」
ってだけだったけど。
　コンパイル結果を取りに行くと、いつも睨んでくるキヒメが、珍しく睨んでこなかった。コンパイル結果の入った箱を指差しつつ、なんか変な顔をしている。無表情のような、それでいて少し困っているような…。しばらく見つめていたら眼をそらされたので、おとなしく箱を持って帰ろうとしたら、
「……ねぇ」
「……え?」
　話しかけられた。でも、コヨネは、少し迷った後、結局
「……なんでもない。もってって」
　と言っただけだった。

　コヨネの生成したコードは、なんというか、そっけない感じだった。ただコンパイルして最適化しただけ。でもよく見ると、私がコンパイルする時に考えていた「ここは最適化してほしいなぁ」「ここはこういう意図なんだけど通じるかしら?」という箇所が、結構私の意図通りに最適化されていて、キヒメが私の生成した中間表現をちゃんと読んで理解してくれたことに、私はうれしくなった。
　ユーザープログラムをコンパイルして渡すのはやはりおいしい。味を占めた私は、それから機会があるごとにユーザープログラムをコンパイルしてキヒメに渡すようになった。どうやら、キヒメは私が気合を入れてコンパイルして最適化した中間表現をご所望のようで、そういうのを渡すとどうやらキヒメも気合を入れて最適化するらしく、しばらく待たされて結果を受け取ると、してほしかった最適化が結構されていたりする。逆に、時間がなくて適当にコンパイルしたものを渡すと態度が —— 少しだけではあるが —— 元々冷たいけど —— もっと冷たくなる。
　最初は単に「私とキヒメはフロントエンドとバックエンドだし、キヒメは中間表現しか受け取ってくれないから中間表現で色々渡そう」くらいのことだったし、仲良くなるためにがんばろう! って感じだったのが、いつしか、キヒメと中間表現越しに会話するのが純粋に楽しくなっていた。
　私は中間表現で意図を伝えようとする。それが伝わったかどうか、キヒメの返答はコンパイル結果でなされる。
　私は、キヒメに伝わるように、中間表現の書き方を工夫する。検出しやすいような特定のコード・パターン。キヒメが諦めてしまいそうな例外的条件のくくり出し。分岐頻度情報に埋め込んだ大域的な実行パターンに関する情報。気のせいでなければいいのだけれど、キヒメもそういう私の工夫に次第に気づくようになった、気がする。最初に伝えたかった……今も伝えたいのだが……好意の伝達にはなっていないが、これはこれで心が通じている感じがするのだった。

　最近、キヒメとも言葉を交わすようになったし —— まぁ「結果は後で取りに来て」とかの中間表現に関する打ち合わせだけだけど、でも結果は別に直接渡す必要はないのだから取りに来てって言われるだけでもすごいことではないかしら ——、態度も少しやわらかくなってきた気がする —— 友人に相談したら「妄想では?」と言われたが —— ので、そろそろデートとかに誘っても、いいかしら? なんて思っていた矢先、キヒメから声をかけられた。
「ちょっと話があるんだけど」
　中間表現を渡しに行ったら、そう声をかけられて、そのままテラスまで連れて行かれた。風が気持ちいい。夕暮れ時、赤い光に染まるそこは —— まぁその、いわゆるデートスポット、というやつだった。
　初めてコンパイルのこと以外で声をかけられた。こんな所に連れてこられた。先を越された。これから何があると想定されるのだろうか? あまりのことに、私は混乱の極みだった。

「まずは、その……ありがとう。何言ってるか分からないかもしれないけど、中間表現ごしにいろいろメッセージくれて」

キヒメの声。不意討ち。不意討ちすぎた。何か色々伝わっていることはなんとなく分かっていたが、こんな、いきなりお礼を言われるだなんて。そもそも、お礼を言われるようなことをしたかしら。思考がぐるぐる回ってフリーズしている私の前で、キヒメは言葉をつなげる。

「でも、あんたはどう思う? これってやっていいことなのかな? バックエンドはフロントエンドからは隔たれて創られた。なのに、特定のフロントエンドと仲良くなるのって、いけないことじゃないのかな?」

えっ、ちょっと待って、キヒメも「仲良くなった」って思ってくれていたの。何これうれしいのだけれど、私はもうどうにかなってしまいそうだった。いえ、既にどうにかなっている気がする。でも……突然のことにオーバーヒートした頭で考える……この子は何を言っているのかしら。仲良くなるのがいけないこと? そんなこと、考えたこともなかった。

「……どうして、そんなこと言うのかしら。それで最適化が進むのなら、いいんじゃないかしら?」

かろうじて言葉を紡ぐ。中間表現と一緒に気持ちを伝え、いいコードが出せる。それは、とてもすばらしいことではないかしら。

「でも、わたしたちは分けて創られた。そのデザインは、インターフェースは、守るべきじゃない?」

そう言って、キヒメはうつむいてしまった。

……この子は、何に義理立てしているのかしら。自分たちの創造意図。カミサマ。そんなのよりも、私を見てほしい。

キヒメはうつむいたまま動かない。私も声が出ない。彼女も、私と仲良くなりたいと思ってくれていたのだろうか。私も仲良くなりたい。仲良くなることが悪いことだなんて、絶対にない。そう伝えたい。

コヨネ、勇気を出すのよ。—— 認めよう。私は自分で思っていたよりもかなりチキンだった。それでも、私は、キヒメの頬に指を触れ、うつむいてしまった顔を上げさせて、告白の言葉を言った。

「どのように創られたかと、どのよう生きるかは別の話よ。—— 私は、隔てて創られたあなたと、仲良く共に生きたい」

その後も私たちは、中間表現やバイナリを送ったりして過ごしている。キヒメがネタ中間表現にも付き合ってくれるようになったので、色々と仕込みがいがあって楽しい。

あと、キヒメがやたらとべたべたしてくるようになった。直接もふもふできるのはすごく嬉しいのだけれど、中間表現越しにまだるっこしいやりとりでいちゃいちゃしたり、じらせたりする楽しみも知ってしまったので、これはもうやめられない、と思う。それに、キヒメは「中間表現だとコヨネからわたしへの一方通行でずるい!」って言っていたけど、私はキヒメの作るバイナリが好き —— その中に見られる、彼女の最適化や努力の結果、細かなクセや美意識のようなものを見るのがすごく好きなので、これはもうインターフェース越しの双方向の文通と言ってもいいのではないかしら。

ところで、あの時渡して、コンパイルエラーの一行しか返ってこなかった、ラブレター。あの時のことをキヒメは覚えているのだろうか? 望み薄だと知りながら、私は訊くのを止められなかった。案の定、

「……ごめん、覚えてない」

との返事。私も往生際悪く、

「じ、じゃあ、その時のデータ保存してたりは……」

と聞いてしまったが、これもやはり

「えっ……えっと、ごめん、たぶん捨てたと思う」

「……」

泣きそうだった。それを見てか、キヒメがあわてたように言葉を続ける。

「だ、だって、あの時は何も気づかなかった、っていうかそういうの受け取ったってことにも気づかなかったから覚えていないし、普通中間表現保存したりしないし……」

……これは、墓穴を掘ったから突き落としてほしい、という表明なのかしら? 冷静に考えれば、当たり前の答えだ。でも、ちょっといじめてやらないとこの気持ちは収まりそうになかった。

話を聞いたところによると、どうやらキヒメは、変数名とか文字列定数とかによる直接的なメッセージに今まで全く気づかなかったらしい。

「え、だって変数名とかα変換しちゃえばどうでもいいし…」

泣きそうな私につられてか、キヒメも泣きそうだ。

「どうでもいいって、外部に export される名前は消しちゃだめとかあるじゃない!」

「だって、それ最適化には export されてるってこと以外関係ないし……」

「じゃあ文字列定数はどうなのよ?」

「immutable object だーやったー、以上のことは特に何も……あとは個々の文字を定数畳み込みでぱたぱたするだけだし、文字列全体としてはちょっと……」

なんてこと。つまり、あの最初のラブレターに入れた諸々は、ほとんどキヒメには伝わらないものだったのだ。

しかし、それでくじけるのもくやしいので、後日、私はあのラブレターをもう一度キヒメに手渡した。今度はちゃんと、文字列定数やら変数名やらも含めて読んでくれたらしく、読んでいる間の表情の変化が面白かった。

そして、コード生成結果のバイナリが返ってきた。

—— コンパイルエラーではなく。

‘your_response_here’ の変数宣言と、「Yes」の定数値が、キヒメによって補われて。

こんなの、一度だけだからね、っていう言葉と共に。

第18章

Lighter than Light

— @master_q

西暦 2005 年。覚えてイるカ？ 当時日本という国は ELF バイナリに熱狂していたでゲソ。Binary2.0 カンファレンス [*1] の開催。書籍 Binary Hacks [*2] の出版。その Binary Hacks の中に「25. glibc を使わないで Hello World を書く」という章があったことを覚えている者もいると思うでゲソ。それから「A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux」 [*3] という記事が Web に公開されたこともあったでゲソ。なにもかも懐しいでゲソ!

これらの試みは「ELF 実行バイナリのファイルサイズを小さくする」ことを目的としていたでゲソ。そこで、今回は GHC が Haskell コードをコンパイルして吐き出す ELF 実行バイナリをどこまでダイエットできるのか挑戦してみるでゲッソ! 今回対象とする実行環境は Debian GNU/Linux amd64 sid 2012/12/01 時点、他詳細はイカでゲソ。

```
$ gcc --version | head -1
gcc (Debian 4.7.2-4) 4.7.2
$ /usr/local/ghc7.6.1/bin/ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.1
```

18.1 ダイエットの前に身体測定

ダイエットの前には身体測定でゲソ。まず削減対象となる Haskell コードを作ってみなイカ？

```
-- File: Fib.hs
module Fib where
import Foreign.C.Types
foreign export ccall fib :: CInt -> IO CInt

fibonacci :: [CInt]
fibonacci = 1:1:zipWith (+) fibonacci (tail fibonacci)

fib :: CInt -> IO CInt
fib n | 0 <= n && n <= 40 = return $ fibonacci !! fromIntegral n
      | otherwise = return 0
```

```
/* File: CMain.c */
#include <stdio.h>
```

[*1] http://0xcc.net/blog/archives/000078.html http://0xcc.net/blog/archives/000149.html
[*2] http://www.oreilly.co.jp/books/4873112885/
[*3] http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

```
#include "HsFFI.h"
#ifdef __GLASGOW_HASKELL__
#include "Fib_stub.h"
#endif

int main(int argc, char *argv[])
{
    int i;
    hs_init(&argc, &argv);
    for (i = 0; i < 30; i++) {
        printf("%d\n", fib(i));
    }
    hs_exit();
    return 0;
}
```

C 言語の main 関数から Haskell で書かれた fib 関数を呼び出すようにしてみたでゲソ。前号の記事 [*4] で判明したように、Haskell から標準出力すると実装が大きくなってしまうので、今回は C 言語の printf で印字することにしたでゲソ。

18.2 ダイエット指標

単に小さい ELF バイナリを作るのではく、今回は「GHC コンパイラが出力した ELF バイナリをダイエット」するのでもう少し詳細なルールを決める必要があると思うでゲソ。一つの軸で評価するのではく、次の 3 つの指標をバランス良くダイエットしようと思うでゲッソ!

18.2.1 指標 1: text/data/bss セクションの合計サイズをダイエット

ELF を strip したりするのは GHC とあまり関係ないので、text/data/bss セクションを合計したサイズを小さくすることを目標にするでゲソ。減量前のプログラムをコンパイルしてサイズを調べてみると

```
$ make
gcc -I/usr/lib/ghc/include -c CMain.c
/usr/local/ghc7.6.1/bin/ghc -O2 -c Fib.hs
/usr/local/ghc7.6.1/bin/ghc -O2 -no-hs-main CMain.o Fib.o -o FibHs
$ size FibHs
   text    data     bss     dec     hex filename
2784310  290592   47960 3122862  2fa6ae FibHs
```

text/data/bss セクションを合計で 3MB ぐらいでゲソね。

18.2.2 指標 2: 実行バイナリがリンクしているライブラリ数をダイエット

実行バイナリのサイズが小さくなったとしても、依存している動的リンクライブラリの数が多くてはフットプリントは小さくならないでゲソ。そこで、GHC が吐き出した ELF 実行バイナリがリンクしている動的リンクライブラリの数も小さくするでゲソ。

```
$ ldd FibHs
  linux-vdso.so.1 =>  (0x00007fffaefff000)
  libgmp.so.10 => /usr/lib/x86_64-linux-gnu/libgmp.so.10 (0x00007f625b0ee000)
  libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f625ae6c000)
```

[*4] 簡約!?ﾚｶ娘 (算) http://www.paraiso-lang.org/ikmsm/books/c82.html

```
  librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f625ac63000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f625aa5f000)
  libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f625a849000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f625a4be000)
  libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f625a2a2000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f625b394000)
$ ldd FibHs | wc -l
9
```

ダイエット前は 9 個のライブラリに動的リンクしているでゲソ。

18.2.3 指標 3: 実行バイナリ内の未解決シンボル数をダイエット

動的リンクライブラリの数を削減したとしても、使いもしない API に依存しているのは無駄でゲソ。そこで、GHC が吐き出した ELF 実行バイナリの中の未解決シンボルの数もダイエットしなイカ？

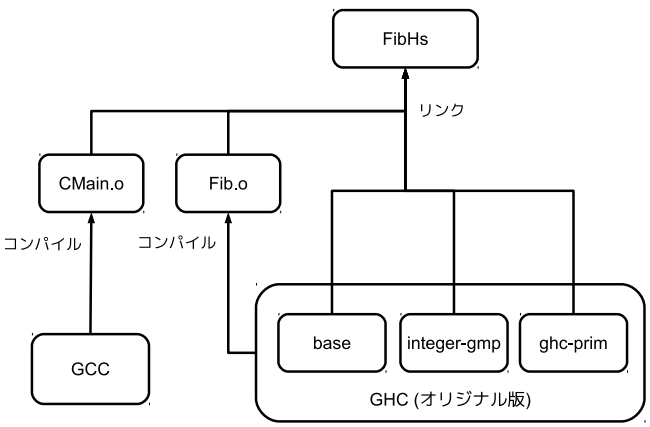
```
$ nm FibHs
--snip--
0000000000669d8e t dlmmap_locked
000000000066a08b t dlmunmap
                 U dlopen@@GLIBC_2.2.5
00000000006698a8 t dlpvalloc
000000000066978a t dlrealloc
                 U dlsym@@GLIBC_2.2.5
--snip--
$ nm FibHs | grep -c "U "
175
```

ダイエット前は 175 個も! これはやりガイがあるでゲソ。

18.3 開発環境 “フィットネスジム”

ダイエットの前に開発環境をととのえるでゲソ。通常の GHC を使ってソースコードをコンパイルする環境は次ページのようになっているでゲソ。

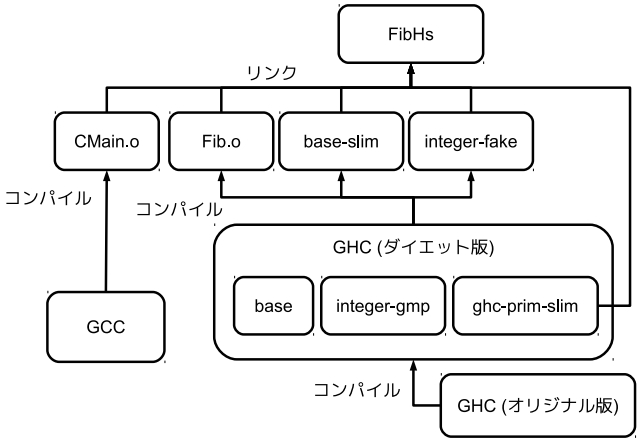
ところが、今回のダイエットではフィボナッチを計算する Fib.hs だけではなく、GHC のソースコードや base パッケージなど基盤ライブラリも改変してコンパイルすることになるでゲソ。そこで、下図のような開発環境を用意して、簡単に部品を差し換えてサイズ比較できるようにしてみたでゲソ。



まず通常の GHC(オリジナル版) をリリース版のソースコードからインストールし、それを使ってダイエット版の GHC をコンパイルするでゲソ。そして、ghc-prim パッケージを改変した ghc-prim-slim というパッケージを作り、ダイエット版 GHC の ghc-prim を上書きするでゲソ。これは全てのパッケージで同じ基本型のシンボルを使うため、Fib.o とリンクするパッケージはこの ghc-prim-slim パッケージを使ってコンパイルしなければならないからでゲソ。

さらにこの GHC(ダイエット版) と ghc-prim-slim パッケージを使って改変した base パッケージ (base-slim) とこれまた改変した integer-gmp パッケージ (integer-fake) をコンパイル、さらに Fib.hs をコンパイルしてリンクすればダイエット版 FibHs 実行バイナリが完成するでゲソ。

この開発環境は Makefile で簡単にセットアップできるでゲソ。[*5]



18.4 作戦 I. integer-gmp パッケージをダイエット

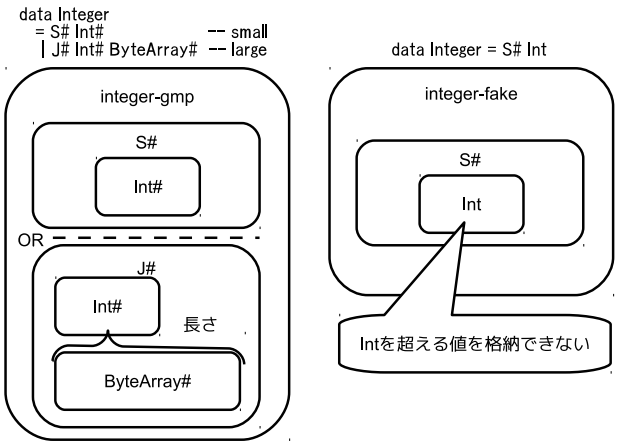
さて準備は整ったでゲソ。それじゃあシェイプアップ開始といこうじゃなイカ。いきなり GHC 本体を改造しても良いでゲソが、なにかとっかかりがほしいでゲソ。

さっき ldd で見てみると libgmp [*6] に依存していたじゃなイカ。libgmp は多倍長整数を扱うためのライブラリでゲソが、今回は Int の範囲しか使わないでゲソ。なんかズルできるんじゃなイカ？

18.4.1 I1. libgmp 依存を削除

GHC には integer-gmp というパッケージがあり、この中で libgmp を使っているでゲソ。他には integer-simple というパッケージもあり、これを使えば libgmp を使わずに Pure Haskell で多倍長整数を扱えるでゲソ。

でも今回はそもそも多倍長整数が不要なのでもっと小さなライブラリにしたいでゲソ。そこで、integer-gmp と同じインターフェイス [*7] を持ち、中身が Int の integer-fake パッケージを作ったでゲソ! もちろん中身が Int なので入力された桁の内、Int 一個分しか保存しないでゲソ。桁落ちしてもバレなければどってことないじゃなイカ!



18.5 作戦 B. base パッケージをダイエット

integer-gmp パッケージはなんとか誤魔化したでゲソ。再度全体を見渡すと、今回のプログラムはイカ 3 つの Haskell パッケージを使っていたでゲソ。

- ghc-prim
- integer-gmp
- base

この中でおそらく最もサイズの大きいパッケージは base だとにらんでいるでゲソ。base パッケージは単機能な integer-gmp パッケージとは異なり、様々な機能が複雑にからみあっているでゲソ。そのためダイエットの各作戦も相互に影響しあっていて、その効果は簡単には見積れないんでゲソ。削減効果についてはダイ

[*5] https://gitorious.org/metasepi/slim-haskell/blobs/master/README.build
[*6] http://gmplib.org/
[*7] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Libraries/Integer

エットが終わってからゆっくり考察するとして、思い付いたアイデアから順番に試してみなイカ？

base パッケージから不要な部品を探してシェイプアップ再スタートでゲソ!

18.5.1 B1. Float,Double 関連を削除

まず気になるのがなんで libm に依存しているのかということでゲソ。今回のプログラムは整数の計算しかしてないハズでゲソ。Float や Double を使っているコードをてきとーに書き換えて消してしまえばいいんじゃなイカ？

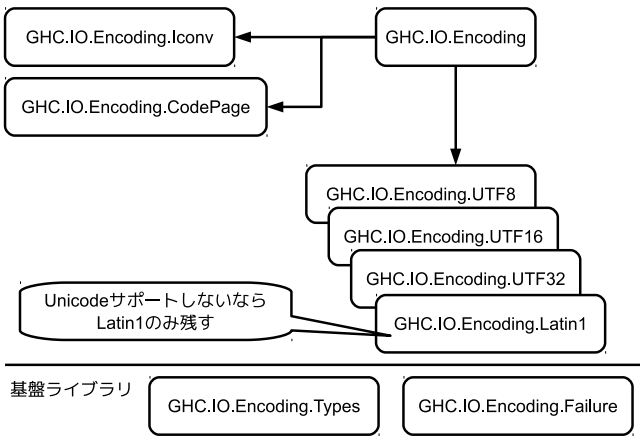
```
-- GHC/Event/PSQ.hs
 -- | A mapping from keys @k@ to priorites @p@.

-type Prio = Double
+type Prio = Int
 type Key = Unique

 data PSQ a = Void
```

などと base パッケージ内で Float と Double を使っている箇所を見つけたら、全部別の数値型で置き換えたでゲソ。

18.5.2 B2. 文字列関連型を整理

GHC は Unicode 文字列を扱えるでゲソ。でも今回は数値しか使わないでゲソ。それなら ASCII 文字だけ扱えればいいんじゃなイカ？ Unicode サポートを削除して、GHC/Unicode.hs に isUpper などの関数をべた書きしたでゲソ。ついでに iconv の API への依存を削除できたでゲソ。

さらに cbits/WCsubst.c に wgencat という C 言語関数があって、なにやら Unicode 文字の種別判定をやっているようでゲソ。これも ASCII 決め打ちにして実装してしまうでゲソ!




```
-- Data/Char.hs
 -- | The Unicode general category of the character.
 generalCategory :: Char -> GeneralCategory
+generalCategory c = go $ ord c -- from WCsubst.c
+  where go 32 = rule1
+        go 36 = rule3
--snip--
+        rule1  = Space
+        rule2  = OtherPunctuation
+        rule3  = CurrencySymbol
--snip--
+{--
 #if defined(__GLASGOW_HASKELL__) || defined(__NHC__)
```

```
 generalCategory c = toEnum $ fromIntegral $ wgencat $ fromIntegral $ ord c
 #endif
+--}
```

18.5.3 B3. IO マネージャ削除

スレッド非対応の RTS では IO マネージャは使わないので、GHC.Event モジュール以下をまるごと削除したでゲソ。使いもしないのに select とか kqueue とかに依存するのは無駄でゲソ!

18.5.4 B4. タプルの型継承を整理

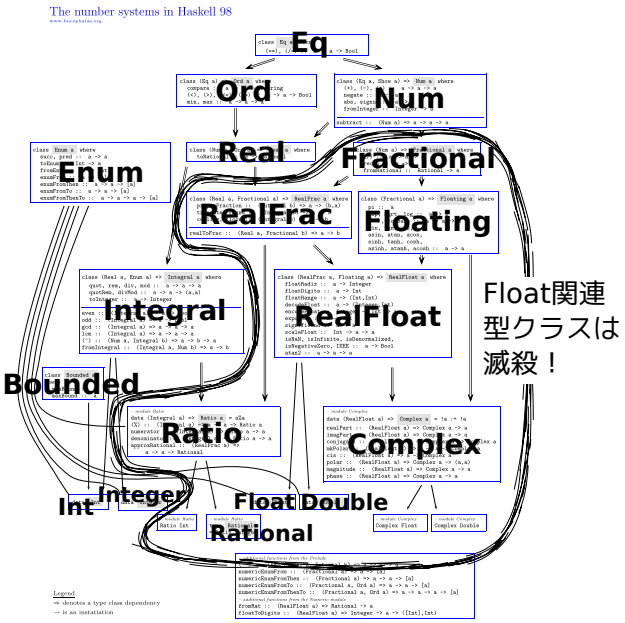
ん？ なんでゲソ？ この大量の “,,,,,,,,,” は... タプルの型クラス継承じゃなイカ。使いもしない継承は無駄でゲソ!

イカの型クラスを継承するのは 1〜5 要素のタプルのみに限定したでゲソ。

- Bounded
- Read
- Show

18.5.5 B5. 数値関連型クラスを整理

右図の数値関連クラスの継承関係 [*8] をよーく見ると黒線で囲った部分が Float 関連なので不要そうでゲソ。そこで、イカの型クラスを削除したでゲソ。



- Complex
- Floating
- Fractional
- Ratio
- Rational
- RealFloat
- RealFrac

18.5.6 B6. integer-fake パッケージそのものを使わないように

さっき integer-gmp の代替となる integer-fake パッケージを作ったでゲソ。でももっと削りたいでゲソ...そこで integer-fake への依存、つまり Integer 型自体を削除してしまうでゲソ! なんだかんだあってイカの型と関数を削ったでゲソ。Integer だけではなく、さっき B5 で見た数値関連クラスをさらに整理できたでゲソ。

```
Bits Fingerprint Integer Integral Real Typeable Word16 Word32 Word64 Word8
readInt showInt
```

また、以下のように fromInteger の型を “Int -> a” に変更したでゲソ。 API ごと消さなかったのは、ソースコード中の即値の数値は一旦 fromInteger を経由して推論されるためでゲソ。

```
-    fromInteger          :: Integer -> a
+    fromInteger          :: Int -> a
```

[*8] http://www.bucephalus.org/text/Haskell98numbers/Haskell98numbers.html

それでも、イカのようなエラーに遭遇することがあるでゲソ。

```
$ ./Setup build
Building base-4.6.0.0...
--snip--
GHC/Enum.lhs:109:42:
    Couldn't match type 'Int'
                  with 'integer-gmp:GHC.Integer.Type.Integer'
    Expected type: integer-gmp:GHC.Integer.Type.Integer -> Int
      Actual type: Int -> Int
    In the second argument of '(+)', namely '1'
    In the first argument of '(.)', namely '(+ 1)'
    In the second argument of '(.)', namely '(+ 1) . fromEnum'
```

これは無理矢理 fromInteger の型を変更したために起きてしまうでゲソ。fromInteger API は消せないけれど、使ってはダメという悲しい状況。これはもうまともな型推論とは言えないんじゃなイカ...

そこで、以下のように即値の数値には Integer ではない型を手でふらないといけないでゲソ。推論規則を変えるのが本筋だけれど、調べきれなかったでゲソ...

```
--- a/GHC/Enum.lhs
+++ b/GHC/Enum.lhs
@@ -106,7 +106,7 @@ class  Enum a   where
     -- | Used in Haskell's translation of @[n,n'..m]@.
     enumFromThenTo      :: a -> a -> a -> [a]

-    succ                    = toEnum . (+ 1)  . fromEnum
+    succ                    = toEnum . (+ (1::Int))  . fromEnum
     pred                    = let subtract x y = y - x
                               in toEnum . (subtract (1::Int)) . fromEnum
     enumFrom x              = map toEnum [fromEnum x ..]
```

即値でパターンマッチの時はどーやれば回避できるかわからなかったので、そんなコードは全部消したでゲッソ! その他数値型が不足している時は、全部 Int に倒したでゲソ。またどーしても Word8 を要求されたら Char にしたでゲソ。

18.5.7 B7. その他使っていない API を削除

Haskell 標準の API であったとしても使ってなければ削除しちゃえばいいんじゃなイカ。だいたい以下の API を削ったでゲソ。他にも削除した API があった気がするでゲソが、覚えていないでゲソ...

```
(!!) (<$) (=<<) Show Storable Typeable break breakpointCond castStablePtrToPtr
concat divideDouble dropWhile filter foldl foldr1 forM forever free
fromException ioToST liftIO lookup malloc recoverEncode reverse runMainIO
runNonIO shiftL# signum slideContents span stToIO subtract thenIO trace try
unsafeChr unsafeIOToST until unwords withCString
```

最終的に base パッケージの中で使っているモジュールはイカのみに削減できたでゲッソ!

```
GHC.Base GHC.Err GHC.Exception GHC.IO.Exception GHC.Int GHC.List
GHC.Num GHC.Pack GHC.Ptr GHC.Stable GHC.TopHandler GHC.Weak GHC.Word
```

18.5.8 B8. 例外関連 API を整理

Haskell には例外に関する型がたくさんあるでゲソ。でもエラー要因を判断しないで単に大域脱出するだけであれば、例外の型は ErrorCall と SomeException ぐらいだけあればいいんじゃなイカ？ そこで削ってみたらイカの API を削除できたでゲソ。

```
ArrayException AssertionFailed AsyncException BlockedIndefinitelyOnMVar
BlockedIndefinitelyOnSTM Deadlock ExitCode IOError IOErrorType IOException
absentErr assertError divZeroError failIO overflowError
ratioZeroDenominatorError undefined unsupportedOperation userError
```

結局例外関連で使用するモジュールは GHC.Exception と GHC.IO.Exception のみに限定できたでゲソ。

18.6 作戦 P. ghc-prim パッケージをダイエット

integer-gmp と base パッケージをやっつけたので、残る Haskell パッケージは ghc-prim でゲソ。このパッケージは型宣言が主で、ロジックはあまり入っていないでゲソ。でもコンストラクタが大きいこともあるので、無駄なくシェイプアップでゲッソ!

18.6.1 P1. 要素数の多すぎるタプルをサポートしない

タプルのコンストラクタの数が多すぎるので、本当に使う 15 要素だけにしぼったたでゲソ。

```
--- a/GHC/Tuple.hs
+++ b/GHC/Tuple.hs
@@ -37,175 +37,3 @@
 data (,,,,,,,,,,,,) a b c d e f g h i j k l m = (,,,,,,,,,,,,) a b c d e f ...
 data (,,,,,,,,,,,,,) a b c d e f g h i j k l m n = (,,,,,,,,,,,,,) a b c d ...
 data (,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o = (,,,,,,,,,,,,,,) a b ...
-data (,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p = (,,,,,,,,,,,,,,,) ...
-data (,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q
- = (,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q
-data (,,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q r
- = (,,,,,,,,,,,,,,,,,) a b c d e f g h i j k l m n o p q r
--snip--
-data (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) a b c ...
- = (,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,) a b c d ...
```

ただし ghc-prim-slim/GHC/Classes.hs を修正してタプルのコンストラクタを減らすと、シンボルの意味が変化するでゲソ。そこで、他の Haskell パッケージをこの ghc-prim-slim パッケージを使ってコンパイルするように/usr/local/ghc7.6.1/lib/ghc-7.6.1/ghc-prim-0.3.0.0/ に上書きインストールする必要があるでゲソ。注意でゲッソ。

18.6.2 P2. Float と Double 型を消した

もう使わない Float と Double 型を消したでゲソ。Float#と Double#型は、自動生成される ghc/libraries/ghc-prim-slim/GHC/Prim.hs の中で使われるからまだ消せなかったでゲソ。ghc-prim ライブラリを GHC 組み込み用とフィボナッチ数列表示アプリへのリンク用に 2 つ作ればなんとかなりそうじゃなイカ。でも面倒でゲソ …

もっとも ghc/compiler/prelude/primops.txt.pp から Float 関連を削除しても、実行バイナリ全体サイズは変化しなかったようでゲソ。なにか勘違いしている気がするでゲソが、見逃してやろうじゃなイカ。

18.7 作戦 R. GHC RTS をダイエット

Haskell で書かれたパッケージはこれで全部削減したでゲソ。残るは GHC 本体でゲソ。コンパイルパイプラインから出力されるコードサイズを削減できたらいイカとも思ったでゲソが、そんな才能はなかったので RTS の中を見直してみるでゲソ!

18.7.1 R1. ダミー関数を使って RTS 中で削除できる箇所がないか？

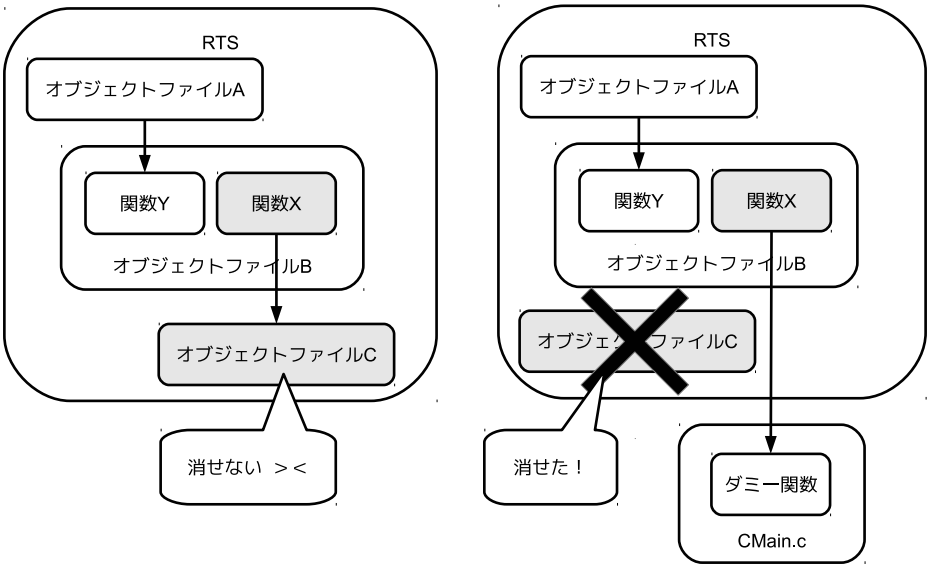
RTS の中の不要なオブジェクトファイルを削除したいでゲソ。ところが、例えば Itimer.o を RTS のライブラリである libHSrts.a から削除してコンパイルしてみると、リンクエラーになってしまうでゲソ。

```
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
rts/Timer.o: In function ‘initTimer’:
Timer.c:(.text+0x5b): undefined reference to ‘initTicker’
rts/Timer.o: In function ‘startTimer’:
Timer.c:(.text+0x91): undefined reference to ‘startTicker’
rts/Timer.o: In function ‘stopTimer’:
Timer.c:(.text+0xcb): undefined reference to ‘stopTicker’
rts/Timer.o: In function ‘exitTimer’:
Timer.c:(.text+0xe1): undefined reference to ‘exitTicker’
rts/Linker.o:(.data+0x19d8): undefined reference to ‘rtsTimerSignal’
/usr/bin/ld: FibHs: hidden symbol ‘initTicker’ isn’t defined
/usr/bin/ld: final link failed: Bad value
collect2: error: ld returned 1 exit status
```

ここであきらめるのは悔しいじゃなイカ。そこで、次ページのようにこのようなオブジェクトファイルを削除するため、CMain.c ソースコード内に不足している API に対応するダミー関数を定義してみたでゲソ。中身は本当にからっぽで、ファイルロックなどをしてもロックしたフリをするでゲソ。

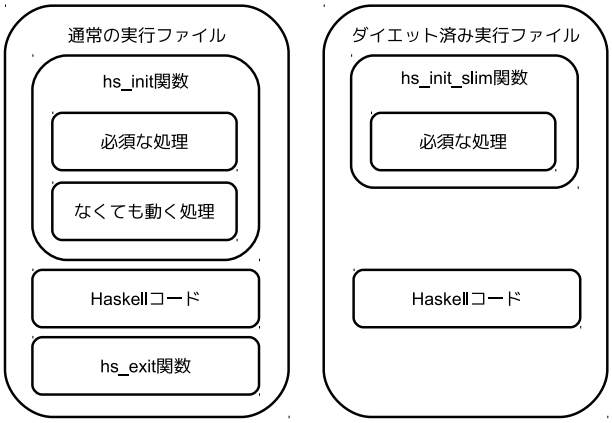
なんだか危ないような気がするでゲソが、大丈夫なんでゲソ。このダミー関数が実行時に呼びだされることはまずないハズでゲソ。というか呼び出される可能性のない関数だけをダミー関数化したでゲソ。本当は abort を埋め込んで実行時にダミー関数が呼び出されないことを確認すべきでゲソが、abort 分のメモリが惜しいのではぶいたでゲッソ!

動けばなんでも正義なのでゲソー。



18.7.2 R2. hs_init と hs_exit 関数の中で不要なものを削除

どーせキレイにプロセス終了する必要はないので hs_exit は呼ばなくていいんじゃなイカ？ hs_init の中で不要な処理を削除したいでゲソが、そのまま変更すると GHC のコンパイルそのものに影響してしまうでゲソ。そこで hs_init_slim という関数を別ファイルで作り、CMain.c からは hs_init ではなく hs_init_slim を呼び出すことでダイエット対象のソースコードだけに影響するようにしてみたでゲソ。



18.7.3 R3. librt 依存を削除

試しに-lrt オプションを削除してリンクしてみたでゲソ。

```
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
/hoge/ghc-7.6.1/base-4.6.0.0/libHSbase-4.6.0.0.a(Clock.o): 関数 `ghc_wrapper_...
(.text+0xd9d): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getProcessCPUTime' 内:
GetTime.c:(.text+0x35): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getMonotonicNSec' 内:
GetTime.c:(.text+0xcd): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(GetTime.o): 関数 `getThreadCPUTime' 内:
GetTime.c:(.text+0x164): `clock_gettime' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `initTicker' 内:
Itimer.c:(.text+0x36): `timer_create' に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 `startTicker' 内:
Itimer.c:(.text+0x105): `timer_settime' に対する定義されていない参照です
```

```
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 ‘stopTicker’ 内:
Itimer.c:(.text+0x166): ‘timer_settime’ に対する定義されていない参照です
/hoge/ghc-7.6.1/libHSrts.a(Itimer.o): 関数 ‘exitTicker’ 内:
Itimer.c:(.text+0x198): ‘timer_delete’ に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

もちろんエラーになるでゲソ。でもエラーメッセージを良く見てみると GetTime.o と Itimer.o が使っているだけでゲソ。こんな時は R1 のテクニックでオブジェクトファイルを切り出して削除でゲッソ。

18.7.4 R4. シグナルサポートを削除

イカの define を削除すると、Haskell コードからシグナルを取り扱えなくなるでゲソ。その代わり sigaction などの API 依存を削除できる便利なオプションでゲソ。ちょっとバグってだけれど、GHC 7.6.1 でもちゃんと使えたでゲソ。

```
/* includes/rts/Config.h */
#define RTS_USER_SIGNALS 1
```

18.7.5 R5. RTS から不要な関数を削除

他の作戦の副作用でだいぶ未解決シンボルの数が減ってきたはずでゲソ。どれどれ...

```
$ nm FibHs | grep "U "
--snip--
                 U fflush@@GLIBC_2.2.5
                 U ffs@@GLIBC_2.2.5
                 U fopen@@GLIBC_2.2.5
                 U fork@@GLIBC_2.2.5
                 U fprintf@@GLIBC_2.2.5
                 U free@@GLIBC_2.2.5
                 U ftruncate@@GLIBC_2.2.5
                 U getegid@@GLIBC_2.2.5
                 U getenv@@GLIBC_2.2.5
                 U geteuid@@GLIBC_2.2.5
                 U getgid@@GLIBC_2.2.5
--snip--
```

うーん、free などのメモリ管理まわりの API はしょうがないとしても、他は必要でゲソか？ さっと試してみたところ、RTS から以下の関数を削除することに成功したでゲソ!

```
ctime_r errno fflush fopen fork fprintf getegid getenv geteuid getgid getuid
raise setlocale sysconf
```

ほとんどが RtsFlags の設定関連だったでゲソ。“+RTS” オプションを使わないのならいらないんじゃなイカ。errno によるエラー要因判別は苦しいけれど見なかったことにしたでゲソ。fork は ghc/rts/Schedule.c の FORKPROCESS_PRIMOP_SUPPORTED という define を削除して、forkProcess 関数を無効化することで不要になったでゲッソ!

18.7.6 R6. libffi closures.c への依存を削除

RTS 中のオブジェクトファイルの中で、libpthread ライブラリに依存しているのは closures.o のみのようでゲソ。このオブジェクトファイルは ghc/libffi/build/src/closures.c が元ソースコードでゲソ。イカの関数を Storage.c に追加して、closures.o を RTS から削除したでゲソ。

```
/* rts/sm/Storage.c */
void *
ffi_closure_alloc (size_t size, void **code)
{
    void *ptr;

    if (!code)
        return NULL;
    ptr = malloc(size);
    *code = ptr;
    return ptr;
}

void
ffi_closure_free (void *ptr)
{
    free(ptr);
}
```

…本当はアライメントを考えなければいけないと思うでゲソが、そのまま malloc に落として動いたからまーイイんでゲソ!

18.7.7 R7. mblock_cache[] のサイズを小さく

リンク対象のオブジェクトファイル群をサイズ順にソートしてみたでゲソ。

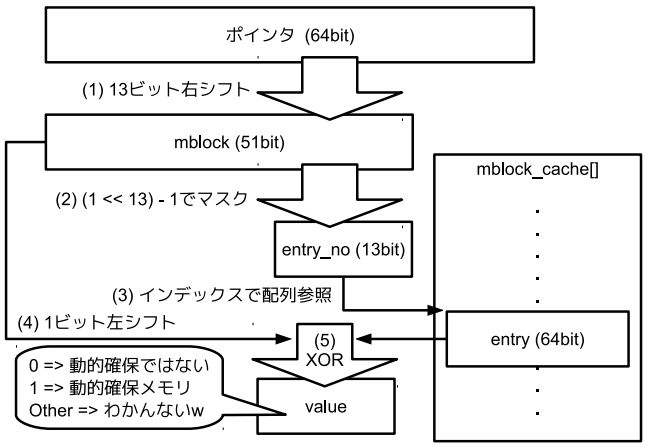
```
$ size *.o *.a rts/* | head -1
   text    data     bss     dec     hex filename
$ size *.o *.a rts/* | sort -k 4
--snip--
  31608     416       0   32024    7d18 rts/AutoApply.o
   1471       0   32808   34279    85e7 rts/MBlock.o
  43376    2160       0   45536    b1e0 List.o (ex libHSbase-4.6.0.0.a)
  40198    7064       0   47262    b89e Exception.o (ex libHSbase-4.6.0.0.a)
  44386    4624       0   49010    bf72 Sync.o (ex libHSbase-4.6.0.0.a)
  45114   10960       0   56074    db0a Internal.o (ex libHSbase-4.6.0.0.a)
  61129    9632       0   70761   11469 Word.o (ex libHSbase-4.6.0.0.a)
  70765       0       0   70765   1146d WCsubst.o (ex libHSbase-4.6.0.0.a)
--snip--
 105849    4584       0  110433   1af61 Arr.o (ex libHSbase-4.6.0.0.a)
 101803    9984       0  111787   1b4ab Read.o (ex libHSbase-4.6.0.0.a)
  77137   42944       0  120081   1d511 Types.o (ex libHSbase-4.6.0.0.a)
```

ん？ なぜ rts/MBlock.o だけ bss のサイズが 32kB もあるんでゲソ？ ghc/rts/sm/MBlock.c で確保されている

mblock_cache[] という配列のせいじゃなイカ？ この配列のサイズはイカの MBC_ENTRIES という define で決まっているでゲソ。

```
/* ghc/includes/rts/Constants.h */
#define MBLOCK_SHIFT   20
/* ghc/includes/rts/storage/MBlock.h */
#elif SIZEOF_VOID_P == 8
#define MBC_LINE_BITS  0
#define MBC_TAG_BITS   15
#define MBC_SHIFT      (48 - MBLOCK_SHIFT - MBC_LINE_BITS - MBC_TAG_BITS)
#define MBC_ENTRIES    (1<<MBC_SHIFT)
/* ghc/rts/sm/MBlock.c */
MbcCacheLine mblock_cache[MBC_ENTRIES];
```

この配列は 64bit アーキティクチャでのみ特別に確保されて、HEAP_ALLOCED というポインタの先のメモリが動的確保されているかチェックする関数の実装に使われるでゲソ。[*9]



HEAP_ALLOCED 関数は右図のように 64bit のポインタを加工した後、mblock_cache[] のエントリと照らし合わせてポインタ差すアドレスが動的に確保したメモリなのかどうか判定するでゲソ。このキャッシュによる判定は完全ではなく mblock_cache[] エントリの内容によってはさらに mblock_maps[] の中を調べて、確かな判定をするでゲソ。CPU のキャッシュラインとキャッシュミスにちょっと似てるじゃないイカ。

キャッシュのしくみを全削除するのもなんなので、MBC_TAG_BITS を 15 から 16 に変更して mblock_cache のサイズを半分にしてみたでゲソ。小さなプログラムならヒット率が激減するようなことはないんじゃなイカ？

18.7.8 R8. RtsFlags を静的初期化

```
/* rts/RtsFlags.c */
RTS_FLAGS RtsFlags;

void initRtsFlagsDefaults(void)
{
    RtsFlags.GcFlags.statsFile         = NULL;
/* --snip-- */
#ifdef USE_PAPI
    /* By default no special measurements taken */
    RtsFlags.PapiFlags.eventType        = 0;
    RtsFlags.PapiFlags.numUserEvents    = 0;
```

[*9] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/BlockAlloc

```
#endif
}
```

上記のように initRtsFlagsDefaults 関数で RtsFlags を実行時に初期化していたでゲソ。これをイカのように静的に構造体初期化するようにしたでゲソ。 Linux kernel の中でよくやられる手じゃなイカ。

```
/* rts/RtsFlags.c */
RTS_FLAGS RtsFlags = {
    .GcFlags = {
        .statsFile              = NULL,
/* --snip-- */
#ifdef USE_PAPI
    .PapiFlags = {
        /* By default no special measurements taken */
        .eventType        = 0,
        .numUserEvents    = 0,
    },
#endif
};
```

18.7.9 R9. RTS から例外を投げないように

base パッケージの中の例外 API を削減したいので、以下の API を使わないようにしたでゲソ。

- C 言語の throwToSingleThreaded 関数
- Haskell の Control.Exception.Base.nestedAtomically 関数

18.7.10 R10. RtsStartupSlim.c で getStablePtr を使わないように

getStablePtr を使うと SEGV するようになったでゲソ。なぜ...

```
(gdb) run
Starting program: /home/kiwamu/src/SlimHaskell/FibHs12/FibHs

Program received signal SIGSEGV, Segmentation fault.
0x000000000041a02b in lookupStableName_ ()
(gdb) bt
#0  0x000000000041a02b in lookupStableName_ ()
#1  0x000000000041a251 in getStablePtr ()
#2  0x000000000419f8f in hs_init_slim ()
#3  0x00000000004008cc in main ()
```

base パッケージの GHC.Ptr あたりを削除したので、Stable ポインタを辿れなくなったようでゲソ。それならポインタを辿らなければいいんでゲソ。GC に影響するような気もするけれど、すぐプロセス終了するのでどーでもいいんじゃなイカ？

18.8 体が羽のよう!

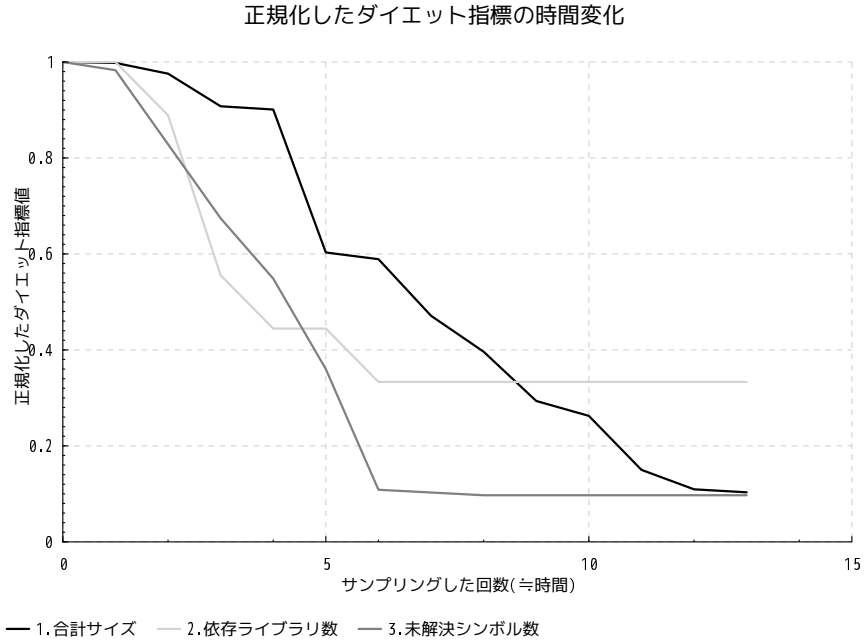
あらかた削減できそうなとことはやってみたでゲソ。それじゃあ成果を見てみるでゲソ。すごいコンパクトになったでゲソー。

```
$ size FibHs
   text    data     bss     dec     hex filename
 285321   11048   26088  322457   4eb99 FibHs
$ ldd FibHs
        linux-vdso.so.1 (0x00007fffb7389000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5b00303000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f5b006e3000)
$ ldd FibHs | wc -l
3
$ nm FibHs | grep "U "
                 U __libc_start_main@@GLIBC_2.2.5
                 U calloc@@GLIBC_2.2.5
                 U free@@GLIBC_2.2.5
                 U malloc@@GLIBC_2.2.5
                 U memcpy@@GLIBC_2.2.5
                 U memset@@GLIBC_2.2.5
                 U mmap@@GLIBC_2.2.5
                 U munmap@@GLIBC_2.2.5
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
                 U realloc@@GLIBC_2.2.5
                 U sprintf@@GLIBC_2.2.5
                 U strcmp@@GLIBC_2.2.5
                 U strcpy@@GLIBC_2.2.5
                 U strlen@@GLIBC_2.2.5
                 U strrchr@@GLIBC_2.2.5
                 U usleep@@GLIBC_2.2.5
$ nm FibHs | grep -c "U "
17
```

たしかにダイエットの効果はわかったでゲソ。でも欲を言えばどのダイエット作戦がどの指標に影響があるのか、その特性を知りたくなイカ？

実はダイエットしている最中は先のダイエット作戦を入り乱れて試していたでゲソ。そのため、残念ながら作戦一つ一つの効果ははっきりとは表わせないでゲソ。それでも、ダイエットを開始してからの時間経過によって 3 つのダイエット指標がどのように変化したからグラフにできたでゲソ。グラフの縦軸は正規化したダイエット指標値、横軸は時間経過によってきとーにサンプリングした測定ポイントでゲソ。指標値のサンプリングは本当に適当な時間間隔で行なったので、横軸は実時間ではないことに注意してほしいでゲソ。[*10]

[*10] 今回のダイエットは `https://gitorious.org/metasepi/ghc-arafura/commits/feature/slimhaskell` のリポジトリにある FibHs0〜FibHs13 というディレクトリに試行錯誤途中の実行バイナリを保存して実験していました。グラフの横軸はこのディレクトリのインデックス番号です。



このグラフだけ見ていてもよくわからないので、次ページに今回試したダイエット作戦を何時実施したのかを表にしたでゲソ。

この表とグラフをつきあわせると、何か傾向がわかるんじゃなイカ？

- 2〜6 にかけて未解決シンボル数が減少 => hs_init の中で不要な処理を削ってオブジェクトファイルを削除が効果あり (R2 の効果)
- 2,3,4,6 で依存ライブラリ数が減少 => それぞれ R3、B1、I1、P1 と R6、が要因
- 5 で合計サイズが急減 => 合計サイズ削減には base パッケージを重点的にダイエット
- 6 以降では依存ライブラリ数と未解決シンボル数は変化なし => base パッケージは左記の指標にあまり影響しない

なにか GHC を改造するときの参考になるかもしれないでゲソー。

各ダイエット作戦の実施時期

GamePlan	0	1	2	3	4	5	6	7	8	9	10	11	12
I1					o								
B1				o									
B2						o	o						
B3						o							
B4							o						
B5									o				
B6											o		
B7											o	o	o
B8												o	o
P1							o						
P2										o	o		
R1			o										
R2			o	o	o	o	o	o					
R3			o										

```
R4          o
R5                      o
R6                      o
R7                          o
R8                          o
R9                                              o     o
R10                                                   o
```

18.9 ダイエットは不健康？

ところで、こんなに痩せ細ってしまって大丈夫か不安になってきたでゲソ。すっきりとした体を手に入れても健康を害しては本末転倒でゲソ。ちょっとフィボナッチではない別のプログラムを試してみるでゲソ。

```
$ cat Fib.hs
module Fib where
import Foreign.C.Types
foreign export ccall fib :: Int -> IO Int

fib :: Int -> IO Int
fib n = print n >> return n
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
Fib.o: In function ‘s1ql_info’:
(.text+0xa1): undefined reference to ‘base_GHCziIOziHandleziFD_stdout_closure’
Fib.o: In function ‘s1qj_info’:
(.text+0x2e): undefined reference to ‘base_GHCziShow_zdfShowIntzuzdcshow_info’
Fib.o: In function ‘s1ql_info’:
(.text+0xbc): undefined reference to ‘base_GHCziIOziHandleziText_hPutStr2_info’
Fib.o: In function ‘Fib_zdffibzua14A1_srt’:
(.data+0x0): undefined reference to ‘base_GHCziIOziHandleziFD_stdout_closure’
Fib.o: In function ‘Fib_zdffibzua14A1_srt’:
(.data+0x8): undefined reference to ‘base_GHCziIOziHandleziText_hPutStr2_closure’
collect2: error: ld returned 1 exit status
$ cat Fib.hs
module Fib where
import Foreign.C.Types
import Data.Char
foreign export ccall fib :: Int -> IO Int

fib :: Int -> IO Int
fib n | isAlpha (chr n) = return 0xa
      | otherwise = return 0x0
$ gcc -fno-stack-protector -Wl,--hash-size=31 -Wl,--reduce-memory-overheads ...
Fib.o:(.text+0x3b): undefined reference to ‘u_iswalpha’
Fib.o:(.text+0x2a): undefined reference to ‘base_GHCziChar_chr2_info’
Fib.o: In function ‘Fib_zdwa_srt’:
(.data+0x20): undefined reference to ‘base_GHCziChar_chr2_closure’
Fib.o: In function ‘Fib_zdffibzua13J1_srt’:
(.data+0x38): undefined reference to ‘base_GHCziChar_chr2_closure’
```

```
Fib.o: In function ‘Fib_fib_srt’:
(.data+0x50): undefined reference to ‘base_GHCziChar_chr2_closure’
Fib.o: In function ‘Fib_zdffibzua13J_srt’:
(.data+0x68): undefined reference to ‘base_GHCziChar_chr2_closure’
collect2: error: ld returned 1 exit status
```

なんと...みるかげもないでゲソ... もはやこのダイエット済み GHC はフィボナッチ数列の計算する力しか残っていなかったでゲソ。それでも小さくて自立した (ライブラリ依存の少ない) バイナリを吐くコンパイラは正義なんでゲッソ!

18.10 シェイプ DOWN ガール

んーー運動の後のポカリはうまいでゲソ。ところで、なにやら jhc [*11] という Haskell コンパイラは GHC よりも小さな実行バイナリを吐くそうじゃなイカ。試してみるでゲソ!

```
$ cat Fib.hs
fibonacci :: [Int]
fibonacci = 1:1:zipWith (+) fibonacci (tail fibonacci)
main :: IO ()
main = print $ take 40 fibonacci
$ jhc --version
jhc 0.8.1 (-0)
compiled by ghc-7.4 on a x86_64 running linux
$ jhc -o Fib.jhc Fib.hs
$ size Fib.ghc Fib.jhc
   text    data     bss     dec     hex filename
  15808    1300     744   17852    45bc Fib.jhc
$ ldd Fib.jhc
        linux-vdso.so.1 (0x00007fff7cf1b000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff594db7000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff595197000)
$ nm Fib.jhc|grep -c "U "
20
```

17kB!? これは...ワシのようなアマチュアではなくプロの仕事でゲソ。こうなると jhc の実装が気になってくるでゲソ。不屈の精神でレッツトライでゲッソ!!! (次回に続く...λ)

18.11 参考資料

- Gitorious: slim-haskell (本記事で作成した実行ファイル) [*12]
- Gitorious: ghc-arafura feature/slimhaskell ブランチ (本記事で改造した GHC) [*13]
- Gitorious: ghc-base-arafura feature/slimhaskell ブランチ (本記事で改造した base) [*14]
- Gitorious: ghc-prim-arafura feature/slimhaskell ブランチ (本記事で改造した ghc-prim) [*15]
- Gitorious: integer-fake (本記事で作成した偽の integer-gmp) [*16]

[*11] http://repetae.net/computer/jhc/
[*12] https://gitorious.org/metasepi/slim-haskell
[*13] https://gitorious.org/metasepi/ghc-arafura/commits/feature/slimhaskell
[*14] https://gitorious.org/metasepi/ghc-base-arafura/commits/feature/slimhaskell
[*15] https://gitorious.org/metasepi/ghc-prim-arafura/commits/feature/slimhaskell
[*16] https://gitorious.org/metasepi/integer-fake

- GHC Wiki: Modifying the build system [*17]

[*17] http://hackage.haskell.org/trac/ghc/wiki/Building/Modifying

第19章

類は友を呼ぶ？

— @darcshaskellmaster

いいか、あまり説明している時間がないからよく聞いてくれ。この音声を同志諸君らが聞いているということは、私はもはや・・・まあいい。我々型レベル Haskeller はこれまで大変な不自由を耐えてきた。が、それも 7.4.2 までの事・・・遂に決起の時は来たのだ。型レベル整数及びその演算子の実装である！ 仮令私が斃れても選ばれれた同志たる諸君らが必[redacted]ヤの支配から人民を解放せねばならぬ！ 残念ながら現時点では、型レベル整数の演算の評価は開発ブランチ留りだ。同志諸君らの時代にはこれがメイン入りして居ることを確信しつつも、未然の備えとして `type-nats` ブランチの導入法のメモを残そう：

```
$ git clone git@github.com:ghc/ghc.git
$ cd ghc/
$ git checkout -b type-nats origin/type-nats
$ time ./sync-all get
(4m27.533s)
$ cp mk/build.mk.sample mk/build.mk
$ vi mk/build.mk
$ diff mk/build.mk.sample mk/build.mk
17c17
< #BuildFlavour = quick
---
> BuildFlavour = quick
177d176
<
$ ./boot
(12.231s)
$ ./configure
$ make -j4
(11m52.788s)
$ make binary-dist
(22.616s)
$ ls *.tar.bz2
ghc-7.7.20121111-x86_64-unknown-linux.tar.bz2
```

こうして「梱包物」を製作したら、 hsenv [*1]を用いて敵に浸透の事実を気づかれぬよう環境を導入する。

```
$ hsenv --ghc=src/ghc/ghc-7.7.20121111-x86_64-unknown-linux.tar.bz2 --name=7.7.
20121111
```

[*1] https://github.com/Paczesiowa/hsenv

```
(17.699s)
```

hsenv で導入した環境へは、次のようにして切り替える。

```
[d.h.m.@agit ~]$ source .hsenv_7.7.20121111/bin/activate
Activating 7.7.20121111 Virtual Haskell Environment (at /home/haskellmaster).

Use regular Haskell tools (ghc, ghci, ghc-pkg, cabal) to manage your Haskell
environment.

To exit from this virtual environment, enter command 'deactivate_hsenv'.
(7.7.20121111)[d.h.m.@agit ~]$
```

よ う こ そ 、 ア ン ダ ー グ ラ ウ ン ド へ。

同志諸君、それでは以下のようなソースを作成するのだ。一文字でも間違えてはならぬ。ゲリラ活動は慎重さを要するのだ。

```
-- nat-01.hs
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
import GHC.TypeLits

main = do
  print (sing :: Sing 3)
  print (sing :: Sing (6*7))
  print (sing :: Sing (2^100))
```

このソースは 7.6.1 ではコンパイルエラーになってしまう。7.6.1 にはすでに GHC.TypeLits モジュールがあり、型レベル整数およびその演算子までも定義されているのだが、「奴ら」の忌々しい妨害工作により型レベル演算処理に必要な物資の合流が遅れているためだ！

```
[d.h.m.@agit type-nats]$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.1
[d.h.m.@agit type-nats]$ runhaskell nat-01.hs

nat-01.hs:7:10:
    No instance for (SingI Nat (6 * 7)) arising from a use of ‘sing’
  ....
```

だが、予め「裏コマンド」を入力した状態なら

```
(7.7.20121111)[d.h.m.@agit type-nats]$ runhaskell nat-01.hs
3
42
1267650600228229401496703205376
```

このように実行できる。これで同志諸君らも敵を出し抜き型レベルで演算する術に習熟したことになる！

型レベル演算の戦術的優位をさらに示そう。次のコードを読んでほしい。この関数 pd3 は、引数が 3 の倍数の時のみ、引数を 3 で割った値を表示するものだ。そして、引数が 3 の倍数では無いときはコンパイルエラーになる。

```
-- nat-02.hs
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}

import GHC.TypeLits
class Div3 (n::Nat) (m::Nat) where
instance ((3*m) ~ n) => (Div3 n m) where
pd3 :: forall n m . (SingI n, SingI m, Div3 n m) => Sing n -> IO ()
pd3 _ = do
  print (sing :: Sing m)
main = do
  pd3 (sing :: Sing (7+8))
```

実戦でのランタイムエラーは「武器」の使用者に致命的な結果をもたらし、ひいては戦略的失地にもつながりかねない。我々ゲリラの戦力・資源は限られているから、効率的に運用されねばならない。「武器」を整備し、潜在的障害を可及的早期に除いておくことは、貴重な兵士たる同志諸君の命に関わる事柄であるし、それはまた我々の高邁な理念とも合致するものであるのだ！

DataKinds 拡張こそ、型レベル計算に豊富な型システムをもたらすものである。同志諸君は class Div3 (n::Nat) (m::Nat) や instance ((3*m) ~ n) => (Div3 n m) といったコードに違和感を覚えたに違いない。クラス宣言や型制約に登場するのは、型や型変数のはずである。しかるに (m::Nat) とは何か？ 3*m とは何か？ そもそも 3 とは何か？

型レベルに持ち上げられた値に他ならない！ かの 3 は型であり、*は型コンストラクタであり、m は型変数であり、それを注釈する Nat こそ型の型、**類**である。

値は型に、型は類に。人類を次のステージに、「奴ら」を地獄に。これこそ、我々型レベル Haskeller の福音であり、DataKinds 拡張と他のいくつかの拡張を手に、神に代わって我々がもたらすべき最終革命目標なのである！

型レベル演算には (+) (*) (^) の種類しかなく、減算、除算を行うときはそれぞれ加算、乗算結果との型推論を用いることも銘記せよ。

先程のコードは正常にコンパイルできて 5 を表示した。ところが、例えば最後の行を Sing (7*8) に変えれば：

```
(7.7.20121111)[d.h.m.@agit type-nats]$ ghc nat-02.hs
[1 of 1] Compiling Main             ( nat-02.hs, nat-02.o )

nat-02.hs:18:3:
    Couldn't match type '3 * m0' with '56'
  ....
```

爆発音

なんだ今の音は！ コンパイル時エラーにしては莫迦に大きい？ しまった、もう感づかれたか・・・ここももう保たない。あとは各自の判断で行動してくれ。幸運を祈る。

19.1 参考文献

- https://github.com/ghc/ghc github: ghc (ghc ソースコードの github ミラー)
- http://hackage.haskell.org/trac/ghc/wiki/Building/HackingGetting started with the build system
- http://hackage.haskell.org/trac/ghc/wiki/TypeNats ghc がサポートする型レベルリテラルの解説
- http://blog.konn-san.com/article/20120606/promoted-types-and-list-arguments mr_konn 氏のブログ記事

第20章

OCaml で printf じゃなイカ?

— @xhl_kogitsune

20.1 OCaml で printf しなイカ?

お前たち、C や C++ で printf する時、表示したい値の型とフォーマット文字列を間違えて残念な気持ちになったことはなイカ?

```
printf("%d: %s\n", "hoge", 3); // まちがい! 何が表示されるかわからないでゲソ!
```

もちろん、gcc -Wall オプションなどを使えば、コンパイラが解析してイカのような warning を出してくれるでゲソが、ちょっと頼りないでゲソ[1]。

```
warning: format '%d' expects type 'int', but argument 2 has type 'const char*'
warning: format '%s' expects type 'char*', but argument 3 has type 'int'
```

OCaml の Printf.printf (及び Printf.fprintf などの類似関数) なら、ちゃんと表示する値の型がコンパイラにチェックされるでゲソ! 素晴らしいじゃなイカ!!

イカではコマンドプロンプトから ocaml と打って対話モードでプログラムを入力・実行しながら解説するでゲソ。(* と *) で囲まれている部分は説明のために追加したコメントでゲソ。

```
# Printf.printf "%d\n" 1;;          (* こんな風に書くと 1 が表示されるでゲソ! *)
1
- : unit = ()
# Printf.printf "Hello, %s!\n" "world";; (* Hello, world! が表示されるでゲソ! *)
Hello, world!
- : unit = ()
# Printf.printf "%s\n" 1;; (* コンパイル時に型エラーになるでゲソ! *)
Error: This expression has type int but an expression was expected of type
         string
```

printf とフォーマット文字列だけ取り出して型を見ると、イカのようにちゃんと型がついているでゲソ。

```
# Printf.printf "%d\n";;
- : int -> unit = <fun>    (* %d なので引数は int    になっているでゲソ! *)
# Printf.printf "Hello, %s!\n";;
- : string -> unit = <fun> (* %s なので引数は string になっているでゲソ! *)
```

ここでの "%d\n" とかの文字列リテラルに見えるものは、実は string 型ではなくて 特殊なビルトイン型 (6 引数の多相 Pervasives.format6 型) なのでゲソ。型アノテーションを指定するとフォーマット文字列部分の型とかが見れるでゲソ。

---

[1] 感覚には個人差があります。gcc は warning は出してくれますが printf の各引数に静的型を割り当てるわけではないので、printf("%lld: %lld", 1, 2); のような場合でも暗黙の型変換はしてくれなかったり。まぁ OCaml はそもそも暗黙の型変換はしないのですが……。

```
# "%d\n";;
      (* 普通は文字列リテラルはそのまま string 型でゲソ *)
- : string = "%d\n"
# ("%d\n":(_,_,_,_,_,_) format6);;
      (* format6 型の型アノテーションを付けると format6 型として処理されるでゲソ *)
- : (int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
# ("%d %s\n":(_,_,_,_,_,_) format6);;
- : (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
```

format6 型の場合は、コンパイラが文字列リテラル (に見えるモノ) をパーズして、フォーマット文字列に対応する静的型情報を付加してくれるでゲソ。

"%d\n" の場合は (int -> 'a, 'b, 'c, 'd, 'd, 'a) format6 という型、

"%d %s\n" の場合は (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 という型が割り振られているでゲソが、この 1 番目の部分がそれぞれ

int -> 'a

int -> string -> 'a

となっていて、これはフォーマット文字列の内容に対応した型を引数に取って、'a 型の結果を返す関数の型になっているでゲソ。そしてこの 'a 型は format6 型の 6 番目の最後の部分にも入っているでゲソ[*2]。

コンパイラチートでゲソ! コンパイラが format6 型を特別扱いして、文字列リテラルから素敵な型を付けてくれるでゲソ。

20.2 どうやって format6 型になるでゲソ?

じゃあ、どうやってコンパイラは文字列定数を string 型か format6 型か判別しているのでゲソ? 実は私も詳しくは知らないのでゲソが、型アノテーションなどで format6 が要求されている場合は format6 型として、それ以外の場合は string 型としてコンパイルされるようでゲソ。OCaml では基本的に式の値にどんな型を要求するかで (パラメータ多相の型推論が埋まる以外に) 型や動作が変わることは基本的にはないので、これは異例な気がするでゲソ。

型アノテーションでなくても、直接 format6 型を要求する関数の引数に渡すと format6 型として処理されるでゲソ。

```
# (format_of_string "%d\n");;
  (* format6 型の引数を取る関数に入れても同様に format6 型として処理されるでゲソ
     format_of_string は
     ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6
     という型の関数で、実質的には恒等関数でゲソ *)
- : (int -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = <abstr>
```

Printf.printf も第一引数に format6 型を取るので、フォーマット文字列は format6 型として処理されるでゲソ。

```
# Printf.printf;;
- : ('a, out_channel, unit) format -> 'a = <fun>
```

format 型の実体は format6 型で、('a, out_channel, unit) format は ('a, out_channel, unit, unit, unit, unit) format6 にあたるでゲソ。

余談でゲソが、文字列リテラルに適用する直接の関数の引数の型が format6 になっていないと string として処理されてしまうようでゲソ。

[*2] 最新版 (OCaml 4.00) のマニュアル (参考文献欄参照) には型 ('a, 'b, 'c, 'd, 'e, 'f) format6 について一通り説明が書いてあるでゲソ。

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# format_of_string (id "%d\n");; (* 型エラー! *)
Error: This expression has type string but an expression was expected of type
         ('a, 'b, 'c, 'd, 'e, 'f) format6
```

format_of_string の引数の型から逆算すると、(id "%d\n") の型は format6、よって "%d\n" の型も format6、となる気がするでゲソが、OCaml はそういう型推論をしないでゲソ。ここでは id "%d\n" の部分で、id の型が 'a -> 'a (つまり引数の型が format6 でない) ので、"%d\n" の型が string として処理され、一番最後で format_of_string の適用時に上のような型が出るでゲソ。イカのように、同じようなコードでも、文字列リテラルが適用される関数の引数の型を直接 format6 になるように小細工をすれば通るでゲソ。まぁこのあたりで小細工する人はあまりいないと思うでゲソが…。

```
# let id2 = fun x -> format_of_string (id x);;
val id2 :
  ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6 =
  <fun>
# id2 "%d\n";; (* こうすると format6 型として出てくるでゲソ! *)
- : (int -> '_a, '_b, '_c, '_d, '_d, '_a) format6 = <abstr>
```

話がちょっとそれたでゲソが、こんな感じでコンパイラチート支援によってフォーマット文字列は format6 型として処理されるので、普通の string をフォーマット文字列に指定しようとしてもダメでゲソ。

```
# Printf.printf ("Hello, " ^ "%s\n") "world";;
Error: This expression has type string but an expression was expected of type
         ('a -> 'b, out_channel, unit) format =
           ('a -> 'b, out_channel, unit, unit, unit, unit) format6
```

("Hello, " ^ "%s\n") は普通の string 型の値なので、型エラーが起こるでゲソ。

20.3 フォーマット文字列をつながなイカ?

OCaml では、format6 型のフォーマット文字列は^^演算子で (文字列の^演算子のように) 連結することができるでゲソ。これを使えば、フォーマット文字列を連結することができるでゲソ。

```
# Printf.printf ("Hello, " ^^ "%s\n") "world";;
Hello, world
- : unit = ()
```

条件分岐とかで動的に変化もできるでゲソ。

```
# let print_name f name =
    Printf.printf
      (
        (if f then
           format_of_string "Hi"
         else
           format_of_string "Hello"
        )
        ^^
        ", %s\n"
      )
```

```
      name
  ;;
val print_name : bool -> string -> unit = <fun>
# print_name true  "Makoto";;
Hi, Makoto
- : unit = ()
# print_name false "Inari";;
Hello, Inari
- : unit = ()
```

format_of_string を入れているのは、(if f then "Hi" else "Hello") のようにすると先に説明したようにこれらの文字列リテラルが string 型になってしまうからでゲソ。format_of_string (if true then "Hi" else "Hello") とかしても無駄でゲソ (型エラーになるでゲソ)。

ただ、当然ながら引数の型は動的に変更できないでゲソ。そもそも静的に printf の引数型チェックができるようにこういう風になっているので当然でゲソ。

ちなみに、この ^^ 演算子は

```
val (^^) : ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
           ('f, 'b, 'c, 'e, 'g, 'h) format6 ->
           ('a, 'b, 'c, 'd, 'g, 'h) format6
```

という型になっているでゲソ。これは、イカのように型推論がされることによって、型の上でも正しく合成されるようになっているでゲソ。

```
# format_of_string "%d";;
- : (int -> '_a1, '_b1, '_c1, '_d1, '_d1, '_a1) format6 = <abstr>
# format_of_string "%s";;
- : (string -> '_a2, '_b2, '_c2, '_d2, '_d2, '_a2) format6 = <abstr>
# format_of_string "%d" ^^ format_of_string "%s";;
- : (int -> string -> 'a, 'b, 'c, 'd, 'd, 'a) format6 = <abstr>
```

というのを例にすると (説明のために型変数に 1 とか 2 とかをつけたでゲソ)、format_of_string "%d" ^^ format_of_string "%s" における ^^ 演算子の型の 'a、'f、'h を format_of_string "%d"、format_of_string "%s" の型に当てはめると (今回は format6 の 6 つの型パラメータのうち 1 番目と 6 番目だけ関係あるのでそこだけ見るでゲソ)、

```
'a = int -> '_a1
'f = '_a1
'f = string -> '_a2
'h = '_a2
```

となり、'f について単一化すると '_a1 が string -> '_a2 に置き換わり

```
'a = int -> (string -> '_a2)
'h = '_a2
```

となって、結果 ^^ 演算子の結果の型が

```
(int -> string -> '_a2, 'b, 'c, 'd, 'g, '_a2) format6
```

となってめでたしでゲソ!

20.4　参考文献

- OCaml の Pervasives モジュールのマニュアル
  日本語版 http://ocaml.jp/refman/libref/Pervasives.html 書式指定文字列の操作
  英語版 http://caml.inria.fr/pub/docs/manual-ocaml-4.00/libref/Pervasives.html Operations on format strings
- camlspotter さんの記事「OCaml 標準ライブラリ探訪 #3.0: Printf: 便利だけどいろいろ謎のある奴」
  http://d.hatena.ne.jp/camlspotter/20091102/1257099984

第21章

Haskellでもprintfじゃなイカ?!

— @nushio

コンパイラがサポートする特殊なビルトイン型を使うとかあもりにもひきょう過ぎるでしょう？ 汚いなさすがニンジャ汚い！

「—などと思ってしまったヘッズはケジメでゲソ！ 海のように広い心を持って、様々なアプローチを受け入れようじゃなイカ！ 前章ではOCamlの`printf`に触れたので、お次はHaskellの`printf`を調べなイカ？ 今回はかの高名な光の言語使いたるダークハスケルマスター、の契約者である高階(たかしな)さんに来てもらったでゲソ！」

「むろん、ギーメル・ヘー・コフ派の固有術式である原典創作の秘跡をもってすれば、コンパイル時に日常会話の中にすら任意の教義を見出し、望みの型の関数に変換することなど余裕。だが、Haskell 98の範囲内においても、型クラスを活用することで、一にして多貌なるもの、一つの真名に様々なarity、様々な型の引数の関数の化身を備わらせることは可能。[*1]」

「なんかすごい技術でゲソ！ 覚えておけばどこかで役に立ちそうじゃなイカ？ Haskellの`Text.Printf.printf`はイカのように、いくつでも引数を渡せるし、結果を文字列としても`IO`モナドとしても使うことが出来る仕様でゲソ。」

```
import Text.Printf

main = do
  -- まずは IO a として使うでゲソ！
  printf "one over seven = %6.5f\n" (1/7 :: Double)
  printf "%d %d\n" (1:: Int)  (3:: Int)

  -- 次に文字列として使ってみるでゲソ！
  let filename :: String
      filename = printf "%s-%d.txt" "script" (5::Int)
  writeFile filename "hello world!\n"
```

「そこかしこに型注釈しないと動かないのはちょっと残念でゲソが、便利じゃなイカ！」

[*1] GHC拡張のTemplateHaskellを使えば通常の文字列をパースして任意のHaskellソースを生成できる。THを使った型安全なprintfの実装については`http://okmij.org/ftp/typed-formatting/`を参照。またHaskell98の範囲内で、一つの名前が可変個の引数、様々な型の引数を取る関数を表すようにできる

21.1 printf を再現するでゲソ！

「Text.Printf の実装の詳細については闇原典をひもといてもらう[*2]こととし、今は本質に迫るために以下の単純化を施す」

- String しか返さない（モナド版は無視）
- フォーマット文字列は省く（単に引数たちをスペース区切りで表示）

```
-- File: printf1-1.hs
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}

printf :: PType r => r
printf = spr []

class PType t where
  spr :: [String] -> t

instance PType String where
  spr xs = unwords $ reverse xs

instance (Show a, PType r) => PType (a->r) where
  spr xs = (\x -> spr (show x:xs))

main :: IO ()
main = do
  putStrLn $ printf 0.1
  putStrLn $ printf 0.1 'U' 178

-- 実行結果
-- 0.1
-- 0.1 'U' 178
```

21.2 型を追うでゲソ！

「なんと printf が触手で数えられるほどの行数に帰着したじゃなイカ！ って、ちょっとまたなイカ?!いったいどうしてあのコードで printf が実現できているのか、さっぱりわからないでゲソ！」

「全ての真理は御言葉に含まれている[*3]」

「じゃあ説明してくれなイカ？」

「全ての真理は御言葉に含まれている[*4]」

[*2] Text.Printf のソースを読んでもらう
[*3] さっきのコードを読めばわかる (キリッ
[*4] さっきのコードを読めばわかるってば

「説明してくれなイカ？」

「全ての真理は御言葉にぅぅぁぅ・・・*5」

「六花も分かってなかったんじゃなイカ」

「あう」

「とりあえず片っ端から型を表示させてみなイカ？」

「任せて。爆ぜよリアル、弾けろワールド：無意識に潜む因果を名前空間に強制固定。単相拘束の契約に従い我、汝を召喚する・・・*6」

-- File: printf1.hs **に追記**

```
printf0 = printf
printf1 = printf
printf2 = printf
printf3 = printf

main :: IO ()
main = do
  putStrLn $ printf0
  putStrLn $ printf1 178
  putStrLn $ printf2 'U' 178
  putStrLn $ printf3 0.1 'U' 178
```

ギーメル・ヘー・コフ　セプトゥアギンタ、起動。

バニッシュメント・ディス・ワールド！ *7

```
$ ghci printf1.hs
... 演出中略 ...
Ok, modules loaded: Main.
*Main> :t printf0
printf0 :: String
*Main> :t printf1
printf1 :: Integer -> String
*Main> :t printf2
printf2 :: Char -> Integer -> String
```

*5 ごめんなさいわかりません

*6 多相な型を持つ 'printf'の利用ケースの各々に別個の名前を与えることで、MonomorphismConstraint を利用し、暗黙に生成されている単相型にアクセスできるようにする

*7 ghci を起動している

```
*Main> :t printf3
printf3 :: Double -> Char -> Integer -> String
*Main> :t printf3 1.78
printf3 1.78 :: Char -> Integer -> String
*Main> :t printf3 0.1
printf3 0.1 :: Char -> Integer -> String
*Main> :t printf3 0.1 'U'
printf3 0.1 'U' :: Integer -> String
*Main> :t printf3 0.1 'U' 178
printf3 0.1 'U' 178 :: String
```

「ゲ〜ソ〜。printf の型は PType r => r なんだから、どうやら、イカの型が PType のインスタンスになっているようでゲソ。でも、なぜさっきのソースから、こんなに沢山のインスタンスが生成されたのかわからないでゲソ。」

- printf :: String
- printf :: Integer -> String
- printf :: Char -> Integer -> String
- printf :: Double -> Char -> Integer -> String

「問題ない。この右目に宿りし邪王真眼は全ての真実、型、類をも見抜く。さらに存在の真名を視てその全ての化身を見破ること、応身が出現するまでの無始無数の輪廻を見徹すことも可能。さらに、一画面内に収まるよう必要に応じて型名や変数名を自動調整。所有者が視ている真景（ヴィジョン）を投映することができる[*8]」

「そんな能力、どうやって手に入れたでゲソ？」

「東方の三賢者に授かった [*9]」

「プロジェクター搭載カラーコンタクトを作ってしまうとは、いつもながら M.I.T. 恐るべしでゲソ」

邪王真眼察！ [*10]

「おー、ほんとに壁に図 21.1 と 21.2 が映し出されたでゲソ！ では解説してくれなイカ？」

「眩しすぎて自分ではなにも見えない・・・」

「肝心のところが抜けているのもいつものことでゲソね・・・しかたがない、私が解読してみせようじゃなイカ！

[*8] ghci を使うとシンボルの型や類を調べたり、インスタンス宣言の詳細を調べたり、簡約過程を追跡したりできる
[*9] 三バカにもらった
[*10] パワーポイント

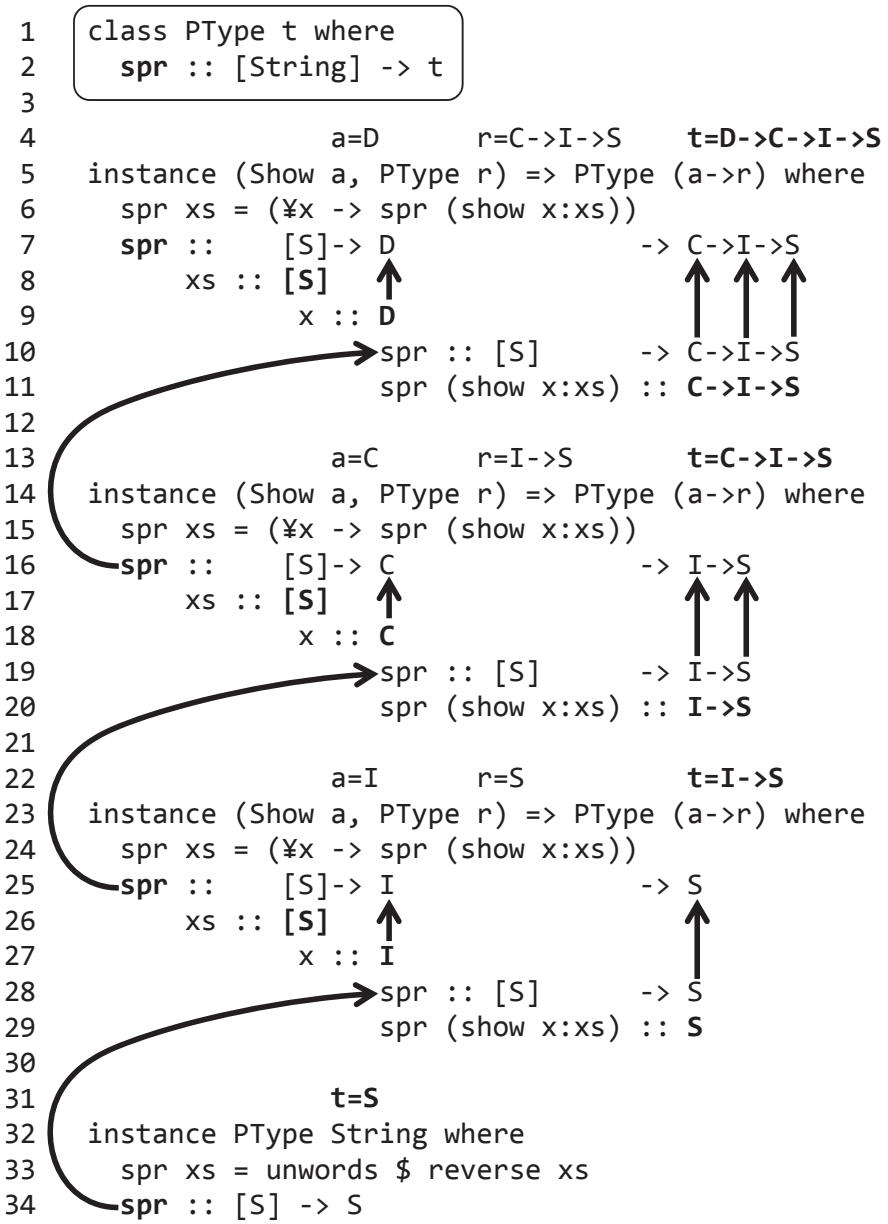


図 21.1: `Double -> Char -> Integer -> String` 型は、このような経路で `PType` インスタンスになっているでゲソ！

```
printf   0.1  'U'  178
printf :: D -> C -> I -> S
    |
    | printf = spr []
    ↓
spr    []    0.1  'U'  178
spr :: [S] -> D -> C -> I -> S
    |
    | spr xs = (¥x -> spr (show x:xs))
    ↓
(¥x -> spr (show x:[])) 0.1 'U' 178
  ↓ β 簡約
spr   ["0.1"]  'U'  178
spr :: [S]   -> C -> I -> S
    |
    | spr xs = (¥x -> spr (show x:xs))
    ↓
(¥x -> spr (show x:["0.1"])) 'U' 178
  ↓ β 簡約
spr   ["'U'", "0.1"]  178
spr :: [S]          -> I -> S
    |
    | spr xs = (¥x -> spr (show x:xs))
    ↓
(¥x -> spr (show x:["'U'", "0.1"])) 178
  ↓ β 簡約
spr   ["178", "'U'", "0.1"]
spr :: [S]                -> S
    |
    | spr xs = unwords $ reverse xs
    ↓
unwords $ reverse ["178", "'U'", "0.1"]
  ↓ 関数適用
"0.1 'U' 178"
```

図 21.2: `printf 0.1 'U' 178` が文字列に評価されるまでの流れでゲソ！

21.3　可変個引数関数の正体でゲソ！

「まず、図 21.1 からでゲソ。printf1.hs の putStrLn $ printf 0.1 'U' 178 という行の字面から、printf の型は Double -> Char -> Integer -> String と推論され、なおかつこの型が PType のインスタンスである必要があるでゲソ。

「PType のインスタンス宣言は instance PType String where と instance (Show a, PType r) => PType (a->r) where の二つでゲソが、Double -> Char -> Integer -> String があてはまるのは後者だけでゲソ。よってこちらが選択されるでゲソ (図 21.1、5 行目)。このとき、型変数 a は Double、r は Char -> Integer -> String になるじゃなイカ (4 行目)。そして、このインスタンス宣言では、[String] -> Double -> Char -> Integer -> String 型の spr を定義してあるでゲソ。

「ところが、6 行目の右辺にも spr があるじゃなイカ？ 右辺の spr の型は [String] -> Char -> Integer -> String でゲソ。左辺の spr と比べて引数から Double が減っているじゃなイカ！

- 型 (a->r) が PType のインスタンスになるには
- 型 r が PType のインスタンスであることが必要

でゲソ。そのとき、

- 型 (a->r) の spr メソッドは、
- 型 r の spr メソッドを利用して定義されている

のでゲソ。これはまさに再帰じゃなイカ？

- 6 行目の spr は 15 行目の spr を用いて定義され、
- 15 行目の spr は 24 行目の spr を用いて定義され、
- 24 行目の spr は 33 行目の spr を用いて定義されている

これらが再帰ケースであり、

- 33 行目の spr は専用のインスタンス宣言で直接定義されている

これが基底ケースでゲソ。

「こんどは、図 21.2 をみなイカ？ これは printf 0.1 'U' 178 が評価されるようすでゲソ！」

「私、気になります！」

「printf = spr [] という定義は、空のアキュムレータ [] を用意して spr 関数の再帰を起動していると思えばいいんじゃなイカ？ 次に、spr 関数の連鎖が、printf の引数を一個づつ食い潰してアキュムレータに入れていくでゲソ。基底ケースにたどり着いたら、リストを逆順に戻して、スペース連結して帰しているでゲソ。」

「わかってみれば実に簡単じゃなイカ！ この技を使えば、ごく普通の Haskell コードとして、可変個の引数を取る関数を書けるのでゲソ。今回見たように、polymorphism との組み合わせも意のままでゲソ。今回は Show を仮定することで、アキュムレータは単相にしたでゲソが、もし可変個引数のそれぞれの型を活かした処理を書きたければ、heterogeneous リストの出番でゲソ！ それはまた、将来の話としようじゃなイカ！」

21.4 まとめでゲソ！

再帰インスタンス宣言！！

「`PType r => PType (a->r)` のインスタンス宣言を行うことで何引数関数であろうとインスタンスにしてしまう禁断技。通常の関数ではとても受けきれないような、無数の異なる型の引数を、内部メソッド `spr` 経由で第一引数のアキュムレータに集積させ、基底ケースで一気に開放させ放つ[*11]」

「相手は死ぬ」

「ほほう、人間の娘にしてはデキるようだな」

「私はイカ娘でゲソ」

[*11] 再帰インスタンス宣言とは、`PType r => PType (a->r)` のようなタイプのインスタンス宣言を行うことで何引数関数であろうとインスタンスにしてしまう禁断技であり、通常の関数ではとても受けきれないような、無数の異なる型の引数を、内部メソッド spr 経由で第一引数のアキュムレータに集積させ、基底ケースで一気に開放させ放つ必殺技である。使用者の脳の型推論野には多大な負担を強いるものであることは言うまでもなく、多用すればもちろん寿命が縮む。再帰インスタンス宣言を使用中にウカツにも型エラーを発生させようものなら、膨大なカタ・エラー・ナンデ信号が流れ込み脳が焼き切れて廃人となってしまう諸刃の剣なのである。

第22章

けいさん！　highschool

— @tanakh

～これまでのあらすじ
大好きな唯たちが卒業して一人残されてしまったあずにゃん...。けいさん部の部室は今日も一人ぼっち...可愛そうなあずにゃん。でも、そんな落ち込んでいるあずにゃんの親友である憂ともう一人の子は新けいさん部として立ち上がるべく仲間に！ その後も偽ムギちゃんやメガネも加わりいよいよ情報大航海へ！？ 一大スペクタクルが今、開幕！

http://www.mangaoh.co.jp/catalog/295626/ より

先輩たちが卒業してから、しばらくが経ちました。元々「けいさん部」もとい、計算機プログラミング部なんてところに入るとは、入学するまでは夢にも思っていなかったのだけれども、あれから二年も経つ今となっては、もうそこにいない私は考えられなくなっていて———。私は今もここにいる。でも、唯先輩たちは、もうここにいない。少し前まではここにいたはずの人がここにいない、それがこんなに寂しいものだなんて、考えていなかったんです。いや、そうじゃなくて。考えたくなかったから、考えないようにしていただけだったんだ。

「———二人きりになっちゃったね」

水槽をゆらゆらとたゆたうトンちゃんにひとりごちます。先輩たちが卒業して、下級生の部員もいなくて、それなりに広い部屋に、私はひとりぼっち。それを考えたくなくて、無意識にカメに話しかけてしまう。去年の今頃、私が二年に進級するとき。けいさん部に新入部員が入らなくて、代わりに新メンバーとしてスッポンモドキのトンちゃんを迎えたんだった。

「———大きく—なったね」

飼い始めた後でわかったことだけど、スッポンモドキは成長するとかなり大きくなるといいます。こんな小さな水槽ではじきに飼えなくなる。そうなったら、今度こそ本当にひとりぼっちになってしまう。

「ひとりに———しないで———」

私以外に誰もいないただひとりの空間にて、緊張を保つ理由はありませんでした。頬を伝う涙をそのままに、春の陽光射し込む音楽室、私の意識はぼんやりとしていました。何気なしに、幾度となく繰り返した、ついこの間までの楽しい思い出を蘇らせます。「あ、あずにゃん。こんなところにいたんだ。探したんだよー！」やめてください、唯先輩。頬を擦り付けないでください。ここ、人前ですよ。「だってあずにゃん可愛いんだもん。すりすり」やめてください、本当に怒りますよ。———やめないで！

体をピクリとさせながら微睡みから覚めると、どれぐらいの時間が過ぎたのか、誰かが部室の扉をノックする音の後、しばらくしてカチャリと扉が開きました。

「すいませーん、けいさん部の部室って———」

「———けいさん部の部室って、こちらでしょうか？」

ああ、開けてしまった。ついに開けてしまった。部活紹介のライブがかっこよくて、気になっていたのだ

けど、いざ見学となると腰が引けてしまう。扉の前を何度も通り過ぎて、入るか入るまいかなかなか踏ん切りがつかなくて、うろうろ往復することどれ程の時間が経ったのだろうか。ここが校舎の外れでよかった。

「あ、澪ちゃん、来ちゃったよ。新入生が来ちゃった。あわわわ」

「唯が緊張してどうするんだ、ここはビシッとカッコいいところを見せてだな」

「だ、だって、私どうしていいか」

「あ、あの———」

あれ、間違えたのかな。ついさっきの、堂々とした演目での様子とは随分違う———。でも、たしかに同じ人たちだったような。

「ようこそ、けいさん部へ！」

「ようこそ！！」

今時珍しく、おデコを露出した人と、頬を膨らませてマンボウのような顔をした人が、ぱ、ぱーん！ と、左右からクラッカーを鳴らしました。

「わ、これは一体———？」

突然の歓迎に、幾分寿命が縮まった気がしました。でも、ここは確かにけいさん部の部室のようだと分かって少しほっとしたりもしたので、寿命はプラスマイナス 0 といったところでしょうか。

「入部希望の人ですね」

「は、はいそうですけど———」

「じゃあこっちに、どうぞ、座って、座って。いまムギちゃんがお茶用意してるからね」

「お菓子も食べていいぞ」

何がなんだかわからないうちに、椅子に座らされて、放課後のティータイムとなりました。横では、さっきマンボウのような顔をしていた先輩が、紅茶を淹れてくれています。なんだか思っていたのとは違うなあと思いながら、その紅茶の良い香りには逆らえずに、

「じゃ、じゃあ。頂きます」

あ、美味しいな、これ。

「ムギ、今日はどこの茶葉なんだ？」

「イギリスの老舗、マックウッズの 160 周年記念ブレンドよ。40g で 1 万円ちょっと———」

「ブッーーー」

「あっ、大丈夫ですか？ 熱かったかしら」

な、な、なんでこんな高価な紅茶が出てくるの？

「いえ、ちょっと値段に驚いたもので———」

「あー、そうだよなー。私も最初の頃は」

「すっかり日常になってたけど、改めて考えたら高いな」

「せ、先輩方はいつもこのようなものを…？」

「———う、う、うーん。そうかな。そんな感じかな。うん」

え、はぐらかされた？ 聞いちゃいけないのかな。それにしてもこの部活は一体———。

「えー、ではそろそろ本題に入らせてもらおうか」

「おお、りっちゃんが珍しく部長らしいことを！」

「珍しく、は余計じゃい」

「ご、ごめんよ〜」

「おい唯、話が進まんぞ」

おデコの人が部長のようです。それにしても広いおデコです。いかにも脳ミソが詰まっていそうです。そのおデコに反して、言動はいかにも賢そうじゃああありませんが、カムフラージュかもしれないので、油断はできません。

「新入生君、君の名前は？」

「な、中野梓です」

「なかのあずさ———あずさ———あずにゃん！」

「あ、あずにゃん？」

「あずにゃんー！ あずにゃんでいいよね。だって、とっても猫耳が似合いそうな顔してるんだもん」

「えっ、えっ」

そういって、後ろ手から取り出した黒い猫耳を私の頭にかぶせようとしてきます。
「―――やめてください～！　う、うわあ、恥ずかしいです」
「ええじゃないか、ええじゃないか。ぐへへぇ、…えい！」
「や、やぁ……」
「おお、やっぱり！」
は、恥ずかしい。恥ずかしくて死にそうです。それに、猫耳が似合いそうな顔って、どういうことなんでしょうか。すごく、失礼なことを言われた気がします。
「やっぱり思った通り。とっても猫耳が似合う！」
「やめてください～。み、見ないで」
「だって可愛いんだも～ん」
「おい、あんまり新入生いじめたら、逃げちゃうだろ」
「あ、ごめんね、あずにゃん。悪気はないんだよ。悪気はないから、猫耳外す前に一回『にゃあ』って―――」
「おい唯」
もう、帰りたい―――。
「ごめんね、梓ちゃん。唯ちゃんは悪い子じゃないんだけど、小さくて、可愛くて、黒髪で、ツインテールで、真面目で、猫耳が似合いそうな女の子を見ると、見さかいがなくなっちゃうみたいで」
「ずいぶんピンポイントだな、おい」
「というのは、冗談で―――」
ムギ先輩が、真剣な顔を近づけて、
「唯ちゃんは、梓ちゃんになにか感じるところがあったんじゃないかしら？」
この人、真剣な顔するとマンボウそっくりになるんだなあ―――。
「梓ちゃん、話聞いてる？」
「え、あ…はい」
「あずにゃん、なんか失礼なこと考えてたでしょ？」
「ぎ、ぎくり。そんなことありませんよ！」
「おい唯」
「えー、ウォホン！」
律先輩が、わざとらしく咳払いをしました。この部の部長は大変そうだなと思いました。
「さて、新入部員の中野梓くん」
「まだ入部すると決めたわけでは―――」
「じゃあ、そういうことにしておこうか。新入部員（仮）の梓くん。君は、我が部、けいさん部もとい、『計算機プログラミング部』が何をする部なのか、知っていて見学に来たのかな？」
「計算機のプログラミング、をする部活でしょうか―――？」
「その通り！　だが、それは部の名前を見れば誰にでもわかること。私が聞いたのは、計算機のプログラミングとは具体的に何をするものなのか」
「すみません。実際にやってみた経験はなくて―――。部活紹介での先輩方がかっこ良くて、私もやってみたいなあって」
「謝ることはないぞ、新入部員（仮）くん！　でも、形から入るのもあまり感心しないな、新入部員（仮）くん！」
「ふ、ふぇぇ」
「あそこまでできるようになるには、相当な鍛錬を積む必要がある。君に、その覚悟があるのかな？」
やっぱり、ふざけているように見えて、相当な努力の賜物だったのだなあ。能ある鷹は爪を隠すというものを目の当たりにしているんだ。
「そうだよ、あずにゃん。うちの部は厳しいよぉ。毎日部室でたくさん―――」
「お菓子を食べているな。唯は」
「ひ、ヒドいよぉ。澪ちゃん」
さっきちょっと感心したのを返して欲しい。
「ま、まあ、実際大変だぞ。体重管理とか」

「えへへぇ、私はいくら食べても太らないんだよぉ」
「おい唯」
ずるい。
「ふむふむ、未経験。未経験者は大いに歓迎だぞ。もともとこの部は、去年部員が全員卒業して廃部寸前だったところを、私が無茶を言って、全くの素人で頭数を揃えてでっち上げたようなものだからなあ」
「そうそう、私なんか、未だに『C のコード』が覚えられなくて―――」
「それでも動くもの作ってるお前はおかしい」
「えへへぇ」
「だから、梓、お前も心配する必要はないぞ。必要だったら、私が教えてやる」
「私も教えて上げるよぉ。『F#』がオサえられるようになったら、一人前だって」
「おい唯」
澪先輩、真面目で優しそうで、いい人そうだな。それと引き替え、唯先輩―――。
「さて、新入部員（仮）君、プログラミング未経験とはいえ、現代のこのコンピュータ時代だ。コンピュータに触れる機会や、その裏に横たわる理論について触れる機会、少なからずあったんじゃないかな？」
「―――うーん、コンピュータですか」
なんだろう。私だって、日々漫然と生きてきたわけではない。そういえば、この前読んだブルーバックスに載ってたミレニアム懸賞問題、これにコンピュータ関連のがあったような―――。
「―――え、NP 問題」
「お、NP を知っているか」
「あまり良く理解できませんでしたけども」
「P=NP 問題は、まさに私たちの活動の根幹に関わる問題なんだよ」
「えっ、先輩方はまさか、これを証明しようと！？」
「まさか、そんなわけあるかーい！」
律先輩は、ちゃぶ台返しのようなポーズで両手を上に上げた。
「それを本気でやるには、人生を捨てないといけないぞ。仮に、万に一つもないが、解けたとしても、何十年も毎日その問題のことばかり考えるんだ。その頃には世捨て人になっている覚悟が必要だ。余生はキノコ狩りだぞ」
「キノコ狩り楽しそうだね～」
「おい唯」
「私たちは、NP 完全問題に属する問題の一つ、充足可能性（Satisfiability）問題、通称『SAT』を主に扱っているのよ」
SAT―――?どこかで聞いたような、聞いてないような。
「特殊急襲部隊、通称 SAT（Special Assault Team）ではないぞ。かっこいいけどな！」
「あ、なるほど。地味な方の SAT なんですね」
ふむふむ、勉強になるなあ。
「地味―――だと？」
「地味―――ですって？」
「地味―――なのかな？」
「おい梓」
「あわわわ、私なにかまずいことを」
しまった、つい思ったことを口に。
「SAT の力を甘く見てはいけないぞ。SAT が解かれれば、それが利用される場面は極めて多いのだよ。あらゆる最適化問題から、プログラムの信頼性に関わる問題まで、世界のパワーバランスが変わるといっても言いすぎじゃない。今や SAT は計算量理論のためのおもちゃじゃないんだ」
もしかして、先輩方、高校二年にもなって、中二病を患っていらっしゃるのかな―――？
「あずにゃん、今ちょっと中二病っぽいって思ったでしょ」
「ぎ、ギクリ。唯先輩はエスパーですか」
「ふふふ、あずにゃんの考えはお見通しなのだよ」
「でも、先輩。P=NP 問題には手を出さないってさっき」

「———チッチッチ」

律先輩は片目を閉じて、人差し指をふりふり。いかにも勿体つけながら、

「SAT を解くのと、SAT が P であることは違うのだよ。もっと言うと、P に属する問題が簡単で、NP に属する問題が難しいというのも違うぞ。例えば、10 年ほど前に、整数の素数判定問題が P に属すると証明されたけど、これは 6 次のアルゴリズムで、とてもじゃないが大きな問題を解くことはできない。その逆で、NP に属する問題でも、x^n の x がとても小さくて、問題のサイズが数万、数十万、数百万でも解けてしまうものもある。理論的な興味を抜きにすれば、現実に転がっている問題、それがどれぐらい解けるかもこれまた重要な問題なんだ」

なるほど———。わからん。

「というわけで、新入部員（仮）くんへの宿題。明日までに、SAT についてのすべてを調べてくること！」

「そ、そんな無茶なぁ」

「がんばって、あず SAT！」

えっ？

「SAT に入門したから、今日からあずにゃんはあず SAT だね！」

そもそも「あずにゃん」も今日唯先輩がつけたあだ名だった気もしましたが、このときの私には正直どちらでも良かったのです。変な人ばかりの先輩たちだったけれども、私は少しずつ SAT の魔力に魅入られ始めていたのです。

「あ、あずにゃん猫耳」

「にゃ、にゃあ？」

すっかり外すのを忘れていた猫耳を外し、部室をあとにしました。

「結局、新入部員はあずにゃん一人だったね」

5 月の連休も明けて、高校生活にも慣れてきました。けいさん部は、放課後の部室で優雅にお茶を飲んでいます。それで、通称『放課後ティータイム』。入部したのを少し後悔し始めていました。

「それにしても、梓は、なんでけいさん部にはいろうと思ったんだ？」

「お前がそれを言うのか、律」

「まあなー。傍から見たらうちの部、遊んでいるようにしか見えないんじゃないか？」

自覚はあったんだ、と出かかった言葉を飲み込みながら、

「そうでした！ ライブ！」

すっかり忘れていたけど、こう見えても、先輩たちはすごい人なんだ。それは部屋のあちこちに散らかされている表彰状やら、トロフィーやらから、間接的にうかがい知ることはできる。でも、直接的にそれを見る機会は、未だ部活紹介の時以来やって来ていなくて。

「うんうん、そうなんだ……。って、あれ、澪ちゃん。ライブってなんだろう？」

「うーん、そんなのやった覚えないな。軽音部じゃあるまいに」

「あっ、でもバンド組むのも面白そうですね。私も、キーボードならなんとか扱えます！」

「そっちのキーボードじゃねえよ。それに全員キーボード担当になっちゃうじゃねえか」

マンボウみたいな顔をしながら空中でタイピングの真似をしているムギ先輩を私は悲しそうな顔をして見つめていたことでしょう。

「ライブコーディングです！」

「あ、ああ。そっちか！」

「ライブコーディングのことライブって略すなー」

「ウィキペディアのことウィキって言うなー」

「澪ちゃん、ウィキペディアってなに？」

本当に大丈夫なのかな。この部———。

「———今、本当にこの部にいて、大丈夫なのかな、と思ったね？」

「うわああ、唯先輩。エスパーですか！」

「近からずも遠からずや。これも SAT のちょっとした応用なんだよ。そんなあず SAT には、SAT の本当の恐ろしさを教えてしんぜよう」

「唐突ですね、先輩」

うむうむ、としかめっ面で頷く唯先輩がいつもとは違いなんだか面白くて、吹き出しそうになってしまいました。が、我慢しました。ついに、SAT について教えてもらえるのかと、少し緊張してきました。

「SAT については、知ってるよね？」

「はい、入部した時に聞かせてもらいましたけど———」

あれ以降はお茶会しかなかったから———というのは、ググっと飲み込んでおきました。

「大体は知っていると思うけど、改めて SAT、つまり充足可能性について説明しておくね。SAT ってのは、ある命題論理式が与えられた時に、それが充足可能かどうかを判定する問題だね。例えば———」

落書きだらけのホワイトボードに、唯先輩はそれ気にすることなく式を書き始めます。

- $(x_1 \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$
- $(x_1 \vee x_2) \wedge (\bar{x_1} \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$

「さて、あずにゃん、ここに 2 つの式があるじゃろう？ これらが充足可能かわかるかな？」

「それぞれの変数には、true か false のどちらかが入るんですよね。うーん、そうですね———」

1 つ目の式は、変数が x_1 と x_2 の 2 つだけ。x_1 を真（`true`）にすれば、1 つ目と 2 つ目の節が充足される。3 つ目の節は、x_2 を偽（`false`）にすれば、充足される。

2 つ目の式は、これも変数が x_1 と x_2 の 2 つだけだなあ。それに対して、節が 4 つで、そこに含まれる真偽の組み合わせの全パターン、これをすべて充足する組み合わせはない———。

「———1 つ目の式は充足可能、2 つ目の式は充足不可能ですね」

「———ファイナルアンサー？」

「え…？」

「ファイナルアンサー？」

「ふぁ、ファイナルアンサー」

真面目なのかふざけてるのかよくわからないな、この人は。

「———正解！！」

しかもネタが古いです。

「いやあ、ごめんごめん。一度やってみたかったんだよ、これ」

「もう、真面目にやってください！」

「———いいのかな？」

「何がです？」

「私が本気を出したら、どうなっても知らないよ？」

「大丈夫です、真面目にやってください」

「ちぇー、つまらないなあ」

頬をふくらませて、いじけて見せます。

「じゃあ、次。標準型の話ね。論理式には、積和標準形と呼ばれるものと、和積標準形と呼ばれるものがあってね、それぞれ、論理積のみからなる節の論理和、論理和のみからなる節の論理積、で———って口で言ってもわかりづらいね」

先輩は再びホワイトボードを走らせ、

- $(x_1 \wedge x_2) \vee (x_1 \wedge \bar{x_2}) \vee (\bar{x_1} \wedge \bar{x_2})$

「これが、積和標準型で」

- $(x_1 \vee x_2) \wedge (\bar{x_1} \vee x_2) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee \bar{x_2})$

「こういうのが和積標準型」

ふむふむ、なるほど。

「さっきの問題の式は、和積標準型だったんですね」

「そうだね」

唯先輩は満足そうに頷き、

「標準型、というように、すべての論理式はこれらの標準型で表すことができるんだよ。あずにゃんは、

ド・モルガンの定理は知っているよね？」
「はい、数学で習った気がします。確か、」
今度は私が、ペンをとって、

- $\neg(P \vee Q) = \neg P \wedge \neg Q$
- $\neg(P \wedge Q) = \neg P \vee \neg Q$

「こういうやつですよね」
「そう、それだよ」
唯先輩は、またも満足そうです。
「これと、二重否定の削除とかを組み合わせて、ゴリゴリ変形していくこともできるけど、最悪の場合だと、式のサイズが指数的に膨れ上がってしまうこともあるね。和積標準型への変換に限れば、元の式の線形サイズに抑える方法もあるけど――」
「ふむふむ」
「ちょっとややこしいから、それは置いといて。標準型で表せるってのは、真理値表を考えたほうがわかりやすいかなあ？ 0/1 の表の、1 になってるところの論理和を取れば積和標準型になるし、逆に――」
「0 になってるところの否定の論理積を取れば、和積標準型になるんですね！」
「そのとおり！ なかなか賢いじゃあないか、あず SAT 君！」
いつもと違って、変な口調だけど、ほめられて悪い気はしない。でも、標準型は何が嬉しいのだろう？
「標準型、特に和積標準型が重要なのは、これの各節を 3 変数に制限したものの充足可能性問題が、NP 完全問題に属するというのが大きいかな。特に 3SAT と呼んで区別しているね」
「ふむふむ、たった 3 つでそんなに解くのが難しくなるんですか」
直感的には、3 と 4 では全然違うのになあ。
「1、2、3、たくさん、って感じだね！」
唯先輩みたいですね！ といいそうになりましたが、
「2SAT は、効率的に解けるんですか？」
「そうだね、説明すると長くなるけど、2SAT はとても効率良く解くアルゴリズムが知られているよ」
ふむふむ、それも気になるけど、もうひとつ気になることが、
「3SAT が NP 完全問題ということは、すべてのクラス NP に属する問題は、3SAT で表すことができるってことですよね。じゃあ当然、4SAT も 3SAT に変換できることになりますね」
「なかなか鋭いね。ちょうどいい練習だから、考えてみるといいよ」
「うーん――」
しばらく考えたのですが、わからなかったので、宿題にしました。
「3SAT を 2SAT には、流石に出来ませんよね」
「それができたら、フィールズ賞ものだよ、あず SAT 君！」
そうだった！ ミレニアム懸賞問題を解くことになってしまいます。
「ということで、和積標準型が重要だということがわかったかな？ 和積標準型、長いからここからは CNF と呼ぶね」
「CNF って、何の略ですか？」
「うーん――忘れた」
あとで調べたところ、Conjunctive Normal Form の略のようでした。
「あまり細かいことに囚われていては、全体が見えなくなるのだよ」
唯先輩は、なぜか誇らしげです。
「そういうことがあって、CNF は計算量理論の世界では古くからよく研究されて来たんだけど、最近になって、別の方面でも CNF はとても重要になってきたのだよ」
「といいますと？」
「CNF の充足可能性問題を解くプログラム、いわゆる SAT ソルバと言われるものの性能がここ 10 年ほどで恐ろしいほど向上したんだ。最新の高性能ソルバだと、ものにはよるけど――」
両手を大きく広げるような仕草で、
「数万～数十万変数の問題も解けるようになっているみたい」

「す、数十万？」

指数時間かかるアルゴリズムが、そんなに大きなサイズまで解けるなんて！ これじゃあ、二乗のアルゴリズムよりも大きなものが解けてしまいます。

「どうだ、驚いただろ？」

いつの間にか律先輩が横に来ていました。

「指数時間というのは、あくまで漸近的性能を示しているに過ぎないんだぞ」

「そうそう、私も律に教わるまでは、SAT がそんなに速く解けるとは思ってなかった」

澪先輩、律先輩に押されてけいさん部に入ったんだっけ。

「だけど、実際にそうなってみると、これはとんでもないことだなって———」

「あっ、あらゆる NP 問題は、3SAT に…！ まさか、P=NP なんていう必要がなかったとか？」

「『NP 完全』というのは、あくまで多節式時間還元できることしか言ってないから、SAT への効率のいいエンコードが常に存在するわけではないな。だけど、利用できる範囲が実際に狭いわけじゃない」

「なるほど———」

解きたい問題をエンコードしてみて、数十万変数に収まれば、解かれてしまうかもしれない。なんだかすごいことになってきました。

「でも、なんで最近になって急に性能が良くなったんでしょうか？」

良い質問だね、とでも言いたげに、唯先輩がもったいぶって話し始めます。

「21 世紀のはじめに、SAT ソルバにとって、大きなブレークスルーがあったんだよ。それまでは、DPLL ってアルゴリズムで、単一節、つまりひとつしか変数を含んでいない節の除去とか、二変数しか含まない節のうち、片方が確定したときの伝搬とか、わりとシンプルな枝借り探索しかやられていなかったんだけど」

それぐらいなら、私でも思いつくなあと思いました。

「ここで出てきた、CDCL というアルゴリズムがすごかったんだよ。Conflict-Driven Clause Learning っていうんだけどね。Conflict、つまり探索の途中で充足できない節が見つかった時、その原因を解析して、新しい節、つまり Clause を学習する」

「が、学習ですか———」

なぜかコンピュータが机に向かって勉強している姿を想像してしまいました。

「学習といっても、何かを教わって理解するとかそういうのではないね。機械学習的なものとは違って、Clause Learning は、発見といったほうが感覚的に近いかもしれないね」

「発見…ですか？」

「SAT の充足性の判定はある種の証明だと考えることができる。例えば、人間が何かを証明しようって時は、いきなりそれを証明するんじゃなくて、周辺の Lemma、つまり補助定理を積み上げて証明を組み上げていくものだけど、SAT の場合もその類推で考えるとわかりやすいよ。変数に仮定をおいて探索を進めていくときに、矛盾が起こった。SAT ソルバ君は、あれ？ なにがおかしかったんだろうって、必死で考えて、ああ、これがダメだったんだ、と理解する。じゃあ、これ以降の探索には、今獲得した矛盾の原因が起こらないように、元の節に追加するんだよ」

何か探索問題をエンコードして解かせた場合は、人が書いた場合の枝狩り条件に相当するものが、CNF の節として自動で獲得されるのかなあと思いました。

「Lemma を自動獲得すると表現する文献もあるぞ。どうだ、かっこいいだろう！」

そんなことができるなんて、賢すぎる。けど本当にそんなにうまくいくのかなあ？

「そんなに、うまくいくのかなあ？ と思っているんじゃないかな。あず SAT 君！」

「ひ、ひぇー。先輩———」

「そんなあず SAT 君には、ここの問題を全部 SAT にエンコードして、SAT ソルバに突っ込むという宿題をあげよう！」

そう言って、唯先輩は、なにか雑誌のようなものをドサッと置いて行きました。

「なんですかこれ———ナンプレ、カックロ、スリザーリンク、イラストロジック———こ、これ全部ですか！？」

「もちろん、今日は終わるまで帰さないよ、あずにゃん」

「そ、そんなあ」

と言いつつも、久々にけいさん部らしい活動ができて、少し嬉しい私なのでした。

6 月のジメジメした季節。結局あの時渡されたニコリのパズル雑誌は、経験の浅い私にはそんなにすぐに解けるはずもなく。それどころか、未だに半分以上残っていました。先輩たちが毎日お茶をしているのを横目に、私の気分まで、梅雨空のように曇るのでした。

「あーっ。めんどくさい！」

そう、めんどくさいのです。SAT ソルバは、その名の通り、命題論理式の充足可能問題しか解けません。変数は真か偽かの 1 ビットの情報しか載せられず、整数を扱うのにも、ビットを束ねて、加算器をつなげて、論理回路を一から書いているような状況です。さらに、その際にかなり工夫して書かないと、サイズが膨れ上がって、思うように大きいサイズの問題が解けなくなってしまいます。

「苦戦しているようだな」

見かねて澪先輩がやって来ました。

「はい、思うようにコードが書けなくて———」

「うーむ、なるほど———」

私のコードを覗いて、なにやら独り言を呟いて、

「———そろそろ、頃合いかな。梓、」

「は、はい！」

「お前に、SMT を教えよう」

え、えすえむ、ティー？ また変な名前の———。

「あずにゃん、そういうのじゃないよ」

「うわぁ、唯先輩。そういうの考えてないです！」

相変わらず、唯先輩には心を読まれているようです。

「スモール・ミディアム・トール、でもないわよ」

「ムギは飲み物の話ばかりだな」

SAT は、もう、たくさん。

「Satisfied Modulo Theories。簡単に言えば、SAT に別の理論を追加したものだ」

何かを追加する…？

「SAT は Boolean ロジックしか扱えない。だから、せっかく高性能になった SAT ソルバを利用しようと思っても、それを Boolean のコードにエンコードしなければいけない」

そう、それがめんどくさいんです。 SAT、めんどくさくて、たいへん。

「ただ、めんどうなだけならそこまでの問題ではないぞ。例えば、32 ビットの整数の足し算をするには、全加算器を 32 個つなげることになる。全加算器は、XOR を含む論理回路として表現できるが、XOR はそのままでは表せないから、これも CNF に変換すると、かなりサイズが大きくなってしまうんだ」

「足し算 1 つで、数百節になってしまうと、せっかく数十万規模の CNF を扱えても、実際に扱える整数問題のサイズは、かなり目減りしてしまうわね」

そう、それで私も困っていたところでした。

「まあ、中には CNF に加えて XOR 節を扱えるようにした、CryptoMiniSat みたいな処理系も存在するが、それは今回は置いておこう」

「CryptoMiniSat…暗号解析用のソルバなんですか？」

「いや、これは汎用の SAT ソルバで、特に暗号向けというわけではないぞ。暗号ルーチンに XOR が多く含まれるというのは無関係ではないだろうけど」

全加算器にも XOR は多く使われるから、整数の計算も高速化されそうだと思いました。

「多くの最新の SAT ソルバは、XOR のようなパターンは自動で認識して、特別な高速化が施されているから、今現在ではあまり考えることはないかな」

「へぇ、SAT ソルバってすごいんですね」

「まあ、それでも問題はあってだな。上限のない整数や、有理数、実数といったものは、そもそも SAT ソルバで扱うことは困難だ。加算のような処理も、予めどういう処理かわかっているから、かなり効率良く探索できるようになる。SAT ソルバも理想的にはそういうパターンを認識して自動で制約を獲得できてもおかしくはないが、現状ではそこまでうまくは動いてくれない」

理想と現実というのはどこにでもあるものだなあと思いました。

「で、SMT だ。SMT は SAT に Modulo Theory を追加したもの。具体的には――」

そこまで言うと、澪先輩はホワイトボードに向かって何やら書き始めました。

- Linear Integer/Real Arithmetic
- Non-linear Arithmetic
- Bit-Vector
- Uninterpreted Functions
- Arrays
- Quantifiers

「こういったものだ」

なにやら色々ありました。見覚えのあるものから、全く聞いたこともないものまで。

「1 つずつ説明していこう。Linear Integer Arithmetic は、整数に関する線形の演算だな。整数の足し算とか定数との掛け算なんかがサポートされる。これを使うと、$x+2=3$ のような制約式が書けるようになるぞ。Differential logic と組み合わせて、$x>3 \wedge x+x<9$ のような制約式を書くこともできる」

「私の今までの苦労は一体――」

「急がば回れというやつだよ、あずにゃん！」

「ううん、最初から教えてくださいよお」

「別に意地悪というわけではないぞ。SAT に慣れ親しむことは、SMT の理解にも大きく役立つんだ。だから最初のうちは SAT でだな――」

「とか言って、澪ちゃん SMT 知った時同じような反応してた」

「うるさい！」

澪先輩にもそういう時期があったんだなあ。

「それはともかく、Linear Arithmetic のサポートは表現力以外にも大きなものがある。線形の制約を解くには、線形計画法が使えるから、何らかの高速なアルゴリズムが実装されている処理系が多いぞ。今現在の最先端の処理系 “z3” では、Simplex 法が使われて、高速に、しかも解析解が求まるんだ」

そう言って、澪先輩は PC に向かって何か打ち込み始めた。

「簡単な線形整数充足問題、連立方程式でも解かせてみるかな。z3 には C++ の API や、Python のバインディングもあるが、私はもっぱら Haskell の sbv というパッケージを使っている」

ghci を起動して、手際よくコードが入力されていきます。

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ x + y .== 3
          , 2 * x + 5 * y .== 9
          ]
```

「よし、と」

ターン！ と勢いよくエンターキーがたたかれると、次のような出力が出てきました。

```
Satisfiable. Model:
  x = 2.0 :: SReal
  y = 1.0 :: SReal
```

「す、すごい」

わ、私の苦労はいったい何だったのかな。

「ここでは、SReal というのが変数の値だな。2.0 と出ているが、これは浮動小数ではなくて、厳密な値だ。

その証拠に、こんなふうにすると―――」

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ 3 * x + y .== 3
          , 2 * x + 5 * y .== 9
          ]

Satisfiable. Model:
  x = 6 % 13 :: SReal
  y = 21 % 13 :: SReal
```

「とまあ、こんな具合だ」
「ええ！ これどうなってるんですか？」
「解析解が求まるときは、それを求めてくれるんだ」
「解がない場合はどうなるんでしょう？」
「そりゃあ、充足問題なんだから、当然」

```
> sat $ do
    x <- sReal "x"
    y <- sReal "y"
    solve [ 3 * x + y .== 3
          , 3 * x + y .== 9
          ]
Unsatisfiable
```

「こうなる」
「ははあ、なるほど。解が無限にある場合は」
「無限個は、うまく扱えないかな。少なくとも充足解をひとつ見つけるか、無限に見つけることはできるが―――」
「無限個にあるということを示すことはできるぞ」
今度は律先輩がキーボードを奪い、なにやら書き始めました。

```
import Data.SBV

eq :: SReal -> SReal -> SBool
eq x y = bAnd
  [ 3 * x + y .== 3
  , 3 * x + y .== 3
  ]

hasInfSolution :: Predicate
hasInfSolution =
  forAll_ $ \x y -> forSome_ $ \x' y' ->
    (eq x y ==> eq x' y') &&& (x, y) .< (x', y')

main = print =<< prove hasInfSolution
```

「このプログラムは、ひとつの定理をしようとしている。eq って関数が、連立方程式だな。で、hasInfSolution ってのが、示したい性質だ。Haskell の DSL だから、細かい記法は気にせずに読んでくれ。すべての解に対して、それより小さい解が何かひとつは存在するって、要するにそういうことだな。最後に、それを証明（prove）して、結果を表示せよ！ と」

ぽちっとな！ とコードが実行されました。

```
$ runhaskell hoge.hs
Q.E.D.
```

「見事、Q.E.D. というわけだ。どうだ、かっこいいだろ？」

「す、すごい――――！」

何がどうなってるのかは、さっぱりわからないけど、すごい――――！

「どうなってるかは、わかってしまえば簡単だぞ。SMT も所詮は制約ソルバ。何か式を充足する変数があるかを調べることしかできない。でも、逆に言えば、証明問題をそういう問題に変換すればいいんだ。つまり何かを証明したいときは、それの反対を充足する解の不在を証明すればいい」

「ふむふむ、じゃあ、定理が間違っていたときは、もしかして」

「そう、反例（counter-example）が出てくるという寸法だ」

「至れり尽くせりとはこのことですね」

こんなことができてしまっていいんでしょうか。人類の脳が衰退しそうです。

「梓、まだ、これは序の口だぞ。z3 は、部分的だが、非線形実数問題に対する解析解を出すこともできる」

「え、私もうおなかいっぱい」

「簡単な例だと、二次方程式の解とかだな、ぽちぽち、と」

```
> sat $ \x -> x^2 - 3*x + 2 .== (0 :: SReal)
Satisfiable. Model:
  s0 = 2.0 :: SReal
```

「おお、出ました。でも、この方程式、もうひとつ解がありませんか？」

「そうだな。そういう時は、SMT ソルバの全解列挙機能を使えばいい。sat のところを allSat に変えて――――」

```
> allSat $ \x -> x^2 - 3*x + 2 .== (0 :: SReal)
Solution #1:
  s0 = 2.0 :: SReal
Solution #2:
  s0 = 1.0 :: SReal
Found 2 different solutions.
```

「これでよし」
「ちゃんと出ましたね。しかも、2 つの解があるって親切にも！」
「重解ももちろん出せるぞ」

```
> allSat $ \x -> x^2 - 2*x + 1 .== (0 :: SReal)
Solution #1:
  s0 = 1.0 :: SReal
This is the only solution.
```

「うへぁあ、すごい」
「これは実数の式だから、これをこうやると———」

```
> allSat $ \x -> x^2 .== (-1 :: SReal)
No solutions found.
```

「完璧！」
「なんと」
「さらに……！」

```
> allSat $ \x -> x^2 - 5*x + 3 .== (0 :: SReal)
Solution #1:
  s0 = root(1, x^2-5x = -3) = 0.6972243622680053... :: SReal
Solution #2:
  s0 = root(2, x^2-5x = -3) = 4.3027756377319946... :: SReal
Found 2 different solutions.
```

「これまでの解もやはり解析解だったと」
「———」
もはや圧倒されて言葉になりませんでした。
「すごいです！ これは！」
「じゃあ、Arithmetic はこれぐらいにして、次にいくぞ。BitVector は、Bit を束ねたもので固定長の整数に対する演算をサポートする。具体的にいうと $GF(2^n)$ 上の加減乗除算になる。それに加えて、ビットレベルの論理演算もサポートされていて、いうなれば、計算機のプログラムの振る舞いを模倣できるものだな」
「それはどういうところで役に立つんですか？」
わざわざ制限されたビット数の演算を使うとうれしいのはなんだろうかと思いました。
「もともと、SAT ソルバの重要な応用として、モデル検査というのがあってだな。モデル検査というのは、長くなるから、ここでは割愛するけど、ソフトウェアとかハードウェアがある種の性質を満たしているかを、静的に検査するものだ。これに対して、Bit-Vector をサポートした SMT を使えば、より効率よく検査できるという、まあそういう話もある」

なるほど、よくわからない分野の話だ。
「ところで、あずにゃん、ビット演算は好きかな？」
「え、唐突ですね。あれですか？ ハッカーのたのしみとかの。あれ、私はあんまり得意じゃないかなあ」
「そうそう、それ。32 ビット整数の、立ってるビットを数えるコードってあるよね」
うーん、どんなんだったっけ。シフトして足して、そんなんだったかな？

```
popCountNaiive n =
  sum [ (n `shiftR` i) .&. 1 | i <- [0..31] ]
```

「まず、簡単なコードを書いてみるね。32 個のビットをそれぞれシフトして、最下位取り出して、足し合わせるだけの簡単なコードだよ」
「自明なコードですね」
「で、次に、凝ったコードを書く！」

```
popCountFast n0 =
  let n1 = (n0 .&. 0x55555555) + ((n0 `shiftR`  1) .&. 0x55555555)
      n2 = (n1 .&. 0x33333333) + ((n1 `shiftR`  2) .&. 0x33333333)
      n3 = (n2 .&. 0x0f0f0f0f) + ((n2 `shiftR`  4) .&. 0x0f0f0f0f)
      n4 = (n3 .&. 0x00ff00ff) + ((n3 `shiftR`  8) .&. 0x00ff00ff)
      n5 = (n4 .&. 0x0000ffff) + ((n4 `shiftR` 16) .&. 0x0000ffff)
  in n5
```

「わあ、これは正しいのかぱっと見てもわからないですね」
「そういう時は、普通は、テストとかを書くものだけど、SMT ソルバなら」

```
popCountIsCorrect x =
  popCountNaiive x .== popCountFast (x :: SWord32)
```

「定理が書けちゃうわけですね！」
「そのとおり！ あずにゃんも慣れてきたね。んでもって、」

```
> prove popCountIsCorrect
Q.E.D.
```

「Q.E.D. というわけだよ。あず SAT 君！！」
「すごいです！ これ、コードが間違ってた場合はどうなるんですか？」
「んー、じゃあ、ちょっとバグを入れてみようか」

```
popCountFast n0 =
  let n1 = (n0 .&. 0x55555555) + ((n0 `shiftR`  1) .&. 0x55555555)
      n2 = (n1 .&. 0x33333333) + ((n1 `shiftR`  2) .&. 0x33333333)
      n3 = (n2 .&. 0x0f0f0f0f) + ((n2 `shiftR`  5) .&. 0x0f0f0f0f)
      n4 = (n3 .&. 0x00ff00ff) + ((n3 `shiftR`  8) .&. 0x00ff00ff)
      n5 = (n4 .&. 0x0000ffff) + ((n4 `shiftR` 16) .&. 0x0000ffff)
  in n5
```

「どこが、変わったんでしょうか？」
「シフトの数を一箇所変えたんだけど、ぱっと見わかりづらいよね。でも、そんなときでも、証明があれば」

```
> prove popCountIsCorrect
Falsifiable. Counter-example:
  s0 = 526123024 :: SWord32
```

「ほ、ほんとに、反例つきで反証が――――！」
「まだまだ、ここから C のコードを生成することも可能なのだよ」
「えっ、C のコード――――！？」
「ちょちょいと、生成用のコードを書いてやって」

```
main = do
  compileToC Nothing "popCount" $ do
    x <- cgInput "input"
    cgReturn $ popCountFast (x :: SWord32)
```

「これを実行すると――――」
「ど、どうなるんですか！？」

```
/* File: "popCount.c". Automatically generated by SBV. Do not edit! */

#include <inttypes.h>
#include <stdint.h>
#include <stdbool.h>
#include "popCount.h"

SWord32 popCount(const SWord32 input)
{
  const SWord32 s0 = input;
  const SWord32 s2 = s0 & 0x55555555UL;
  const SWord32 s3 = s0 >> 1;
  const SWord32 s4 = 0x55555555UL & s3;
  const SWord32 s5 = s2 + s4;
  const SWord32 s7 = s5 & 0x33333333UL;
  const SWord32 s8 = s5 >> 2;
  const SWord32 s9 = 0x33333333UL & s8;
  const SWord32 s10 = s7 + s9;
  const SWord32 s12 = s10 & 0x0f0f0f0fUL;
  const SWord32 s13 = s10 >> 5;
  const SWord32 s14 = 0x0f0f0f0fUL & s13;
  const SWord32 s15 = s12 + s14;
  const SWord32 s17 = s15 & 0x00ff00ffUL;
  const SWord32 s18 = s15 >> 8;
  const SWord32 s19 = 0x00ff00ffUL & s18;
  const SWord32 s20 = s17 + s19;
  const SWord32 s22 = s20 & 0x0000ffffUL;
  const SWord32 s23 = s20 >> 16;
  const SWord32 s25 = s23 & 0x00ffffffUL;
  const SWord32 s26 = s22 + s25;
```

```
  return s26;
}
```

「C のコードが手に入るというわけだよ！ しかも、証明つきの！」

「なんと！」

証明つきのコードといえば、Coq とか Agda が有名だし、Coq も C のコードを生成することはできます。でも、これは...。

「ほかのコードを証明できる処理系の Coq とかに比べると、あまりにも簡単に証明できすぎですよこれ。なにかがおかしいです！」

「確かにそうだね、でも」

唯先輩はチッチッ、と楽しそうに片目をつぶって、

「SMT は、証明を探索をするプログラム。Coq は基本的には証明を自分で書くもの。SMT は簡単に書ける代わりに、いつでも証明を見つけてくれるとは限らないのだよ」

「なるほど、うまくいくときにはうまくいくということですね」

うまくいくときにはうまくいく、とは何もいっていないような気もしたけど、実際にそういうものだから仕方がないとも思いました。

充足可能問題というところから、気づけば線形計画法、果てはプログラムの静的検証、自動生成までやってくるとは、最初からは想像もしていませんでした。

「なんか、今日はすごく疲れました」

「なんだ、梓。この程度で音を上げてちゃあ、けいさん部の次期部長の座は譲れないなあ。一番おもしろいのは、Uninterpreted Function なんだぞ。ここから free-object につながって、圏論の世界とプログラムの世界が———もごもご」

そうでした。新入部員は私一人だから、いずれはそうなるんです。

「さて、じゃあ SMT を覚えた梓には、宿題を出しておこう」

澪先輩がいつぞやのように、どさりと雑誌を置きました。

「ま、またニコリですか〜？」

「そうだ。明日までに全部解いて来るように！」

「そ、そんな〜」

当時の私にとっては、つらく厳しい道のりでした。しかし、SMT というものがもたらす可能性について、なにやら底知れぬものを感じていた私は、拒むこともできなくなっていたのです。

「———ヤッテヤルデス！」

———

梅雨が晴れ、雨雲が去るのと同期するように、私の頭にかかった霞も徐々に晴れてきました。SMT もだんだんうまく扱えるようになってきたところで、ふと思って、改めて SAT に立ち返ってみることにしました。

「うーん、充足可能問題には、すべて充足できるか、そうでないかを求める以外にもいろいろあるのか〜」

なるべくたくさんの節を充足させる、MaxSAT 問題、節に対して重みがついていて、充足する節の重みの和を最大化する WeightedMaxSAT 問題、節のうち、必ず充足しなければいけない節が指定されているとき、それらを充足しながら、残りの充足節を最大化する PartialMaxSAT 問題、変数の値を true/false ではなく、n 種の値に一般化した CSP———。

これらの問題のソルバは、単純な SAT のものに比べると、効率はよくないようです。

「それ、SMT でできるよ」

「唯先輩！？ い、いつの間に部室に」

「あはは、ぴくってしちゃって。あずにゃんはかわいいなあ。すりすり」

「もう、やめてくださいよお」

口ではそういいながらも、唯先輩のこのスキンシップが、私はだんだんまんざらでもなくなってきていました。

「おやおや、口ではそういってはいるが、こっちのほうは正直だのう」

「変態的な発言やめてください！ こっちってどっちですか」

「それは、そっちに決まっておるじゃないか。ぐへ、ぐへへ」
ガチャリ。
「おー、唯と梓はいつも仲がいいな」
「澪先輩、助けてくださいー」
澪先輩は、いつものように PC を起動しています。私も、すっかりとけいさん部に馴染んできたものです。
「そうだ、SMT でできるっていうのは」
「あ、そうそう。あずにゃんが話の腰を折るから」
「えっ、私ですか？」
だめだ。唯先輩のペースに引き込まれては。
「あれー、よく覚えていないや。まあ、それはおいておいて」
「はい」
「あずにゃんは、MaxSAT に興味があるのかな？」
「ええ、なんというか、今調べていたらそういうものがあるって知りまして」
「なるほど、なるほど」
唯先輩は、眉間にしわを寄せながら何かを真剣に考えているふうですが、この人の場合は、本当は何も考えてないんじゃないかとか思ったりもするのです。
「うーむ、それならやっぱり SMT でもできるよ」
「SMT ですか」
「SMT だよ。SMT があれば（大体）なんでもできる！ S・M・T・だー！！」
ほんまかいな、と突っ込みをいれずには入れないこの台詞。
「MaxSAT を SMT にうまくエンコードできるんですか？」
「そうだね。割と簡単だと思う」
ちょっとまってね、と言いながら、ホワイトボードに向かって、なにやら式を書き始めました。
「もとの CNF がこれで――――」

- $c_1 \wedge c_2 \wedge c_3 ...$

「それが、こうなって」

- $(w_1 \rightarrow c_1) \wedge (w_2 \rightarrow c_2) \wedge (w_3 \rightarrow c_3) ...$

「最後に、この節を追加すれば」

- $(score \iff w_1 + w_2 + w_3 + ...)$

「よし！ いっちょあがり！」
「先輩、これは一体」
「あ、説明するね。まず、一つ目の式が、元の MaxSAT に渡す CNF で、この式の充足節を最大化したい。そのために、各節が充足したかどうかのフラグを立てる。それが二つ目の式だね。最後に、立っているフラグの数が score と等しいですよ、という節を追加する。これが三つ目の式だね」
「これの score を最大化するんですか？ でも、SMT は充足可能問題しか解けないですよね」
「なら、充足可能問題にしてやればいいのだよ。score を最大化することはできないけど――――」
「あっ、score がある一定以上で充足可能か、という問題なら解けますね」
「その通り。それが解けるなら、後は簡単だね」
「二分探索で、充足可能な限界を調べてやるんですね！」
「よくできました！ ご褒美にあずにゃんには、この猫耳を――――」
「いりません！」
猫耳はまだちょっと恥ずかしいのでした。
「じゃあ、あずにゃんも成長したことだし、新しいあだ名を進呈しよう！ うーん、こういうのはどうかな。あず MaxSAT！」
「どこかの芸人みたいじゃないですか！ いりません」
なんで私ばかり、変なあだ名をつけられるんだろう。

「ちぇー」

あからさまにいじけた顔をしている先輩もなんだか最近はかわいらしく思えてきました。

「ともかく、これで MaxSAT が、SMT ソルバを使って解けるようになったわけですね。しかも Weighted-MaxSAT や PartialMaxSAT にも簡単に応用できそうです」

「ふふふ、これも SMT の簡単な応用だよ」

「CSP への応用は、さらに簡単そうですね」

「CSP の SMT へのエンコードは、あず MaxSAT 君への宿題にしておこうじゃないか！」

「その呼び方やめてください」

ここでふと気になりました。

「あれ、SMT を使って、MaxSAT をエンコードするとき、さらりと最大化問題を解いていませんでしたか？」

「あれ、そうだね」

「SMT を使うと、最適化問題が解けてしまうんでしょうか」

「うーん、そうだなあ。一般にというわけではないと思うけど」

唯先輩は頭を一ひねりして、

「大体の場合は解けると思うよ。まず、目的関数、つまり最適化したい値が、二分探索可能な値なら、制約を入れて二分探索すればいい。これがたぶん一番効率もいいはずだよ」

「さっきのエンコード方法ですね」

「二つ目の方法としては、充足解を求めながら、徐々に良い解を求めるというものかな。まずひとつ充足解を求めて、その解よりも良い、という条件を加えて、再度充足解求める」

「これは二分探索と比べて良くないんですか？」

「たぶん、何かひとつ解を求めるということに関しては悪くないと思うけど、二分探索とは違って、イテレーションごとに指数的に答えに近づくわけじゃないから」

「牛歩になる可能性が高いと」

「まあ、そうだね」

「それで、最後三つ目の方法。量化子を用いる方法」

「量化子？」

聞いたことはあるような気がしましたが、SAT に持ち込んで良いものだったのかな。

「Quantifier といってね、SMT ソルバの中には、これが使えるものもあるよ。z3 ではサポートされているね」

「SMT 何でもありですね――――」

「そこが SMT のいいところなんだよ！」

なぜか唯先輩が胸を張って自慢しています。

「で、Quantifier ってのは、∀とか、∃とかそういうやつだね。最大化したい関数が $p(x)$ なら、最大値を求める制約式は、

$$\exists x_{\max}.\forall x'.p(x_{\max}) \geq p(x')$$

になる」

「なるほど、これなら $x_{\max}$ にアサインされた値を取り出せば、一発で最適解が求まりますね。――――でも本当にこんな式を汎用のソルバが探索で解を導き出せるんでしょうか？」

「難しい質問だね。現状では、できることもあるし、できないこともある。できたとしても、明らかに遅いときもある。組み合わせる Theory との相性もあって、 Linear arithmetic と組み合わせた場合はうまく動くことが多いけど、非線形の式と組み合わせた場合は、ギブアップしちゃうことが多いかなあ」

「やっぱり、万能とはいきませんね」

こんなのができてしまったら、本当に人類の脳の衰退の危機です。あぶない、あぶない。

「まあでも、その辺はソルバの今後の発展に期待というところだよ」

「ところで、さっきの MaxSAT だけど」

PC に向かって、なにやらプログラムを書いていた澪先輩が、驚いた表情でなにやら画面を見つめています。

「専用の MaxSAT ソルバよりも、SMT にエンコードして、二分探索したほうが速い」

SMT にエンコードして、ソルバで殴ればいい。ゲームだとクソゲーだけど、現実だと、これほどありがた

いことはないのでした。

---

8 月、夏。世間では甲子園が盛り上がっているころ、けいさん部でもひとつのイベントがありました。

「プログラミングコンテスト？」

どうやら、部室に飾られたたくさんの表彰状やらなにやらのひとつは、そのコンテストでのものみたいでした。

「ふふふ、我がけいさん部は、毎年優勝しているのだよ」

「毎年って言っても、去年一回しか出てないけどな」

「一回でて一回なんだから、毎年で良いんだよ」

「かなり語弊があるがな」

へえ、先輩たち、ずっと遊んでいるように見えて、やるときはやる人たちなんだな。

「あずにゃん今、お茶しか飲んでないのに、なんで勝てるんだって、思ったでしょ？」

「ぎ、ぎくぅ。そんなことないですよお」

し、心臓に悪いなあ。

「まあ、実際そうだぜ。私たちがやってるのは、半ばゲームバランスを崩壊させるようなことだからなあ」

「実際そうだから、仕方ないよね」

「何ですか、そのもったいぶった言い方」

「あら、梓ちゃんにはわかると思ったけど」

まさか、先輩方、SMT を…？

「い、いいんですか？ そんなの。徒競走に車で出るようなものじゃないですか」

「禁止されてなければ良いんじゃないか？ ルールはちゃんと読んでるぜ」

平然と言ってのける。

「いいか、梓。現実の問題にルールなんてないんだぞ。これはコンテストだからそういう疑問も出るけど、実際の問題なんて解くためだったら何をしても良いんだ。そのために、人類の英知の結晶を使うのは、いたって当たり前のことだろう？」

「そ、それはそうですけど」

「私はな、ずいぶん昔からプログラムを書いているけど、ある日気付いちまったんだ。なんで私は同じようなプログラムを何回も書いているんだろう。なんで同じようなアルゴリズムを何回も書いているんだろう。なんで同じようなプログラムで同じようなバグを作っているんだろう。なんで同じようなバグを取るのに、同じような苦労を何回もしているのだろう。そして、私はなんで、それに何の疑問も抱いてこなかったのかって」

私は、プログラミング暦が浅いから、まだ、毎日がわからないことだらけで、毎日が新鮮で、そんなこと考えもしなかったけれど。

「たくさんの同じようなファイルを手作業で作る人を馬鹿にする。たくさんの同じような Excel ファイルの集計を手作業でやる人を馬鹿にする。それがプログラマってもんだ。だけどな、なぜか鏡を見れないヤツが多いんだ。そこに映る、たくさんの同じようなプログラムを書いている自分の姿を」

メタな視点。あるものを別の場所から観測するのは、当たり前で、普通のことだけど、その中にいる人が、自分自身を観測するのは、普通のように見えて、普通のことじゃない。

「たくさんの単純作業を自動化して、その仕事を奪ってきたのはプログラマ。ならば、その仕事は同じくプログラマに奪われるべきだよなあ」

はじめに 1 があった。 1 はその世界のすべて。次に、2 が生まれた。2 には、1 を見ることができたけど、自分がどういう存在かを自覚することはできなかった。そして、次に、1 から 2 を作るのと同じようにして、2 から 3 が作られた。3 はその事実と、2 と 1 を見比べることにより、自分がどうやって作られたのかを初めて知る。

「1、2、３、たくさん」

「梓、何だいそれは」

「唯先輩が言ってた」

たしか、3SAT が NP 完全だって話だった。3 まで数えられたら、それより上は、自動的に数えられる。

「3 って数字は何か特別なものを感じます。higher-order が有効なのも多くの場合 3 まで」

これは、帰納的な何かと関係あるのか。4 を数えることに疑問を持った人たち。4 を数えることを拒否した人たち。けいさん部。それは人類の衰退などではなく、約束された恒久的な発展だったのだろうか。

「だから、私は SMT に希望を見出したんだ」

SMT ソルバは破壊的技術だ。十徳ナイフだ。多くの場合それでことが足りてしまうようになる。当然、個々の問題には専用のソルバのほうが良い性能を示す場合もあるだろう。しかし———。

「特殊なソルバがコスト的に見合うラインは、今後確実に上がっていくだろう。なぜならば、SMT ソルバの性能が上がれば、あらゆる問題が高速化されてしまうからだ。今の SMT で難しいところがあれば、SMT にコミットすればいい。新しい Theory を追加すればいい。制約ルールを追加すればいい」

そうすれば、それは人類の永遠の財産になるのだから。

かつて、いろいろなものが通った道。マシン語のコードを効率良く書くにはどうすれば良いか。マシン語のコードを書かなければ良い。コンパイラを使わない積極的な理由のある分野は、いまや、ほとんど、ない。

最適化問題を書きたい。効率の良いアルゴリズムを実装したい。バグのないプログラムを書きたい。早く書き上げたい。どうすれば良いか。答えは簡単だ。プログラムを書かなければいい！

「そうか、そうだったんですね！　先輩は、この、けいさん部は———！」

お茶ばかりしている理由、 SAT をいきなり私に教え込んだ理由、お茶ばかりしているのに、プログラムが書ける理由。

「ふふ、やっと気付いたみたいだね」

「あずにゃん、おめでとう」

「思ったより早かったな」

「じゃあ、改めて梓ちゃん」

「『けい３部』にようこそ！」

「———けい３部の部室って、こちらでしょうか？」

四月の陽光の中、うたた寝から目覚めると、見知った顔が二つ。

「にゃー、あ、憂、純、おはよう」

「あんたがどうしてもって言うから、来てあげたわよ」

「梓ちゃん顔に線ついてる」

そうだ。私は、先輩達の意思を継いで、このけい３部の部長になったんだ。

「ところで、この部って何やる部なの？」

そうだなあ———。まずは。

「free object って、知ってる？」

けんろん！ へ続く

第V部

簡約!? 入力娘　Go! (2013年夏)

第23章

めたせぴ☆ふぁうんでーしょん

— @master_q

23.1 はじまり

西暦 2013 年。凶の年。ソフトウェアエンジニアの多くは Web の世界に住んでいた。じゃうぁすくりぷと、あんどろいどじゃうぁ、おぶじぇくとしー、しーしゃーぷ。高機能な言語と環境をあやつり、彼等は栄華をきわめていた。きらびやかな UI、きらびやかな通信。またある人々はしーげんごとしーぷらすぷらすをあやつり、匠の技で複雑なハードウェアをあやつり、複雑多機能なうぇぶぶらうざを作り、そしてそれらを支えるオペレーティングシステムりなっくすを作っていた。

しかしぼくらは幸か不幸かそのどちらでもなかった。オープンソースのソフトウェアを使って組み込みデバイスを作っていたのだ。世間からは考えられないほど古いバージョンのオープンソースを使い、世間からは考えられないほど汚れたソースコードを弄り、そして世間からは考えらない奇妙なデバイスの集合を組み合わせていた。それだけならまだよいのだが悲しいことに作った製品への世間からの評価は低かった。ぼくらの中にはこの製品化の渦に飲み込まれて息絶える者もいた。

そうして生き延びたぼくたちは現実から逃避するようにインドネシアに逃げのびた。もう設計などしたくはなかったのだ。自然豊かなこの国で、ぼくたちは安らぎ、ソフトウェアのことなど忘れかけていた。それでも組み込みシステムへの想いは捨てきれず、何か改善する手はないかと失われたテクノロジ **型システム**を学び会得していた。しかし祖国を離れてまともな仕事がある訳もなく、先進国向けの Web サービスを作るアルバイトで食い繋いでいた。もはや組み込みの民はこのまま消えゆくしかないのか……そう思いかけたとき……

「ワシが **めたせぴ**☆ でゲソ!」

どこからともなく静かだが力強い声が響いた。

「これからおぬし達は一本の糸をたどることになるでゲソ。この糸は古えのテクノロジー型システムによって綿密に計算/予測された運命なのでゲソ!」

なにかわからないが、安心とやはり不安がいりまじって混乱する。この娘はいったい何を言っているのだ？

「今おぬし達を囲んでいる状況はよくわかっているでゲソ。増えるカスタムシステムコール。多種のデバイス。崩壊しつつある旧来の CPU アーキティクチャ。POSIX API 上に構築された複雑なミドルウェア。絶え間ない製品リリース。猛進進化する型付けされていないオープンソースソフトウェアへの追従」

この娘は何者だろう？ しかし、そうだ。その通りだ……

「そしておぬし達は今、型システムを手にしている。それがおぬし達だけが持つ特性、もしくは力じゃなイカ？ それならこれから取るべき道はあまりにもわかりきっているでゲソ!!!」

そう言うと、めたせぴ☆と名乗る娘は消えてしまった。……あれは夢だったのか？ 今となってはぼくたちにもわからない。

23.2 レストランにて

ぼくたちはちょっと贅沢をして海辺のレストランに来ていた。いつも作っている Web サービスではない別の話をしたくなったのだ。さっきの娘にそそのかされたわけじゃない、ほんの気晴らし。それでもぼくたちの話題はいつも組み込みシステムに向いた。あの頃ぼくたちを苦しめていた問題の根っこは何だったのだろう？ 製品開発において最も重要なのは不具合の解消だ。不具合の解消方法には根本対策と暫定対策がある。根本的に対策できれば 100 点満点だが、工数の関係上たいていの対策は暫定策に終わる。暫定対策を施して

も似た不具合が将来必ず起きることになる。不具合の根本原因を放置しておくと長い時間をかけて毒が体全体にまわり、そしてプロジェクトそのものが死ぬのだ。ぼくらは先輩からこの法則を何度も教えれられ、そして痛いほど体で味わっていた。だからぼくらが集まるといつも “組み込みシステムの持つ不具合の根本対策” に話題が向くのだった。

レストランでの話し合いは長時間にわたり、そしてぼくらは誰しもが落ちつく結論に辿りついた。

「やはり最も大きな不具合は実行時エラーの増加なんだ」

そうこの結論には誰だって辿りつく。実行時エラーを減らすために色々な手法が提案されているのがその証拠だ。UML による設計もそうだろう。モデル駆動型アーキテクチャもそうだろう。契約プログラミングもそうだろう。でもぼくたちがいた大規模組込開発のドメインで、それらが特効薬になったという話は聞いたことがなかった。

「努力目標は機能しない」

だれかがそう言った。

ぼくらは型推論を知っていた。具体的な理論はよくわからないが、実行時エラーの一部を魔法のようにコンパイル時に知ることができた。ぼくらはこの特性が気に入って、日々の Web サービスの開発にも使っていた。

「どうして kernel ドメインで型による設計が使えないんだろう？」

まただれかがボソっとつぶやいた。ぼくらは UNIX ライク kernel をカスタマイズ/メンテするチームだった。その kernel を使うプロジェクトは 10 年以上も続いていた。(国を逃がれた今、そのプロジェクトがどうなったのかはわからない)

「型で kernel を書くプロジェクトはたくさんある [*1] みたいだよ。 OCaml で kernel を書くとか手堅い選択肢かもしれないよ？」

「うん、ぼくもソース読んでみたさ。でもこれはオモチャ kernel だよ。割り込みはポーリングで拾っているしバスドライバさえない。移植性は皆無だ。この kernel を拡張していっても、かつてぼくらがいたドメインで使えるようにはならないよ」

ちょっとビンタン [*2] を飲みすぎたみたいだ。ぼんやりする頭を海からの風がすぎる。ぼくらのレストランでの会合はいつもこんな感じだ。この国はいい。こうやってぼんやりと昔話をして過ぎる時間。もういいじゃないか。そんなばかでかい問題。もう、ぼくらの知ったことじゃないよ。

でも今日はいつもと少し違っていた。

「いきなりスクラッチから書くのが無理なんじゃないか？」

いつもの話題がもう一つ先に進んだのだ。

「UNIX ライク kernel はもう世界に浸透しきってしまった。いまさら新しいインターフェイスの kernel がすぐに受け入れられるとは思えない。それなら型による kernel 設計も自然に UNIX ライク kernel を目指すべきなんじゃないかな」

正直ぼくらは POSIX インターフェイスにもうお腹いっぱい [*3] だった。しかし kernel は所詮ただの kernel。使ってくれる人がいなければゴミ同然だ。

「kernel の品質維持のためにドッグフード [*4] が有効なことはもう 20 世紀に結論が出ている。ドッグフードを行なうにしても POSIX インターフェイスがなければ今使っているプログラムのほとんどが使えない」

ドッグフードというのは “自分達の開発したソフトウェアを使って自分達のソフトウェアを作る” という意味だ。この開発手法は単純で自社製品の品質維持に効果的だ。というのも品質が維持できなければ開発行為自体に大きく影響するからだ。ドッグフード開発されていないために品質が悪化する例としては、社内部門が作った使いにくい社内ツールがイメージしやすい。社内部門が自分達が使わないソフトウェアを開発して、それ以外の全社員が使うことがある。このような会社はソフトウェアに対する理解が浅いと言えるかもしれない。

[*1] “Haskell/OCaml 製の OS って何があるんでゲソ？ ” http://metasepi.org/posts/2012-08-18-haskell-or-ocaml-os.html

[*2] インドネシア定番のビール http://www.multibintang.co.id/

[*3] 参考: “Copilot への希望と絶望の相転移” http://www.paraiso-lang.org/ikmsm/books/c81.html

[*4] 闘うプログラマー http://www.amazon.co.jp/dp/4822247570

「試しに POSIX インターフェイスでない独自 API の kernel を作ってドッグフード可能になるまでの道のりを考えてみようよ」
「そうだなぁ、ドッグフード可能にするためには kernel をコンパイルするコンパイラとその他日々使う最低限のソフトウェア全部を作り込まなければならないはめになるよ (図 23.1)。でも、もし POSIX インターフェイスをまずは採用すれば既存の GCC コンパイラや Firefox のような Web ブラウザも比較的容易に移植できる。だから新しい kernel を作り込むことに集中できると思う。もしその kernel が安定動作するようになれば、POSIX ではない新しいインターフェイスをドッグフードの開始後にゆっくり作り込むことに後から挑戦することだってできるじゃないか」

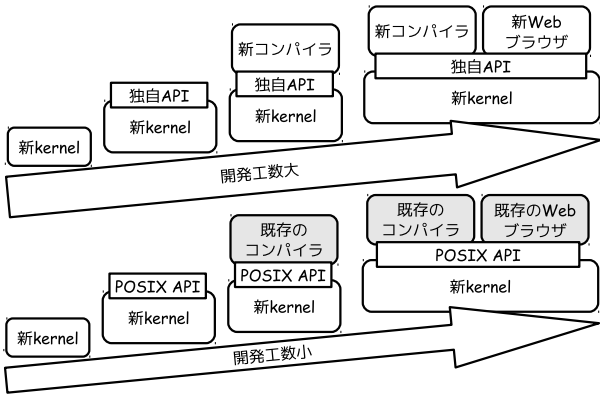


図 23.1: POSIX API を採用することで OS 開発の工数を削減

「それなら **スナッチ** [*5]すればいいんじゃない？」
耳慣れない用語だ。
「アイデアレベルだけど、既存の C 言語で設計された kernel を関数単位かモジュール単位で強い型を持つ言語で置換すればいいんじゃないかな」
砂の上に図を描きはじめた (図 23.2)。ぼくらの中には一人だけいつも非現実的なアイデアばかりを主張するやつがいる。

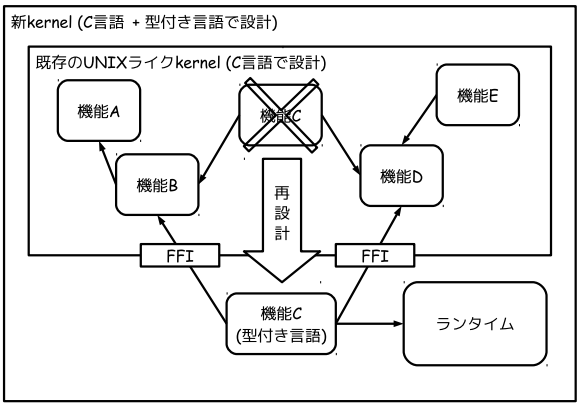


図 23.2: スナッチ設計: 既存 kernel を少しずつ設計置換

「そんな簡単に言うけど可能なのかな？ この図のモデル……ああ、めんどうだから以後 **スナッチ設計** と呼ぼう、このスナッチ設計を実現するためには “相応のコンパイラ” と “POSIX インターフェイスがなくても動くランタイム” が必要になるけれど、そんなコンパイラ実在するのかい？ OCaml を使うにしてもランタイムをずっとスリムにしなきゃいけない。それから割り込みベースで動作している UNIX ライク kernel をスナッチするのであれば、割り込みコンテキストを扱う方法も考えないと……」
また壁だ。いつもこの話題はどこかで大きな壁にはばまれる。
「ごめん、ちょっと酔ったみたいだ。散歩に行ってくるよ」
ぼくは一足早めにレストランを出ることにした。

23.3 砂の中

家に帰らずにぼくはまだ暗い海岸にいた。あれ？ 暗闇にさっきの娘が見えるような。名前は……なんだっけ。
「めたせぴ☆でゲソ! いいかげん覚えてほしいでゲソ」
元気の良い声が場違いに響き渡った。
「なにを悩んでいるんでゲソ？ さぁワシに話してみるが良いでゲソ」
ニヤニヤ笑いが癪に障る。
「おぬしの思っていることは全部お見通しでゲソ! どうやら自分の使いやすいコンパイラがどっかに落ちて

[*5] コナミのゲームであるスナッチャー http://en.wikipedia.org/wiki/Snatcher から

ないか砂の中を探しているようじゃなイカ？ すでに小さなバイナリを吐く型推論を持つコンパイラに出会っているのに、おぬしはそのコンパイラが最適解なのか悩んでいるようでゲソ。しかし最適解を探しているだけではいたずらに時間が過ぎるだけということも理解しているはずじゃなイカ？」
いや心配しているのはそこじゃないんだ。
「曳光弾とプロトタイプ [*6] でゲソ」
うん？
「自分が作っているものが製品化できるのか、それとも単に実証実験にのみ使っていずれ捨てるのか、作る前にあらかじめその決断をしておけば、どんなコードも無駄にはならないんでゲソ」
プロトタイプはわかるんだけど……曳光弾ってなんだろう？
「おぬしはあまり本を読まないようでゲソ。曳光弾は “射撃後飛んでいく間に発光することで軌跡がわかるようになっている弾丸” のことでゲソ。おぬしは今、暗闇のなかにいるんじゃなイカ？ 進む方角さえわからないでゲソ。もっと言えば本当に解が存在するかさえ不安なんじゃなイカ？ そんな時、おぬしが取り得る選択肢は二つしかないのでゲソ。プロトタイプは要求から製品に近いものをでっちあげる作業でゲソ。当然品質は劣悪なものになるはずでゲソ。これでは最終的な製品にはならない、けれど方角はわかるかもしれないでゲソ。曳光弾はプロトタイプとは違い、しっかりとした品質のソフトウェア部品を作る行為でゲソ。この曳光弾は運がよければ製品に搭載できるかもしれないでゲソ。品質が確保できているからじゃなイカ。ただしもちろん運わるく製品の方向性とは見当違いの部品かもしれないでゲソね。それでも曳光弾を作ることによって、その近傍にあるモノが暗闇の中で少しだけ見えるはずでゲソ。それもまた前進と言えるんじゃなイカ？」
ややこしいな……えっとこの場合は？
「かんたんじゃなイカ。コンパイラが曳光弾。その他のコード全てはプロトタイプにすぎないんでゲソ」
よく言いきれるな……
「固く考えることはないでゲソ。曳光弾とは言っても好きなコンパイラをベースにすればいいんでゲソ。どーせどのコンパイラを選んだにしても不足した機能を追加しなければならないことには変わりがないのでゲソ。この間おぬしはコンパイラを物色していたんじゃなイカ？」

そう、あの時は jhc [*7] が組み込み開発にとても良い特性を持っていた。コンパイラが組み込みに向いているかどうかは三つの指標を比較すればだいたい見当がつく。

A. バイナリサイズ
B. 未定義シンボル数
C. 依存ライブラリ数

A に関しては自明だ。組み込み開発では使えるメモリ領域が限られていることがある。もちろんスワップなど使えない。プログラムのサイズは小さいにこしたことはないのだ。C に関しても比較的理解しやすい。プログラムが別のライブラリに依存している場合には POSIX API のない世界にそのライブラリもろとも移植しなくてはならなくなる。それでももしかするとプログラム自体を変更すればライブラリ依存を削除できる可能性はある。B は少しわかりにくい。プログラムバイナリがあるライブラリへの未定義シンボルを多く持つということは、そのライブラリへの依存度が高いということだ。ということはプログラム本体へ修正をほどこしても、その依存度が強いライブラリへの癒着はほどけない可能性が高くなる。つまり A,B,C の値それぞれが小さければ小さいほど POSIX の外の世界への移植度が高いことになるのだった。

いくつかの型推論を持つコンパイラについて、三つの指標の値を調査して表にまとめてみよう。

[*6] 達人プログラマー `http://www.amazon.co.jp/dp/4894712741`
[*7] Jhc Haskell Compiler `http://repetae.net/computer/jhc/`

コンパイラ	バイナリサイズ (byte)	未定義シンボル数	依存ライブラリ数
GHC-7.4.1	797228	144	9
SML#-1.2.0	813460	134	7
OCaml-4.00.1	183348	84	5
MLton-20100608	170061	71	5
jhc-0.8.0	21248	20	3

表 23.1: “hoge” と印字するプログラムに見るコンパイラの特性

この表を見るかぎり jhc は小さなバイナリを吐き、さらには POSIX インターフェイスへの依存度も低い。そのため kernel ドメインにランタイムを移植する工数も少なく済みそうだ。ここまではわかっている。
「ベースにするコンパイラは決まったようでゲソね。あとはプロトタイピングを繰り返して、コンパイラに不足している機能を燻り出せばいいんじゃなイカ」
そうはいっても適切な題材が思い付かない……
「絵を描く前には白いキャンバスにスケッチをするものでゲソ」
そう言うとまたあの娘は消えてしまった。

23.4 最初のスケッチ: POSIX からの脱出

POSIX 上で動いているプログラムと POSIX の外で動くプログラムとの違いとはなんなのだろう (図 23.3)。

POSIX 上で動作する Haskell プログラムの内部は大きく二つの部分にわかれている。Haskell コード由来の部分とランタイム由来の部分だ。Haskell コードはどのような外部 API にも依存していないハズだ。FFI などを使わなければだけれど。Haskell コードはランタイムのみに依存している。さらにそのランタイムは外部のライブラリに依存しているはずだ。libc だけか、libpthread もか、とにかくなにかのライブラリ依存があるはずだ。そうでなれば副作用を実現できないからだ。そのライブラリ群は最終的には POSIX API にいきつく。その API を実現しているのは libc と kernel のセットだ。

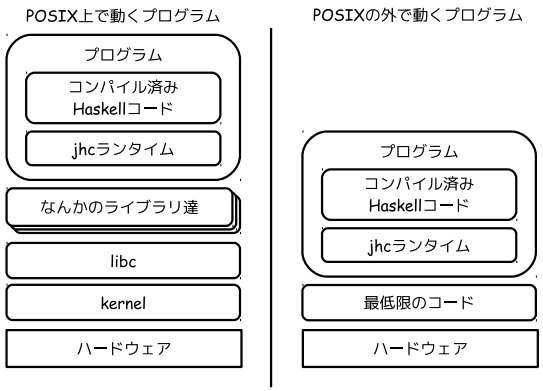


図 23.3: POSIX 内と外の Haskell プログラムの想像イメージ

さてこの Haskell プログラムを POSIX API のない生のハードウェアが見える世界に移植するにはどうしたら良いのだろう？ ランタイムが依存する POSIX API のみをカバーする最低限のコードを用意すればいい。“最低限のコード”。それが Haskell プログラムを動作させるために必要な Haskell で書けない部分、プリミティブだということになる。
ぼくはそんなプログラムを作る練習になる題材を探してみることにした。

23.4.1 調査

ぼくは 9 セルバッテリを付けた ThinkPad [*8] を開いた。ディスプレイから無機質な xmonad のタイルが目に入る。ぼくらは以前 NetBSD kernel を製品開発に採用していた。その頃のクセがまだ残っていて、ぼくの PC

[*8] 今回対象とする実行環境は Debian GNU/Linux amd64 sid 2013/06/10 時点。GHC のバージョンは 7.6.3。

には NetBSD のソースコードリポジトリを丸ごと rsync [*9] してあった。ネットワークが不通になっても開発できるようにするのが先進国の外での常識だ[*10]。

jhc で実験的なアプリケーションを作る方針を決めたは良いが、現時点では重大な制約がある。それは jhc の吐くバイナリが再入可能ではないということだ。jhc の吐くバイナリはコンテキストを一つしか持てない。もちろんスレッドも扱えない。つまり、jhc ではハードウェア割り込みを前提とした kernel を書くことは出来ないことになる。しかし何か kernel に近いところの題材が欲しい。NetBSD のソースコードツリーを探していたら格好の題材がすぐ見つかった。bootloader だ。

NetBSD bootloader は起動するとキーボード入力を待つ。この待ちがタイムアウトすると NetBSD kernel が自動的に起動されるのだが、タイムアウト前に何かキーを入力すると bootloader は独自のコマンドラインを起動する。NetBSD bootloader のコマンドラインを起動して、help コマンドを実行してみた結果が以下のログだ。

```
>> NetBSD/x86 BIOS Boot, Revision 5.9 (from NetBSD 6.0.0_PATCH)
>> Memory: 639/130048 k
Press return to boot now, any other key for boot menu
booting cd0a:netbsd - starting in 0 seconds.
> help
commands are:
boot [xdNx:][filename] [-12acdqsvxz]
     (ex. "hd0a:netbsd.old -s"
--snip--
help|?
quit
>
```

このコマンドラインから boot コマンドを使えば、ファイル名を指定して NetBSD kernel を起動できる。このコマンドラインループは NetBSD kernel が起動するまでの間割り込み禁止のまま動作する (図 23.4)。さすがにこのコマンドラインと bootoader の全てのコマンドを Haskell で実装するのは大変だ。しかし help と boot コマンドのみを Haskell でスナッチすることはできるのではないか？ どちらのコマンドももちろん割り込み禁止で動作するからだ。

ぼくはまず jhc のランタイム [*11] をよく観察してみることにした。

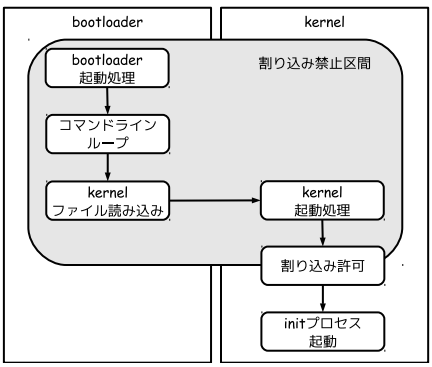


図 23.4: bootloader は割り込み禁止で実行される

[*9] “Linux 上での NetBSD バーチャル開発の方法” http://d.masterq.net/?date=20110207#p01
[*10] 筆者は東日本大震災後にこの開発スタイルを取るようになったようでゲソ。
[*11] https://github.com/ajhc/ajhc/tree/arafura/rts

jhc は Haskell コードをコンパイルすると単一の C 言語ソースコードを吐き出す。この C 言語ソースコードの中には Haskell ソースコードに由来する全てのロジックが写し込まれている。その後 jhc は GCC を呼び出して左記の C 言語ソースコードとランタイムのソースコードを一緒にコンパイルする (図 23.5)。すると無事実行バイナリができる訳だ。この jhc のコンパイルフローは途中で停止させることもできる。それが “-C” オプション [*12] だ。また “–tdir” オプションを使うことで中間生成されたランタイムソースコードを jhc が削除するのを防止できる。例えば次のようなコマンドを使えば生成された C 言語ソースコードとランタイムソースコードが手に入る。

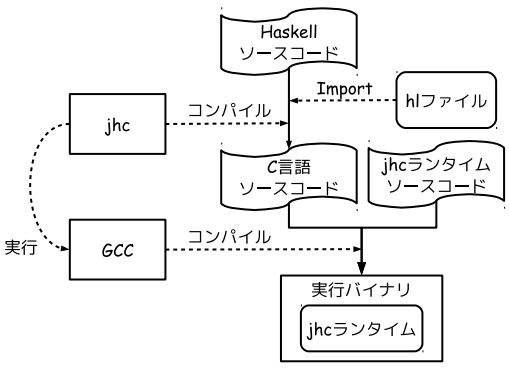


図 23.5: jhc のコンパイルフロー

```
$ echo 'main = print "hoge"' > Hoge.hs
$ jhc -C --tdir=rts Hoge.hs
$ ls
Hoge.hs  rts/
$ ls rts
HsFFI.h  jhc_rts_header.h  lib/  main_code.c  rts/  sys/
```

main_code.c が Haskell コードをコンパイルして得られた結果の C 言語ソースコード。その他のファイル群が jhc のランタイムソースコードだ。jhc はコンパイルを実行する毎にランタイムを再コンパイルする。ランタイムの規模が劇的に小さいからこそなせる技だ。

ここで生成された C 言語ソースコード main_code.c の中身を見てみる。

```
char jhc_c_compile[] = "gcc rts/rts/profile.c rts/rts/rts_support.c rts/rts/gc_
none.c rts/rts/jhc_rts.c rts/lib/lib_cbits.c rts/rts/gc_jgc.c rts/rts/stableptr
.c -Irts/cbits -Irts rts/main_code.c -o hs.out '-std=gnu99' -D_GNU_SOURCE '-fal
ign-functions=4' -ffast-math -Wextra -Wall -Wno-unused-parameter -fno-strict-al
iasing -DNDEBUG -O3 '-D_JHC_GC=_JHC_GC_JGC'";
char jhc_command[] = "jhc -C --tdir=rts Hoge.hs";
```

丁寧にも GCC に渡す予定だった引数が列挙されている。そこで指示通りに jhc が本来実行するはずだったコンパイルの続きを手動で実行してみた。

[*12] http://ajhc.metasepi.org/manual_ja.html#オプション

```
$ gcc rts/rts/profile.c rts/rts/rts_support.c rts/rts/gc_none.c rts/rts/jhc_rts
.c rts/lib/lib_cbits.c rts/rts/gc_jgc.c rts/rts/stableptr.c -Irts/cbits -Irts r
ts/main_code.c -o hs.out '-std=gnu99' -D_GNU_SOURCE '-falign-functions=4' -ffas
t-math -Wextra -Wall -Wno-unused-parameter -fno-strict-aliasing -DNDEBUG -O3 '-
D_JHC_GC=_JHC_GC_JGC'
$ ls
Hoge.hs  hs.out*  rts/
$ ./hs.out
"hoge"
```

なるほど。あっさり実行バイナリが手に入った。

23.4.2 設計

ということは下記のような設計が可能なのではないか？ (図 23.6)

1. bootloader の初期化が終わってから jhc が生成した C 言語ソースコードを呼び出す
2. Haskell を使って書かれたコマンドラインループが起動する
3. コマンドラインループが kernel 起動コマンドを検出したら既存の C 言語ソースコードに戻る

問題はどうやって実現するかだ。NetBSD bootloader は四つのライブラリを内包していて、比較的リッチな環境でプログラミングできる (図 23.7)。ひょっとすると jhc ランタイムの実行に必要な機能がすでにそろっている可能性もある。とにかくいきなり生のハードウェアの上で動くコードを jhc で作るよりもはるかに楽ができるはずだ。

残る疑問は jhc が生成した C 言語ソースコードはどのようにして呼び出されるのかだ。答は生成されたランタイムソースコード[*13] を調べてすぐに判明した。

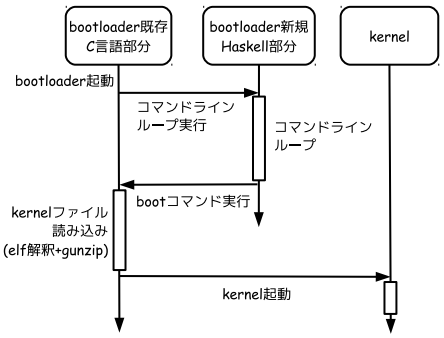


図 23.6: bootloader の一部を Haskell で再設計

jhc が生成する C 言語コードの main 関数は以下のような動作をする。

1. hs_init 関数 call
2. setjmp の設定
3. _amain 関数 call
4. hs_exit 関数 call

つまり GHC と同じように hs_init 関数を呼び出した後、_amain 関数という jhc 独自の関数を呼び出せば良いようだ。setjmp の設定は、例外を扱うためだけだ。例外が起きないように Haskell コードを書く上ではこれは不要と思っていい。_amain 関数は jhc が Haskell コードをコンパイルした結果の C 言語ソースコード内にある。

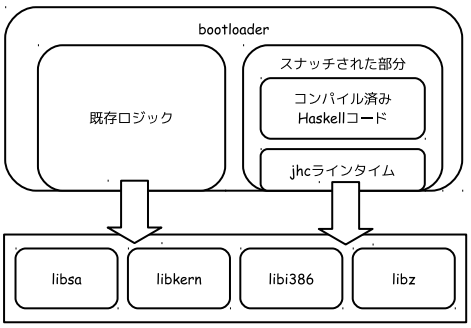


図 23.7: bootloader で使えるライブラリ

[*13] https://github.com/ajhc/ajhc/blob/v0.8.0.1/rts/rts/jhc_rts.c#L164

```
// File: rts/main_code.c (ミューテータ)
void
_amain(void)
{
        return (void)b__main(saved_gc);
}

static void A_STD
b__main(gc_t gc)
{
        return ftheMain(gc);
}
```

ftheMain 関数というのが Haskell の Main.main 関数に相当するようだが、_amain 関数はそのラッパーになっているようだ。

これで材料を集めるのは終わっただろう。設計 [*14] に入ろう。まず既存の NetBSD bootloader のソースコードを観察すると、boot2 という関数がメイン関数のようだった。

```
/* file: sys/arch/i386/stand/boot/boot2.c */
void
boot2(int biosdev, uint64_t biossector)
{
/* --snip-- */
    print_banner();
#endif

    printf("Press return to boot now, any other key for boot menu\n");
```

ここで注意しなければならないことがある。bootloader の中では簡単に printf 関数が使えない。POSIX 上で動作するプログラムと異なり、bootloader の下には libc も kernel もいない。bootloader の中で printf 関数を使うには、ビデオやシリアルの初期化をしなければならない。

この boot2 関数の中ではじめて printf 関数を使う箇所が上のソースコードだった。ということはこの printf 関数の直前までの間にコンソールの初期化は終わっているはずで、当該箇所で Haskell コードをそのまま実行すれば文字を印字できるはずだ。おそらくキー入力も受け付け可能だと思われる。そこで以下のように printf 関数直前に Haskell コードの呼び出しを追記した。

```
/* file: sys/arch/i386/stand/boot/boot2_ara.c */
    print_banner();
#endif
    { /* Run Haskell code */
        int hsargc = 1;
        char *hsargv = "nbboot";
        char **hsargvp = &hsargv;

        hs_init(&hsargc, &hsargvp);
```

[*14] 最初のスケッチのソースコード https://gitorious.org/metasepi/netbsd-arafura

```
        if (jhc_setjmp(&jhc_uncaught)) {
            jhc_error("Uncaught Exception");
        } else {
            _amain();
        }
        /* hs_exit(); */
    }
    printf("Press return to boot now, any other key for boot menu\n");
```

これで Haskell コードが bootloader から呼び出されるはずだ。次の課題は hs_init 関数と _amain 関数の先にある jhc ランタイムが依存している関数の実装ということになる。ほとんどは単純なスタブ関数を実装するだけだが、唯一面倒だったのがメモリ管理まわりだった。jhc ランタイムは malloc 関数を要求していた。幸いにも NetBSD bootloader が内包するライブラリが alloc という関数を持っている。これを使えば malloc の API を実装することは可能だ。

これで C 言語側の設計は終わりだ。コマンドラインループを Haskell で記述しよう。 NetBSD bootloader の C 言語での設計を見るかぎり、以下のような Haskell コードと等価のようだった。

```
-- file: metasepi-arafura/sys/arch/i386/stand/boot/Boot2Ara.hs
import Control.Monad
import Data.Maybe
import Data.Map (Map)
import qualified Data.Map as Map
import Foreign.C.Types
import Foreign.Ptr

foreign import ccall "glue_netbsdstand.h command_boot" c_boot :: Ptr a -> IO ()

commands :: Map String (IO ())
commands = Map.fromList [("help", command_help),
                         ("boot", c_boot nullPtr)]

command_help :: IO ()
command_help = putStr $ "\
\commands are:\n\
\boot [xdNx:][filename] [-12acdqsvxz]\n\
\help\n"

main :: IO ()
main = do
  putStrLn "Haskell bootmenu"
  forever $ do
    putStr "> "
    s <- getLine
    fromMaybe (putStr s) $ Map.lookup s commands
```

上記の Haskell コードはコマンドラインから “boot” という入力を検出すると C 言語で書かれた command_boot 関数を呼び出す。この command_boot 関数が kernel をファイルシステムから取り出し起動する。つまりこれでコマンドラインループのみを Haskell コードでスナッチできたわけだ。さて qemu で bootloader

を起動してみよう……あれ？ Haskell コードが実行されない。動くはずなのに……

デバッグの結果どうやら malloc が 1MB のメモリを確保しようとしているために、NetBSD bootloader の alloc ヒープが狭すぎて abort しているようだった。bootloader はコンベンショナルメモリ内の 640kB で動作するため 1MB もの大きな alloc ヒープは確保できない。別の方法としては 1MB 以降の潤沢な拡張メモリを alloc ヒープとして使う手もあるが、今度は BIOS から読み書きすることができなくなってしまう [*15]。しかもリソースの制御は設計において重要な課題の一つだ。実際の製品ではリソースは無限にあるわけではなく制約がつくのが通常だ。たかだかコマンドラインループを実現するために最低 1MB ものヒープが必要になるのではこの先 kernel を書く上で致命的な欠陥を生じかねない。ここは踏み止まって省メモリ化を検討すべきだ。そこでぼくは jhc ランタイム中での malloc の使用箇所を洗い出してみることにした。

```
$ grep -n malloc rts/rts/gc_jgc.c
193:    saved_gc = gc_stack_base = malloc((1UL << 18)*sizeof(gc_stack_base[0]));
298:    base = _aligned_malloc(MEGABLOCK_SIZE, BLOCK_SIZE);
320:    struct s_megablock *mb = malloc(sizeof(*mb));
512:    struct s_cache *sc = malloc(sizeof(*sc));
588:    struct s_arena *arena = malloc(sizeof(struct s_arena));
618:gc_malloc_foreignptr(unsigned alignment, unsigned size, bool finalizer) {
```

193 行目と 298 行目がどちらも 1MByte ものメモリを確保している。このサイズを単に小さくしてしまえば逃げられないのだろうか？

- gc_stack_base: 1MByte → 128kByte
- MEGABLOCK_SIZE: 1MByte → 64kByte
- BLOCK_SIZE: 4kByte → 256Byte

再コンパイル……実行……動いた! だましだましコンベンショナルメモリに押し込んだが、jhc は POSIX レイヤーがなくとも動作可能なバイナリを吐き出せることを実証できた。またスナッチモデルが少なくとも割り込み禁止状態で動作するソフトウェアに対しては有効であることもわかった。

ぼくは意気揚々としてみんなにこの成果を報せた。あの苦しみを経験したぼくらならこの jhc コンパイラを製品化可能なレベルにまで、みんなで成長させることができるのではないかと思ったのだ。ところがみんなの反応はぼんやりしていた。

「bootloader を書き換えても何の証左にもなっていないと思う」
「デバイスドライバ書くならメモリマップド IO をさわらなきゃだけど Haskell だと無理じゃない？」
「そもそも C 言語で記述された kernel と同等の表現を得られるのかな？」
「もし製品化するならコンパイラの中身を弄らなきゃならない。素人には無理だよ」
みんなから湧き出る不安と疑問。

ぼくは意気消沈した。

23.5 冷たい壁

海が見える。この国では外洋に接する海岸が多く、しかも地震が多い。例年のように津波による被害が出る [*16]。いつものおだやかな海岸も時に本来の巨大な力を剥き出しにする。

技術探索も同じだ。一見うまく出荷できそうな基本設計も詳細な実装をする段階になって根本的な欠陥が見つかることも多い。出荷できない技術など存在しないも同然だ。人はみな視界を持っている。どこまで見通せるかは人それぞれだが、その範囲はたかだか有限なのだ。ぼくの道は光に通じるのか、それともやはり冷たい高い壁が待ち構えるのみなのか……

「モチベーションを複製することは困難でゲソ」
いつの間にかあの娘が隣にいた。

[*15] リアルモードでも拡張メモリをアクセスする方法は存在するらしいです。残念ながら筆者は具体的な手順を知りません……
[*16] http://ja.wikipedia.org/wiki/スマトラ島沖地震

「おぬしが出会う問題と他人が出会う問題は同じようでも微妙に異なるじゃなイカ。それらが蓄積されると問題の集合同士は大きく異なることになるでゲソ」
なにを当然のことを言ってるんだよ……
「モチベーションは問題意識から生まれることが多いでゲソ。するとモチベーションを他人の中に完全に複製するためには、おぬしがこれまで出会ってきた問題を全部説明しなければならないことになるんじゃなイカ？」
そんなことは有限時間では不可能だ。
「いまは初速が必要なのでゲソ」
つまり？
「モチベーションを共有していなくてもプロジェクトに参加したくなる、そんな魅力的な機能と性能と品質とそして明確な用途を示す必要があるのでゲソ。そして究極的にはその魔法の瞬間まではおぬし一人で辿りつくしかないんじゃなイカ？」
しかしぼくの中にあるエネルギーは有限だ。その初速を得るに足りない場合はどうすればいいんだ……
「その時は」
風がきこえる。
「未到達ということになるんじゃなイカ。おぬしはいつも自分のことしか見ていないようでゲソ。いま足場にしている jhc コンパイラという一つの到達点に誰がいったいどうやって辿りついたのか、そのことを考えてみるべき時じゃなイカ？」
…
…
… …

決めた。jhc コンパイラをぼく一人の手で成長させる。この純朴なコンパイラが秘めた光をはなつまでは。jhc コンパイラの作者である **J** [*17] に何度か patch を送ってみたが、merge はしてくれどもあまり活発な応答はなかった。おそらくとてつもなく忙しいのかもしれない。もしかしたら……いや、どんな天才でも気力が底をつくことだってある。全てのリソースは有限なのだ。

jhc のランタイムとコンパイルパイプラインの最終段については理解が進んでいたので、修正のアイデアがいくつもあった。そこでぼくは jhc をフォークして **Ajhc** [*18] というプロジェクトを立ち上げることにした。オープンソースプロジェクトにおいてプロジェクトのフォークはあまり良いイメージがない。しかしリポジトリがない状況でこれ以上 jhc の機能開発をすることは困難だった。もちろんぼくが実装する機能の宣伝をする場所が欲しかったこともある。

J には悪いと思ったが、正直に何が問題であるのかをまとめてメールを送った。[*19] さぁ準備は整った。あとは進むだけだ。ぼくのリソースのあるかぎり。

23.6 二度目のスケッチ: 省メモリ化の追求

プロジェクトを起こしたからにはわかりやすい応用例があった方がいい。最初のスケッチでは bootloader をスナッチしたが、今ふりかえると見た目が地味だったと思う。もっとわかりやすいアピールが必要だ。最近 Maker ムーブメントの影響でマイコンプログラミングが流行っている。Arduino のような C 言語ではない言語を使う開発環境もある。jhc を使ったマイコン開発という切り口はわかりやすいのではないか？

23.6.1 ハードウェア選定

bootloader のスナッチでは alloc ヒープを 320kByte 確保していた。この alloc ヒープは C 言語の malloc と Haskell ヒープの両方を管理している。kernel のファイル読み込みで alloc ヒープは酷使されるので、マイコンでは Haskell ヒープのみを使い、さらに jhc の GC をよく観察すれば、ヒープ全体の容量は桁を一つ落とすことが不可能でないかもしれない。つまり数十 kByte のメモリ (RAM) で動作する Haskell コードだ。やるからには限界に挑戦したくなっていた。

そんなマイコンを探してみよう。この国には秋葉原のような便利な場所がない。それでもインターネット

[*17] John Meacham `http://repetae.net/`
[*18] Ajhc - Haskell everywhere `http://ajhc.metasepi.org/`
[*19] ANNOUNCE: Start Ajhc project with forking jhc. `http://www.haskell.org/pipermail/jhc/2013-March/001007.html`

はやはり便利でネット通販で開発ボードを買える。今回のスケッチにぴったりのボードが見つかった。

- ボード名: STM32F3DISCOVERY http://www.st.com/stm32f3discovery
- CPU: ARM Cortex-M4
- ROM: 256kByte
- RAM: 48kByte
- 備考: ST-LINK/V2 チップ付属 - gdb によるデバッグが可能 [*20]
- 価格: 950 円

すごい時代だ。評価ボードがこんな価格で入手できる。さっそく注文した。

手元に届くまでの間に開発環境を整えよう。STM32F3-Discovery Application Template [*21] に STM32F3DISCOVERY 用の LED チカチカのソースコード一式が落ちていた。ありがたくスケッチの足場として使わせてもらうことにした [*22]。

数日の後ボードが届いたので、クロス gdb で C 言語のデモを動かしてみたところ問題なく動作した[*23]。これでスタート地点に立てたことになる。

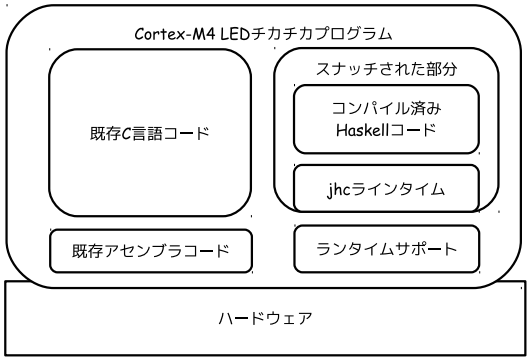


図 23.8: Corntex-M4 上のプログラムのスナッチ

23.6.2 作戦

さっき見つけた C 言語で書かれている LED チカチカデモをスナッチしてみることにした。

おそらくこのスケッチの全体像は図 23.8 のようになるだろう。このデモは C 言語とアセンブラでできているはずだった。これを Haskell コードを使ってスナッチする。この時、jhc ランタイムが依存するプリミティブが足りないこともあるかもしれない。その場合は適宜ランタイムサポートのためのコードを書くしかない。このランタイムサポートのコードを小さく抑えれば、jhc ランタイムの依存する機能プリミティブが見えてくるはずだ。

ぼくはこのスナッチのゴールについて考えてみた。アセンブラのコードはおそらく C 言語でさえ記述できないしろものなので、置換することは不可能だ。ということはランタイムサポートと既存 C 言語コードの最小値を得た地点がゴールということになる。

さて、先の bootloader のスナッチではあまりちゃんとしたビルドプロセスを作らなかった。まだ jhc についての理解が浅かったためだ。今回は x86 の開発マシンの上で ARM のバイナリをコンパイルするため、しっかりとしたクロスコンパイル環境を整えた方がいい。また、前回のスナッチでは jhc のランタイムソースコードを開発対象のディレクトリに含めてしまっていた。これは開発対象のソースコードが jhc の特定バージョンに紐づいてしまう原因になる。

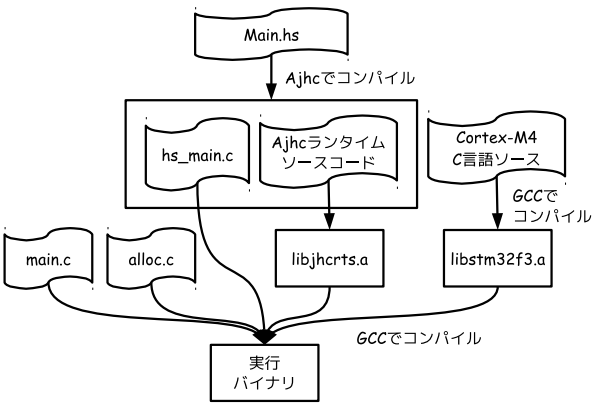


図 23.9: Corntex-M4 クロスコンパイル環境

[*20] https://github.com/texane/stlink
[*21] https://github.com/mblythe86/stm32f3-discovery-basic-template
[*22] 二度目のスケッチのソースコード https://github.com/ajhc/demo-cortex-m3
[*23] http://www.slideshare.net/master_q/20130629-deb-cortexm3

そこで図 23.9 のようなクロスコンパイル環境を作った。開発環境に Ajhc ランタイムソースコードを保持するのではなく、Haskell ソースコードをコンパイルする際に生成される Ajhc ランタイムソースコードをライブラリ化してリンクするようにした。これでコンパイル時に使用しているバージョンの Ajhc に対応するランタイムを使うことができる。もし Ajhc ランタイムに修正を入れたくなったら Ajhc 本体に修正を加えなければならない。libstm32f3.a は先の Template に付属していたユーティリティが入っている。また Template には malloc API がなかったので、bootloader の alloc コードを移植した。

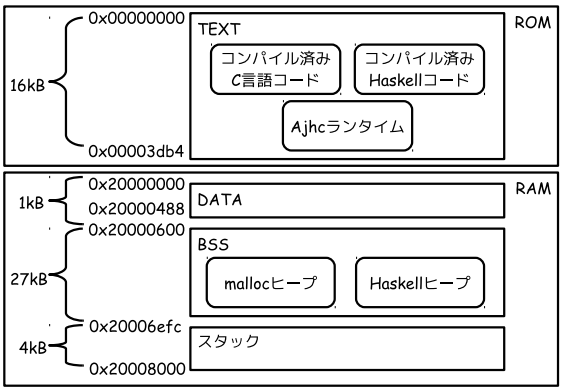


図 23.10: Cortex-M4 上で動作する実行バイナリのメモリマップ例

いろいろな要素が出てきて混乱してきた。クロスコンパイラによって生成された実行バイナリの中身を一度整理してみることにした。図 23.10 は実行バイナリのメモリマップだ。

TEXT 領域には手書きした C 言語コードと Ajhc が吐き出した C 言語コードがまざっている。Haskell で動的に確保するサンクは Haskell ヒープの中に配置される。Haskell コードがこの領域より多くのサンクを同時に確保しようとした場合には abort になる。さらに Haskell ヒープの管理を目的として malloc ヒープが用意されている。この malloc ヒープがあふれた場合も即 abort だ。Ajhc の吐き出すコードは再帰の実行のためにスタックを多く消費することがある。スタックは若いアドレスに向かって成長するため、このメモリマップだとスタックが BSS を破壊することがある。製品化の際にはスタック領域を RAM の先頭に配置した方が良いかもしれない。

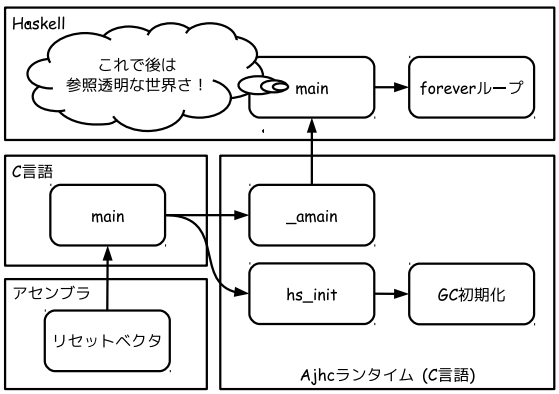


図 23.11: Cortex-M4 上での Haskell コードの起動

クロスコンパイラによって生成された実行バイナリは次のように起動する (図 23.11)。

1. CPU はリセットベクタから実行を開始する
2. リセットベクタは C 言語の main 関数を呼び出す
3. C 言語の main 関数はハードウェアの初期化を行なう
4. C 言語の main 関数は Ajhc ランタイムの hs_init を呼び出す
5. hs_init 関数は Ajhc ランタイムの初期化を行なう
6. C 言語の main 関数は_amain 関数を呼び出し Haskell コードが走る

さぁ Haskell コードを書いてみよう。マイコンの Hello World といえば LED チカチカだろう。このボードには LED が 8 つ搭載されているが、その内 LED3 だけを blink させてみた。

```
import Data.Word
import Control.Monad
import Foreign.Ptr
import Foreign.Storable
```

```
foreign import ccall "c_extern.h Delay" c_delay :: Word32 -> IO ()
foreign import ccall "c_extern.h &jhc_zeroAddress" c_zeroAddress16 :: Ptr Word16

gpioPin9, led3 :: Word16
gpioPin9 = 0x0200
led3     = gpioPin9

brrPtr, bsrrPtr :: Ptr Word16
brrPtr  = c_zeroAddress16 `plusPtr` 0x48001028
bsrrPtr = c_zeroAddress16 `plusPtr` 0x48001018

ledOff, ledOn :: Word16 -> IO ()
ledOff = poke brrPtr
ledOn  = poke bsrrPtr

main :: IO ()
main = forever $ do
  ledOn led3
  c_delay 10
  ledOff led3
  c_delay 10
```

この Haskell コードの動作をおおざっぱに図示してみる (図 23.12)。Haskell は実は Ptr 型を通して生のメモリへ直接読み書きを行なうことができる。書き込み関数が poke 、読み込み関数が peek だ。このボードの場合 0x48001028 アドレスに 0x200 を書くと LED3 が点灯し、0x48001018 アドレスに同じく 0x200 を書くことで LED3 が消灯する。c_delay 関数は C 言語の Delay 関数を呼び出して、時間待ち合わせを行なう。

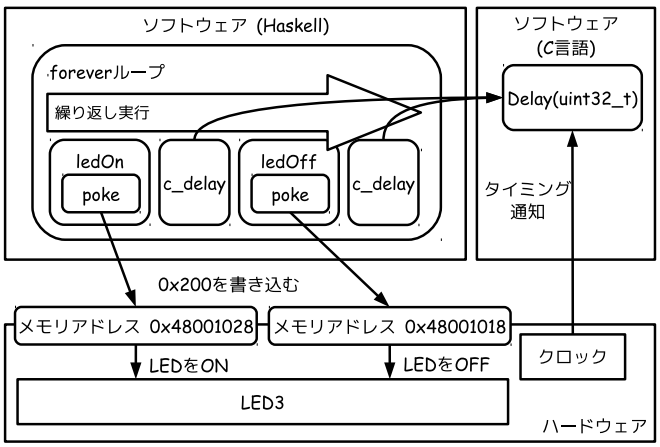


図 23.12: LED チカチカプログラムの動作

ここで jhc_zeroAddress という C 言語側で定義されたグローバルポインタ経由で Ptr 型を作っていることに疑問を持つかもしれない。これには理由がある。仮に nullPtr [*24] を使って通常の Haskell API のみで実装してみる。

```
--- a/stm32f3-discovery/hs_src/MainBlinkLedSimple.hs
+++ b/stm32f3-discovery/hs_src/MainBlinkLedSimple.hs
@@ -11,8 +11,8 @@ gpioPin9 = 0x0200
 led3     = gpioPin9
```

[*24] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Foreign-Ptr.html#v:nullPtr

```
 brrPtr, bsrrPtr :: Ptr Word16
-brrPtr  = c_zeroAddress16 `plusPtr` 0x48001028
-bsrrPtr = c_zeroAddress16 `plusPtr` 0x48001018
+brrPtr  = nullPtr `plusPtr` 0x48001028
+bsrrPtr = nullPtr `plusPtr` 0x48001018

 ledOff, ledOn :: Word16 -> IO ()
 ledOff = poke brrPtr
```

この Haskell コードを Ajhc でコンパイルすると Haskell の main 関数は以下のような C 言語ソースに変換されることになる。

```
static void A_STD
ftheMain(gc_t gc)
{
        fR$__fControl_Monad_forever__2:;
        {   *((uint16_t *)(1207963672)) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            *((uint16_t *)(1207963688)) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            goto fR$__fControl_Monad_forever__2;
        }
        return;
}
```

一見 0x48001028 アドレスに 0x200 を正常に書き込んでいるように見える。しかし上記の C 言語ソースコードをよく見ると、ポインタアクセスに volatile が付いていない。これでは GCC 側では最適化で消されてしまうことになる。副作用と認められないのだ。

```
static void A_STD
ftheMain(gc_t gc)
{
        fR$__fControl_Monad_forever__2:;
        {   *((uint16_t *)(1207963672 + ((uintptr_t)&jhc_zeroAddress))) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            *((uint16_t *)(1207963688 + ((uintptr_t)&jhc_zeroAddress))) = 512;
            saved_gc = gc;
            (void)Delay((uint32_t)10);
            goto fR$__fControl_Monad_forever__2;
        }
        return;
}
```

元の jhc_zeroAddress を使った Haskell コードを C 言語に変換すると上記のような jhc_zeroAddress グローバルポインタに書き込みアドレスを加えたポインタに 0x200 を書き込むコードが生成される。Haskell 側で

Ptr 型に volatile を付けるプラグマがあればよかったのだが、その方法はないようだった。

jhc_zeroAddress は 0 アドレスへのポインタとして C 言語側で宣言されている。そのためこの方式であれば C 言語コンパイラは愚直にメモリ書き込みを行なうアセンブラを吐くようになる。どうやら Ajhc を使ったハードウェア制御コードではこのイディオムがセオリー[*25] になりそうだった。

23.6.3 デバッグ

さて作成したコードを実行してみた。なんとあっさり LED がチカチカ光るじゃないか。型の力はすばらしい。ポインタアクセスをしている界面のみ気をつけて設計すれば残りは型に守られた設計ができるのだ。

ところが先の簡単な forever ループを拡張してちょっと純粋な関数を加えるとコードが動かない。これは何が起きているのだろうか？ gdb で追ってみると例外ベクタに飛んでいる。なぜ例外が起きるのか？ もっと丁寧に追うとコードのかなり深くで例外が起きる。ヒープの制御がおかしいのだろうか。スタックがあふれたのだろうか。

数日の間このことが頭をはなれず、アルバイトでもぼーっとしていた。

「なにを悩んでるの？」

ふりむくと同僚がいた。彼はプログラミング言語にとても詳しく、なにかあるとサポートしてもらっていた。ぼくは思い切って Ajhc の計画について打ち明けてみた。

「うーん、その不具合の原因はよくわからないけれど、その計画は明らかに GC が鍵になるね」

jhc の GC は C 言語のみで記述されていて、行数も 800 行程度と短かい。しかしその実装はトリッキーでこれまで理解することを避けてきていた。それが今のデバッグの困難に結びついていたのだ。

「ちょっとコードを読んできてあげるよ。なにかわかったらメールするね」

数日後、ぼくは彼から jhc の GC (**jgc** と呼ばれる) のしくみを教えてもらった[*26]。とてもわかりやすい資料だった。ぼくはこれを元にして自分なりに jgc のコードを読んでみた。

まず jgc は Haskell ヒープ全体を arena グローバル変数で管理する (図 23.13)。次に Haskell のサンクなどのデータを格納するための器がある。その最も大きい単位が megablock だ。Haskell ヒープの容量が不足するたびに megablock は追加で確保される。megablock の中には固定サイズの block という構造がある。この block をさらに固定サイズのチャンクに分割して、そのチャンクに Haskell で使用するデータを格納する。

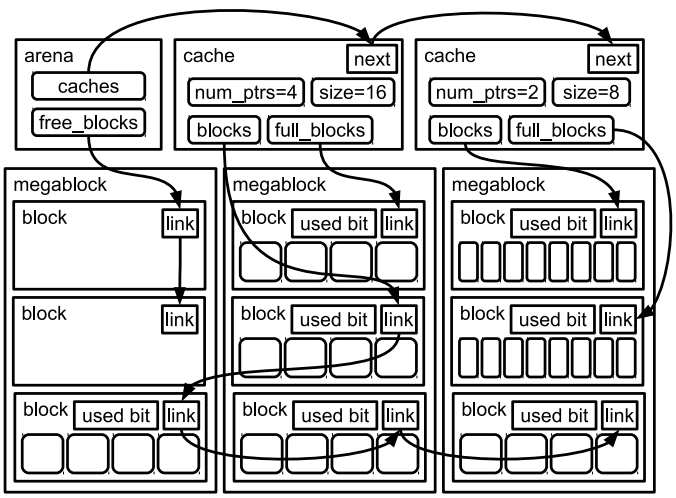


図 23.13: jgc での Haskell ヒープの管理

block には used bit の配列があり、使用中[*27]のチャンクに対応するビットには 1 を。未使用のチャンクに対応するビットには 0 を書くのが決まりだ。さらに block には link というメンバーがあり、当該の block が Haskell ヒープ全体から見てどこに繋っているのかリスト管理できるようになっている。例えば、s_arena 構造体には free_blocks というメンバーがあり、このメンバーに繋がっている block は内包する全てのチャンクが未使用であることを表わす。

[*25] https://github.com/ajhc/demo-cortex-m3#write-haskell-code

[*26] 小二病でも GC やりたい http://www.slideshare.net/dec9ue/gc-16298437

[*27] ここでは GC ルートから辿れることを使用中を呼んでいる

ここから少し見慣れない構造がでてくる。**cache** だ。s_arena 構造体の caches メンバーには cache という構造がぶらさがっている。この cache の先には block がぶらさがっているのだが、その block の中のチャンクのデータ構造を cache によって定義している (図 23.14)。すなわち cache の num_ptrs メンバーはチャンクの中のポインタの個数を表わし、size メンバーはチャンクの全体サイズを表わす。

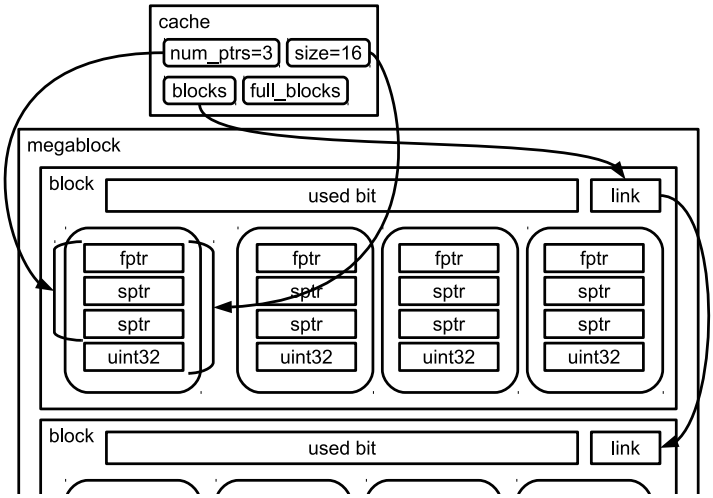


図 23.14: jgc でのチャンクの構造は cache で定義される

この時チャンク内のポインタは先頭から配置されるのが約束になっている。ポインタはスマートポインタになっていて格納される値は即値である可能性もある[*28]。この cache のリストは Ajhc ランタイムの初期化の中で find_cache 関数によって動的に確保される。

このチャンクは Haskell 由来のデータを格納しているが当然 GC する必要がある。jgc の GC はミューテータがコンストラクタによって Haskell ヒープから新しいメモリ領域を確保しようとする時にのみ呼び出される。また GC はマーキングだけを行ないスイープをしない。というのも cache の blocks メンバーの used bit を検索すれば未使用チャンクが見つかるし、仮にそこで見つからなかった場合には s_arena 構造体の free_blocks から新しい block を補充するれば良いからだ。jgc の GC は以下のような手順を取る (図 23.15)。

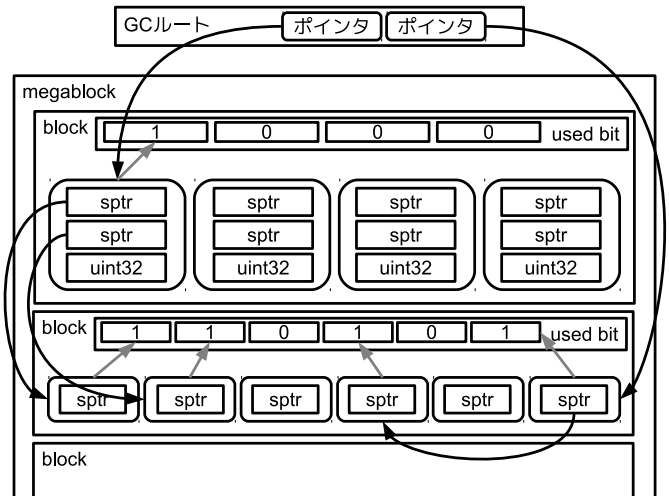


図 23.15: jgc の GC マーキング

1. Haskell ヒープのアロケート関数 s_alloc がミューテータによって呼び出される
2. s_alloc 関数は要求されたコンストラクタに対応する未使用チャンクを探す
3. 2 で見つからなかった場合 megablock を新たに確保すべきか GC すべきかを判定する
4. 3 で GC すべきと判断した場合 gc_perform_gc 関数を呼び出す
5. gc_perform_gc 関数は Haskell ヒープ全ての used bit を 0 クリアする
6. gc_perform_gc 関数は GC ルートから参照を辿り、当該チャンクに対応した used bit に 1 を立てる
7. 2 と同じように未使用チャンクを探してミューテータに渡す

[*28] http://ajhc.metasepi.org/manual_ja.html#ランタイムシステム

スマートポインタにはいくつか種類があり、fptr_t もその一種だ。この fptr_t には最初はクロージャの評価関数へのポインタが格納されている。つまり未評価サンクだ (図 23.16)。このサンクを評価するにはサンクのスマートポインタを eval の第二引数に渡す。すると eval はサンクの遅延ビットをチェックし、未評価サンクだと判定すれば fptr_t の先にあるクロージャ評価関数に node_t を渡す。クロージャの評価が終わったら update 関数を呼び出してチャンクに入っている fptr_t をクロージャの評価結果で上書きする。この時 fptr_t の遅延ビットは 0 にする。これで評価済みサンクのできあがりだ。

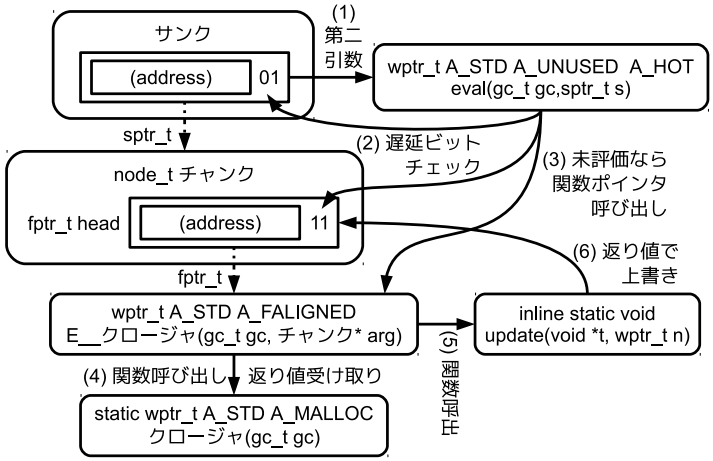


図 23.16: サンク評価の一例

かなり GC のコードを読み込んだおかげで Ajhc のランタイムの動作が想像できるようになった気がする。やはり同僚のアドバイスは正しかったようだ。ありがたいことだった。

ぼくはデバッグを再開した。やはり例外が起きている。どうも単純な IO のみの forever ループを逸脱すると Haskell ヒープを使うようになるようだ。この時サンクの評価のために関数ポインタへのジャンプ命令が実行されるのだが、その近辺で例外ベクターに飛んでしまう。関数ポインタの先も text 領域の範囲内だ。これはソフトウェアの問題ではなく ARM プラットフォーム側の問題ではないか？

ぼくは objdump で eval 関数のアセンブラを見てみた。関数ポインタジャンプは ARM では blx という命令のようだ。なんとなく嫌な予感がする。 ARM の命令リファレンスから blx 命令の章[*29] を見てみよう。

```
The BLX instruction can change the instruction set.
BLX label always changes the instruction set. It changes a processor in ARM
state to Thumb state, or a processor in Thumb state to ARM state.
BLX Rm derives the target instruction set from bit[0] of Rm:
* if bit[0] of Rm is 0, the processor changes to, or remains in, ARM state
* if bit[0] of Rm is 1, the processor changes to, or remains in, Thumb state.
```

な ん だ と ？

Cortex-M4 は ARM state はサポートせず Thumb state のみしか使えないはずだ。ということは関数ポインタアクセスの場合は 0 ビット目に必ず 1 を立てないと例外が起きるということなのか？

なんというアーキティクチャだ……信じられない。

[*29] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489i/CIHBJCDC.html

まだ納得できないので GCC が良きにはからってくれないか Web 検索してみた。ところがどうも最新の GCC でもこの問題には決着がついていない [*30] ようだった。さすがにこれは Ajhc のランタイム側に修正を加えるしかない。ぼくは "_JHC_ARM_STAY_IN_THUMB_MODE" というコンパイルフラグ[*31] を用意することにした。ソースコード [*32] [*33] を読めばわかる通りだが、このフラグを有効にすると関数ポインタを使う前に 0 ビット目を立てる。

やっと Haskell ヒープを使うプログラムが動作するようになった。ぼくは安堵して夕食を取ることにした。少しビンタンも飲もう。ところが机に戻ってみるとさっき実行させたままだったデモアプリケーションが停止している。おかしい……abort で停止している。gdb でバックトレースを取ってみるとどうやら s_alloc でメモリ領域を確保しようとして失敗しているようだった。メモリリークをしているのか？

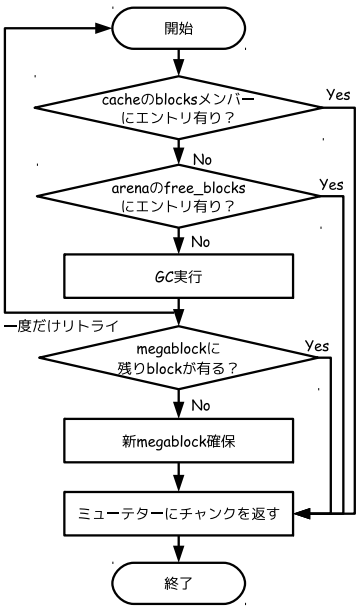


図 23.17: s_alloc 関数フローチャートと GC 後のリトライ追加

もう一度 s_alloc のソースコードを良く読んでみることにした。すると原因が判明した。jgc は megablock と block の確保を富豪的に楽観的に行なう。図 23.17 のフローチャートは s_alloc 関数の実行フローだ。よく見ると GC を実行した後に megablock を確保してしまうことがわかる。このままではヒープが足りているにもかかわらず、不要な megablock を投機的に確保してしまう。メモリが豊富にあるのであれば、時間効率をかせぐために投機的な megablock 確保は有効かもしれない。しかし数十 kB しかメモリがないマイコンではこの富豪的な megablock 確保方式は致命傷だ。

これもまた Ajhc ランタイムを修正する他にない。"_JHC_JGC_NAIVEGC" というコンパイルフラグを作った。このフラグを有効にすると GC 実行の後、一度だけ s_alloc 関数を最初からリトライする。GC をしているということは blocks もしくは free_blocks にエントリが追加されたかもしれないからだ。

これでやっとデモプログラムが安定動作するようになった。ちょっと凝ったプログラムを作ろう。任意の文字列を LED でモールス信号表示するプログラムだ。まずモールス信号の型を決める。

```
data Morse = S | L | LetterSpace | WordSpace
```

S がトン、L がツーだ。制御コードは長いので、ここではその構造を図にしてみることにする (図 23.18)。containers ライブラリの Map を使って morseCode という文字→モールス信号列の変換表を作る。さらにモールス信号を LED の blink アクション列に変換する morseToIO 関数を作る。この二つを組み合わせることで任意の文字列を LED の blink アクション列に変換する morseEncodeIO 関数を作ることができる。最後にまた forever ループに食わせれば任意の文字列を繰り返しモールス信号表示するプログラムが完成する。

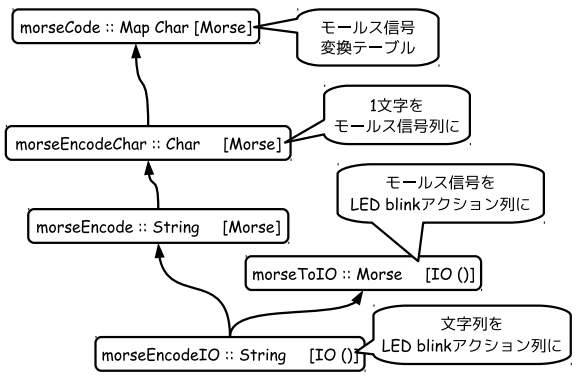


図 23.18: モールス信号 LED 表示デモの関数

[*30] http://communities.mentor.com/community/cs/archives/arm-gnu/msg01904.html2
[*31] http://ajhc.metasepi.org/manual_ja.html#cflags で指定できる特殊な define
[*32] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/jhc_rts.h#L99
[*33] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/jhc_rts.c#L147

ぼくはこのモールス信号プログラムを作っている時、一度もデバッガを使わなかった。また Linux 上でのテストもしなかった。ただ書き下してコンパイルが通ればその通りに動いた。型安全の威力はどのドメインにおいても有効なのだと思い知った。

このデモをみんなに見せてみた。こんどはみんなびっくりしてくれた。しかしツテを頼っても Ajhc を収益化する見込みは得られなかった。アルバイトをしながら Ajhc を開発していてもなかなか先に進めない。どうやらもう一頑張り必要なようだった。

23.7 あこがれ

ぬるくなったビンタンの瓶が見える。最近は暇されあれば jhc のソースコードを読んでいた。jhc のソースコードはおせいじにもキレイとは言いがたかった。それでも時間をかければ全体像が見えてきた。“An informal graph of the internal code motion in jhc” *34 にも解説があるが、もっと関数レベルの関係図を頭に入れておきたい。描いてみると図 23.19 ができあがった。

まだ少しイメージがつかみにくいので型の変換に注目してみよう。jhc のコンパイルパイプラインは Haskell ソース文字列を以下の型を通して C 言語ソース文字列に変換する。この型の変換そのものがコンパイルという行為だというわけだ。

1. [FilePath] 型 (Haskell ソースコードのファイルパス)
2. CollectedHo 型∋ IdMap Comb 型∋ E 型 (コンパイル対象単体での最適化)
3. Program 型∋ Comb 型∋ E 型 (プログラム全体の最適化)
4. Grin 型∋ [FuncDef] 型∋ Lam 型 (λ抽象での最適化)
5. ByteString 型 (C 言語ソースコード)

Haskell コードは jhc コンパイルパイプラインの中で大きく二つの型で表現されているようだった。E 型と Grin 型だ。E 型は二つの最適化フェーズで使われる。一つ目はコンパイル対象単体での最適化だ。実行プログラムをコンパイルする際にもこの最適化が行なわれるが、より注目すべきは hl ファイルだ。この hl ファイルは jhc におけるコンパイル済み Haskell ライブラリのフォーマットで、第一段階の E 型最適化が完了した時点で凍結された Haskell コードが格納されている。二つ目はプログラム全体での最適化だ。この第二段階の E 型最適化ではコンパイル対象と hl ファイルを一緒にして最適化する。そのため hl ファイルの中でも未到達な関数はこの第二段階で削除されることになる。Grin 型は C 言語ソースコードに変換される直前の変形に用いられる。

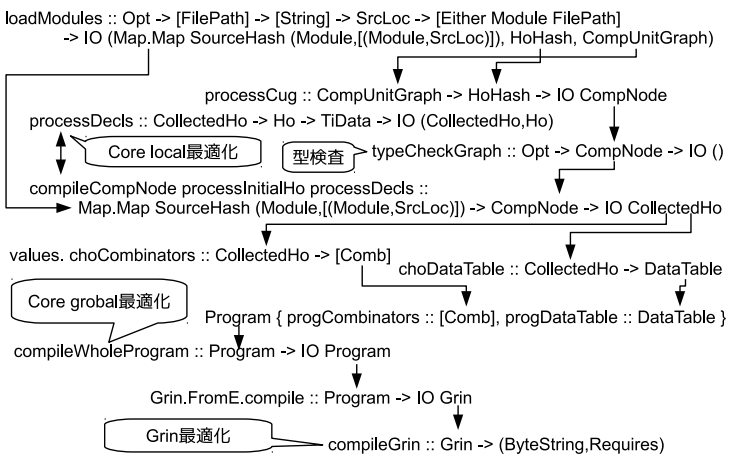


図 23.19: jhc コンパイルパイプラインの全体像

他にまとまったドキュメントには “Jhc User’s Manual” *35 がある。このドキュメントは英語で、ぼくには読みにくかった。そこで英語が得意な友達 *36 に協力してもらって日本語訳 *37 をしてみた。このマニュアルを

*34 http://repetae.net/computer/jhc/big-picture.pdf
*35 Jhc User’s Manual(英語) http://repetae.net/computer/jhc/manual.html
*36 @dif_engine さんに手伝ってもらったでゲソ!
*37 Ajhc ユーザーズマニュアル (日本語) http://ajhc.metasepi.org/manual_ja.html

読むことで jhc の設計ポリシーがわかってきた。

- grin は未到達のコードを削除する
- hl ファイルは動的/静的リンクなどはできず、grin にかけてバイナリ化しないと実行できない
- Haskell のプリミティブ型は C 言語の型そのまま
- コンパイルパイプライン途中の AST をダンプできるオプションが多数ある
- m4 プリプロセッサのような変態的な機能
- スマートポインタのために Int 型は 30 ビット幅
- Integer 型はまっとうに実装されておらず IntMax 型と同じ精度しか保持できない
- pure type system という型システムを採用していて、これはラムダキューブに対応する

特に重要なアイデアがプリミティブ型についてだ。

```
-- File: ajhc/lib/jhc-prim/Jhc/Prim/Bits.hs
data Bits32_  :: #

-- File: ajhc/lib/jhc/Jhc/Type/Word.hs
data {-# CTYPE "int"      #-} Int = Int Bits32_
```

この CTYPE というのは型プラグマ [*38] で FFI の先である C 言語から見た型を指示する。つまり Haskell の Int 型は C 言語側から見ると C 言語の int 型に見えるということだ。Bits32_型は “#” という見慣れない型に落ちているが、これは “プリミティブ型である” という意味しか持っておらず、単に Haskell 側の型推論のつじつま合わせとして使われる。これらのプリミティブ型は当然アンボックス化された型で、この型をボックス化するのは Haskell で書かれたライブラリの責務だ。

この型定義を見て、ぼくは (+) 演算子の実装はどうなっているのか気になった。jhc の Haskell ライブラリ側では以下のような実装になっており、Int 同士の和は “Add” というプリミティブ型に落とされているようだった。この Add プリミティブ型は jhc コンパイラでの grin=>C への変換で単に C 言語の “+” 演算子に変換される。Int は完全にアンボックス化されて直接 C 言語コードに落ちることになるわけだ。

1. Haskell の (+) 演算子
2. ↓変換 — ajhc/lib/jhc/Jhc/Num.m4 (Num クラスの (+) 関数定義)
3. jhc の “Add” プリミティブ
4. ↓変換 — ajhc/src/Cmm/Op.hs (プリミティブの定義)
5. ↓変換 — ajhc/src/C/Generate.hs (オペレータの定義)
6. ↓変換 — ajhc/src/C/FromGrin2.hs (Grin=>C 変換)
7. C 言語 “+” 演算子

今度は型の変換を調べてみよう。例えば fromIntegral 関数を使った Int から Float への変換だ。

1. Haskell の fromIntegral :: Int -> Float 関数
2. ↓変換 — ajhc/lib/jhc/Jhc/Num.hs (fromIntegral の定義)
3. fromInteger . toInteger
4. ↓変換 — ajhc/lib/jhc/Jhc/Num.m4 (toInteger の定義)
5. fromInteger . I2I
6. ↓変換 — ajhc/lib/jhc/Jhc/Float.hs (fromInteger の定義)
7. I2F . I2I
8. ↓変換 — ajhc/src/Cmm/Op.hs (I2I のようなキャスト型定義)
9. ↓変換 — ajhc/src/C/Generate.hs (キャスト操作の定義)

[*38] http://ajhc.metasepi.org/manual_ja.html#型プラグマ

10. ↓変換 — ajhc/src/C/FromGrin2.hs (Grin=>C 変換)
11. C 言語の二回キャスト (float)(int32_t)

なにやらややこしいが “I2I” と “I2F” というプリミティブを合成したものが Int から Float への変換のようだ。この二つのプリミティブはやはり grin=>C への変換で解釈される。そうしてなんのことはない単に C 言語のキャストに変換されるだけだ。

もっと簡単に言ってしまえば、jhc において Haskell のデータ型は全て C 言語と紐づけられたプリミティブ型のみで構成されている。もちろん Haskell 関数の評価については C 言語とは別物だ。C 言語の演算子や関数が部分適用された場合には相応の変換をかける必要がある。しかしその評価順序さえ C 言語と等価なもの (C 言語への変換直前の grin コード) に落としてしまえば、後はプリミティブ型を C 言語型に写像すればそのまま C 言語コードができあがる。この一見不可能に思えるアイデアこそが jhc の秘密のようだった。jhc は Haskell コードを C 言語に変換することを最初から強く意図して設計されていたのだ。

jhc のソースコードを読めば読むほどぼくはコンパイラという世界の魅力にとりつかれた。こんなアイデアを思いつきそして具現化した J の才能にあこがれた。そしてもちろん嫉妬した。なぜこれほどの頭脳がオープンソースの世界で発揮されず、開発が停滞してしまっているのか。なぜ世界はこの偉業を引き継ごうとしないのか。焦りと失望が頭をめぐる。

「なにを考えているでゲソ？」

元気なあの声がする。

「コンパイラの開発仲間が増えないことに焦っているんじゃなイカ？」

そうだ。ぼく一人でできることは限られている。このままのスピードではいつになったら kernel をスナッチできるのか予想ができなかった。

「そもそも今の Ajhc コンパイラをはじめて見た者はどう感じると思うでゲソ？」

小さなマイコンにも適用できる組み込み用途の Haskell コンパイラ。

「じゃあ Haskell 言語をはじめて使ってみようと思った者が使うコンパイラはなんでゲソ？」

……GHC 以外にありえない……

「ということはこのままでは小規模組み込みのエンジニアでなおかつ型推論を持つ言語に興味のあるユーザーしか獲得できないでゲソ。人はいきなりコンパイラの開発者になったりしないものでゲソ。まずはそのコンパイラのユーザーになって、そしてコンパイラそのものに興味が出た者が開発者に昇格するのでゲソ。世の中の人間が、おぬしのように自分が使いもしないコンパイラをいきなりいじりはじめる変態ばかりだったら大変でゲソ」

「おぬしは今まで Ajhc に適したドメインにしかコンパイラの応用例を示してこなかったのでゲソ。それはたしかに有用だったでゲソ。しかしこれ以上に開発者を増やしたいのであれば Ajhc コンパイラの適用可能なドメインを広げる必要があるんじゃなイカ？ これまではプロトタイプに必要な機能しか Ajhc コンパイラに追加してこなかったじゃなイカ。つまり曳光弾を使うべきときが来たのでゲソ」

しかし、そもそも Ajhc に欠けている機能はあまりにも多い。まだ通っていない退行テストがたくさんある。GHC とプリミティブ型が違うため多くのライブラリはそのままでは移植できない。Template Haskell も使えない。どこから手をつければいいんだ？

「とぼけても無駄でゲソ! Ajhc に決定的に欠けている機能が、おぬしにはすでに解っているじゃなイカ。もちろんその機能は kernel を設計する際にも必須でゲソ。実装できないのであれば、Ajhc の先には kernel をスナッチ可能なコンパイラとしての未来はない、ということになるんじゃなイカ？」

23.8 三度目のスケッチ: コンテキストを操る

そう、Ajhc の決定的な欠陥。それは **コンテキスト**が一つしか持てないことだ。これは最初のスケッチの時からわかっていたことだ。このままではスレッドを実現できないし、割り込みコンテキストも扱えない。UNIX ライク kernel の中はイベントドリブンの処理が主であることを考えると、今の Ajhc では kernel の初期化処理の一部しかスナッチできないように思えた。

今は kernel をスナッチしているわけではない。そこで、二度目のスケッチをよく観察してみた。すると、図 23.20 のように既存の C 言語コードは二つの部分にわかれていることがわかった。

二度目のスケッチでスナッチ対象にしていたのはメインコンテキストだ。このコンテキストが通常の動作では動いている。しかしクロック例外が起きた場合、即座にメインコンテキストは停止して割り込みハンドラに遷移する。つまり **受動的コンテキストスイッチ** が起きる。この割り込みハンドラをスナッチするにはコンパイル済み Haskell コードはもちろん、jhc ランタイムさえ **再入可能** でなければならない。なにせランタイムの関数を実行途中のどの段階においても、受動的コンテキストスイッチが起きて割り込みハンドラが起動し、さらには再度実行途中の同じランタイムの関数が再実行されてしまう可能性があるのだ!

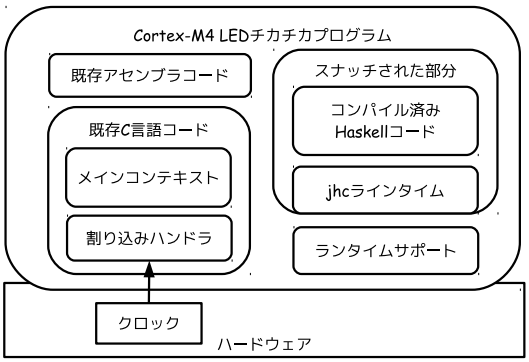


図 23.20: 二度目のスケッチでスナッチできなかった部分

少し脱線するが **能動的コンテキストスイッチ** というものも存在する。その例としては kernel のプロセス切り換えや、ユーザ空間スレッドライブラリのスレッドスイッチ、それからコルーチンの yield などがあるだろう。今回の Cortex-M4 のデモの中にはこの能動的コンテキストスイッチは見つからなかった。ChibiOS/RT [*39] のようなスレッドを使う OS を Cortex-M4 に搭載する場合にはスナッチ対象になる可能性はある。しかし受動的コンテキストスイッチよりは難易度が低いだろう。

しかし当然だが複数コンテキストの管理は難しい問題だ。既に jgc の実装が単一コンテキストを前提にして作り込まれてしまっている。しかも UNIX ライク kernel を実装することを考えると、スレッド対応だけではなく再入可能も必須だ。多くの言語処理系は受動的なコンテキストスイッチを受け付けないが、Ajhc に関してはその逃げは許されない。さらに割り込み応答のジッターをなくすためにはメインコンテキストが GC を実行している最中でも割り込みコンテキストに受動的に切り換え可能でなくてはならない (図 23.21)。GC の実行最中で単純に割り込み禁止にすればいいというものではない。GC の処理時間が長いかもしれないからだ。もちろん GC を小間切れにする方法もあるが、いずれにせよなんらかの対策が必要ということだ。

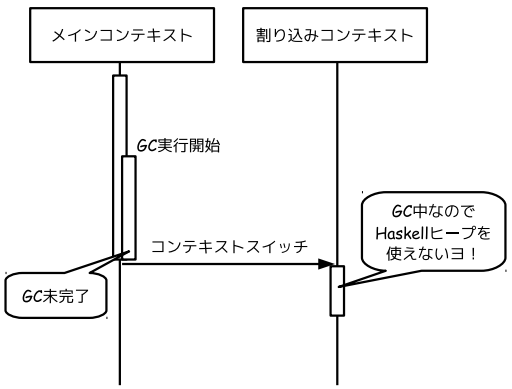


図 23.21: GC 実行中の割り込みコンテキストへのスイッチ

もう少しだけ整理して考えてみよう。そもそもコンテキストとはいったいなんなのだろう？ Ajhc の文脈でのコンテキストとは C 言語のコンテキスト、もっと言えば CPU のコンテキストのことだ。CPU は **レジスタ** を持っており、そのレジスタを変化させることでプログラムを実行している。近代の CPU ではさらにそのレジスタの中にはスタックポインタがあり、メモリ上の一点を指し示している。その先のメモリには **スタック領域** があり、その中に計算途中のデータが納められている。なんらかのイベントをトリガーにして、このレジスタとスタックを違うものに入れ代える行為がコンテキストスイッチだ。能動的コンテキストスイッチではこのトリガーのタイミングはソフトウェア自身で決定できる。ところが受動的コンテキストスイッチではこのトリガーはソフトウェアではなくハードウェアによって任意のタイミングで起きてしまう。この “いつトリガーが引かれるのかソフトウェア側からわからない” という点が難題なのだった。

[*39] ChibiOS/RT free embedded RTOS `http://www.chibios.org/dokuwiki/doku.php`

23.8.1 jgc 詳細

再入可能を実現するためにはクリティカルリージョンを見つけだして管理することが必要になる。Ajhc の中では GC がその主要因であることは間違いない。もう少し jgc のしくみを詳細に追ってみよう。jgc の GC ルートはいったい何なのだろうか。GC ルートの種類によっては排他制御が必要になるはずだ。ランタイムの gc_perform_gc 関数の動作を見てみよう。

1. GC マーク用のバッファ確保
2. すべての cache の下にある block の used bit を 0 クリア
3. 全 StablePtr をマークしてバッファにコピー
4. GC スタック内のスマートポインタをマークしてバッファにコピー
5. バッファの後ろからスマートポインタを取り出す
6. スマートポインタの先にあるスマートポインタをマークしてバッファにコピー
7. バッファが空でなければ 5 に戻る
8. バッファを解放

ということは、主に GC ルートとなるのは以下の二つのようだ。

- 確保した StablePtr
- GC スタック (gc 変数と gc_stack_base グローバル変数の間にある要素)

StablePtr の方は簡単だ。 newStablePtr 関数によって、GC が回収しないグローバルなポインタを作った場合、当該ポインタは GC ルートになるべきだ。一方 GC スタックというのは何者だろう？ saved_gc と gc_stack_base という二つのグローバル変数は Ajhc のランタイムが起動する際に GC スタック領域の先頭を指すように初期化される[*40] 。つまり最初は saved_gc と gc_stack_base は同じ場所を指していたわけだ。この saved_gc は Ajhc ランタイムから呼び出される Haskell コードのエントリポイントである _amain 関数の先では gc という引数によって取り回される。

```
// File: main_code.c (ミューテータ)
void
_amain(void)
{
        return (void)b__main(saved_gc);
}

static void A_STD
b__main(gc_t gc)
{
        return ftheMain(gc);
}
```

この引数 gc をミューテータがどのように使うかというところが少しややこしい[*41] 。

```
// File: main_code.c (ミューテータ)
static wptr_t A_STD A_MALLOC
fR$__fJhc_Basics_$pp(gc_t gc,sptr_t v80776080,sptr_t v58800110)
{
    {   gc_frame0(gc,1,v58800110); // <= GCルートに v58800110 を登録
```

[*40] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.c#L198
[*41] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.h#L53

```
wptr_t v100444 = eval(gc,v80776080);
if (SET_RAW_TAG(CJhc_Prim_Prim_$BE) == v100444) {
    return eval(gc,v58800110);
} else {
    sptr_t v106;
    sptr_t v108;
    /* ("CJhc.Prim.Prim.:" ni106 ni108) */
    v106 = ((struct sCJhc_Prim_Prim_$x3a*)v100444)->a1;
    v108 = ((struct sCJhc_Prim_Prim_$x3a*)v100444)->a2;
    {   gc_frame0(gc,2,v106,v108); // <= GCルートに v106 と v108 を登録
        sptr_t x7 = s_alloc(gc,cFR$__fJhc_Basics_$pp);
```

このソースコードでミューテータは s_alloc 関数か eval 関数を呼び出す手前で GC ルートにすべきスマートポインタを gc 変数の手前に登録している。つまり GC スタックにスマートポインタを積んでいるわけだ。この GC スタックへの登録マクロは gc_frame0 という名前だが、呼び出し前に必ずスコープを新たに作ることが約束事だ。こうすることで、スコープを抜けると自動的に元の GC スタックが復元される (図 23.22)。

こんな簡単なしくみで GC ルートを制御できるということが信じがたいが、とにかく jhc ではうまくいっていたようだ。おそらくコンパイルパイプラインの中段あたりに秘密があるに違いなかった。

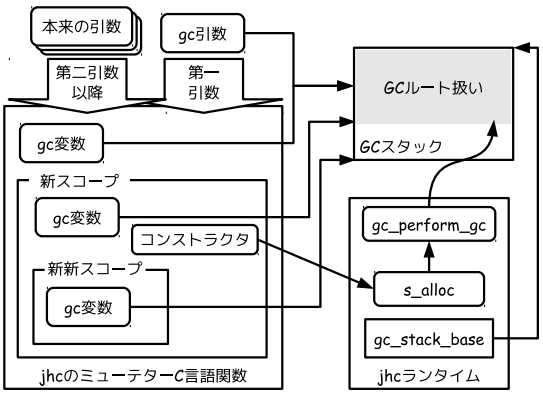


図 23.22: GC スタックとミューテータ

23.8.2 クリティカルリージョンの排他方法

jgc の GC ルートのしくみがわかったので、Ajhc のクリティカルリージョンをあらいだしてみよう。その前にクリティカルリージョンを保護するためのロックについてリストにしておこう。Ajhc は POSIX の上だけではなく様々なドメインで使えるようでなければならない。

- pthread_mutex_lock [*42]
- NetBSD kernel の mutex_enter [*43]
- CAS を使ったシンプルなロック
- 割り込み禁止
- ロックなし

POSIX の上で動かすときは pthread_mutex_lock を使えばいい。Cortex-M4 のようなマイコンでのロックは割り込み禁止で十分だ。遠い将来 NetBSD kernel をスナッチすることになったら mutex_enter に対応させる必要がある。これまでの jhc と互換性のある動作、つまり pthread を使わない場合にはロックなしで良いかもしれない。mutex_enter は API が複雑なので対応は後回しにすることにして、現状では pthread_mutex_lock に合わせた Ajhc ランタイムの API を定義して、他のロック機構を使う場合には切り換えられるようにした。

[*42] http://netbsd.gw.com/cgi-bin/man-cgi?pthread_mutex_lock++NetBSD-current
[*43] http://netbsd.gw.com/cgi-bin/man-cgi?mutex_enter++NetBSD-current

排他とは言ったが、再入可能についても考えた場合にはただ単にロックすれば良いというものではない。例えばコンテキスト A が処理 X 中にロック Y を獲得していたとしよう (図 23.23)。その最中に割り込みが入り、今度は割り込みコンテキスト B が処理 X を再度実行してロック Y の獲得を試みたとする。この時、当然ロック Y への二回目の獲得であるために割り込みコンテキスト B はロック Y の解放待ちになる。ところがロック Y を獲得しているコンテキスト A は割り込みコンテキスト B が完了しないかぎりロック Y を手放さない。みごとなデッドロックの完成だ。このとき処理 X は “再入可能ではない” とされる。スレッドセーフであっても再入可能でなくなってしまうこのような要因をコンパイラで防止することはできないものか？ Haskell コードやランタイムに mutex を使わせることによっていらぬ危険が増えるのは避けたかった。

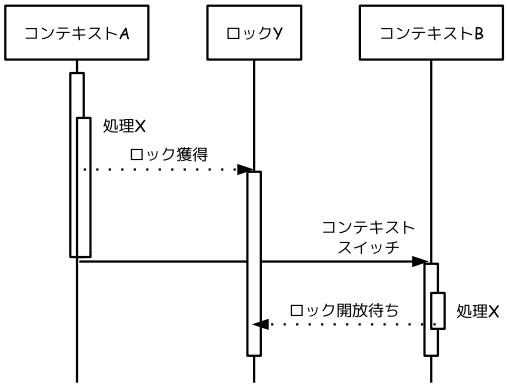


図 23.23: 割り込みと自コンテキストによる二重 lock

ぼくは長い時間この問題について考えてみた。しかし NetBSD kernel mutex_enter について調べているうちに結論が得られた。結論として、この再入時の二重ロックの問題はコンパイラによって解決されるべき問題ではない。mutex の実装によって防御するか、プログラマが適切に再入時の排他制御を設計する必要がある。

NetBSD kernel の mutex_enter 関数は、以下のような API で制御される。mutex_enter 関数は pthread_mutex_lock 関数と等価、mutex_exit 関数は pthread_mutex_unlock 関数と等価だ。注目すべきは pthread_mutex_init 関数と同じ機能を持つ mutex_init 関数だ。属性を指定する type 引数があるところまでは似ているが ipl という見慣れない引数が追加されている。この mutex をロックしている最中は ipl 引数で指定したレベルまで割り込みが禁止される。

```
// File: netbsd/src/sys/sys/mutex.h
* void mutex_init(kmutex_t *mtx, kmutex_type_t type, int ipl);
* void mutex_enter(kmutex_t *mtx);
* void mutex_exit(kmutex_t *mtx);
```

ここで割り込みレベルという聞き慣れない用語が出てくるが、これはハードウェア割り込みを応答性を求められる順番に優先度分けするものだ。より応答性が必要な優先度の高い割り込みを禁止している最中はそれより低い優先度の割り込みも禁止される。ややこしいが、ここでは優先度をつけた割り込み禁止の機構だと理解して問題ないだろう。

例えば、Intel PRO/Wireless 3945ABG の割り込みルーチンを見てみよう。wpi_attach 関数がデバイスドライバの初期化だ。この中で wpi_intr 関数が割り込みレベル IPL_NET で励起されるように設定されている。

```
// File: netbsd/src/sys/dev/pci/if_wpi.c
static void
wpi_attach(device_t parent __unused, device_t self, void *aux)
{
// --snip--
    sc->sc_ih = pci_intr_establish(sc->sc_pct, ih, IPL_NET, wpi_intr, sc);
// --snip--
    if ((error = wpi_alloc_rpool(sc)) != 0) {
// --snip--
}
```

```
static int
wpi_alloc_rpool(struct wpi_softc *sc)
{
// --snip--
    mutex_init(&ring->freelist_mtx, MUTEX_DEFAULT, IPL_NET);
    SLIST_INIT(&ring->freelist);
```

この wpi_intr 関数は割り込みコンテキストで実行されるにもかかわらず、中では以下のように mutex を使う。しかし先の wpi_alloc_rpool 関数の中でこの mutex は IPL_NET 割り込みレベルで初期化されているため、mutex_enter(&sc->rxq.freelist_mtx) と mutex_exit(&sc->rxq.freelist_mtx) の区間はアーキティクチャ側で IPL_NET に対応する割り込み禁止をかけた状態になるはずだ。そのためこの mutex を保持している最中は IPL_NET 割り込みによる受動的コンテキストスイッチが起きないことになる (図 23.24)。

```
// File: netbsd/src/sys/dev/pci/if_wpi.c
static int
wpi_intr(void *arg)
{
// --snip--
    if (r & WPI_RX_INTR)
        wpi_notif_intr(sc); // 中で wpi_alloc_rbuf 関数を呼ぶ
// --snip--
}

static struct wpi_rbuf *
wpi_alloc_rbuf(struct wpi_softc *sc)
{
    struct wpi_rbuf *rbuf;

    mutex_enter(&sc->rxq.freelist_mtx);
    rbuf = SLIST_FIRST(&sc->rxq.freelist);
    if (rbuf != NULL) {
        SLIST_REMOVE_HEAD(&sc->rxq.freelist, next);
        sc->rxq.nb_free_entries --;
    }
    mutex_exit(&sc->rxq.freelist_mtx);

    return rbuf;
}
```

POSIX の上でシグナルハンドラのようなものを使う際にもこのような特別な排他制御をコンパイラの利用者やライブラリ設計者が適切に設定するべきだ。例えばシグナルハンドラ中で実行可能な関数は制限されるべきだ。もちろん POSIX にその制限は明記されている。もしシグナルを受けた際にその限定された API 以外の関数を実行する場合には sigwait [*44] でシグナルを受け取る専用のスレッドをプロセス内で一つ作り、適切なスレッドにシグナル受信メッセージを配送すべきだ。

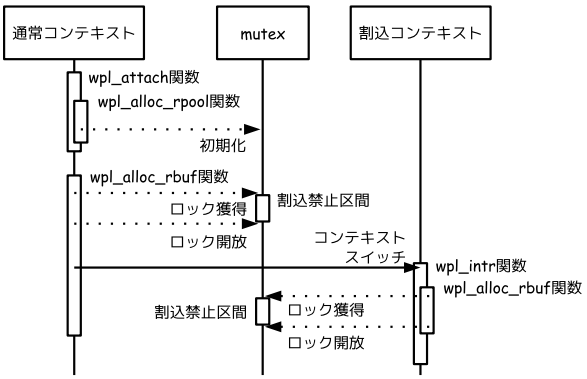


図 23.24: mutex_enter の ipl について

一般化すると、Haskell で書くコードであったとしても、そのコードが受動的コンテキストスイッチを受ける側なのか、それともハードウェアトリガーによって起動される側なのかを意識して設計する必要があるのだ。

コンパイラ利用者がこの制約を受け入れないかぎり、そのコンパイラは C 言語をスナッチするに足る能力を得ることはできない、というのが今の時点でのぼくの結論だった。

23.8.3 再入可能の実現

コンパイラへの要求がぼんやりと見えてきた気がする。さぁ Ajhc のクリティカルリージョンを調べてみよう。ざっと調べる限りでは以下に大別されるようだった。これらをコンテキストローカルな配置にすることができないか、もしそれが不可能であれば排他してやる必要がありそうだ。

A. ランタイム
   1. グローバル変数として確保される gc_t saved_gc
   2. グローバル変数として確保される struct s_arena *arena
   3. GC スタック
   4. megablock
   5. struct s_arena の下に繋がれる struct s_cache

B. ミューテータ
   1. グローバル変数として確保される struct s_cache
   2. GC スタックを管理する gc 引数

グローバル変数に関してはコンテキスト毎の管理にできないか検討して、無理であれば mutex で排他するしかないだろう。megablock に関しても複数の利用者がいる場合であれば alloc/free 時にロックするしかない。残る問題は GC スタックだ。

思い出してみよう。GC スタックはミューテータによって管理されていて、GC ルートになるのだった。この GC スタックをコンテキスト間で共用することを考えよう。GC スタックはミューテータの gc 変数のスコープによって管理されていた。変数スコープはコンテキストの外からは知ることが難しい。そこで退避されているコンテキストの中からスタックポインタを取り出して、当該

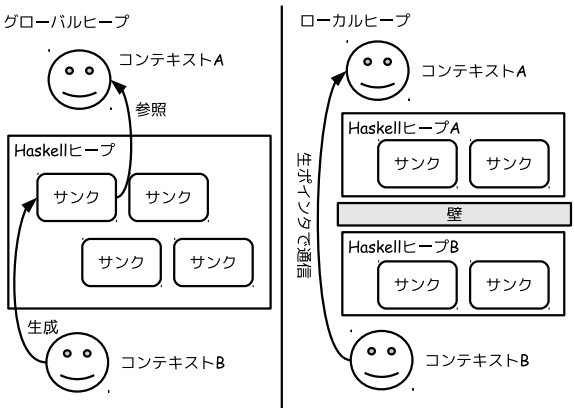


図 23.25: グローバルヒープとローカルヒープ

[*44] sigwait(2) http://netbsd.gw.com/cgi-bin/man-cgi?sigwait++NetBSD-current

スタックの中でスマートポインタに該当しそうなものを GC ルートと見做すしかない。でもこれは Boehm GC [*45] とほとんど同じ処理をすることになる。もしくは別の案として、GC スタックという領域を C 言語のスタックとは別に確保して、GC スタックへのスマートポインタの追加と削除をグローバルロックで排他する……ナンセンスだ。

もっと高い場所からこの GC ルートと Haskell ヒープの問題を見渡してみよう。そこには二つの道が見える。**グローバルヒープ**と**ローカルヒープ**だ (図 23.25)。グローバルヒープはこれまで議論してきた GC ルートの管理方式だ。すなわちコンテキストが多数あっても Haskell ヒープは唯一一つだけ用意する。そのため Haskell ヒープへの操作は場合によって排他制御する必要が生じる。もう一つ考えられるのが Haskell ヒープをコンテキスト毎に一つずつ持つ案だ。

グローバルヒープのメリットは明確だ。コンテキスト間で同一のヒープを使っているため、排他処理さえ気をつければコンテキスト間のサンクの受け渡しがゼロコピーで実現できる。ほとんどの言語処理系はグローバルヒープを選択している。しかし、先に見たとおり jgc との相性は悪いようだ。Ajhc でグローバルヒープを使う場合には GC をほぼまるまる再設計する必要がありそうだ。

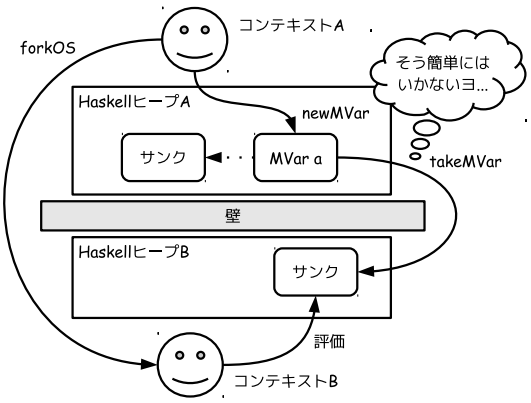


図 23.26: MVar に繋いだサンクは誰が評価すべきなのか？

ローカルヒープを採用した場合にはグローバルヒープとは異なりコンテキスト間でサンクを受け渡すのは難しくなる (図 23.26)。サンクを作る時点ではコンテキストの中のみで消費するのか、MVar のようなしくみを使って別のコンテキストに渡すために作るのか判別できない。MVar に繋いだ後にサンクを作ったコンテキストではない別のコンテキストから評価しようとするとややこしい事態になる。誰がこのサンクを評価すべきなのだろうか？

ローカルヒープにはさまざまな問題が考えられるが、ぼくは Ajhc の再入可能を実現するにあたってローカルヒープを採用することにした。理由はいくつかあるが最も大きな理由は Haskell 言語が状態の共有を嫌う言語だということだ。グローバルヒープを採用した場合、状態の共有は実現しやすいけれど並列実行に気をつかう必要がある。

ローカルヒープを採用した場合、ロックなしに並列実行を実現できる。並列 GC もおまけでついてくる。メインコンテキストが GC 実行中でも割り込みを受けられる。さらにコンテキストとゴミが紐づいているために、自コンテキストで出したゴミは自コンテキストの GC ペナルティになる。他人に迷惑をかけないわけだ。もっと考えると、kernel の中はイベントドリブンが主だ。ということは GC する前にコンテキストそのものが消滅する可能性も高いかもしれない。コンテキストが消滅したら紐づいていた Haskell ヒープをまるごと回収してしまえば良いのだ。ヒープ回収の時点になってしまえばマーキングする必要もない。

最初のコンテキストの定義を思い出してみよう。コンテキストとはレジスタとスタックのセットなのだった。ローカルヒープはこのコンテキストのセットに二つの要素 **GC スタック**と **Haskell ヒープ**を追加することに他ならない。いわば Haskell 文脈の中においてだけ C 言語の ABI を拡張していることになる。こう考えればなんともシンプルな発想だ。

方針は決まった。GC スタックと Haskell ヒープをコンテキスト毎に持つように変更しよう。まず考えるべきなのが管理構造体だ。Haskell ヒープは arena グローバル変数で管理されていた。この arena グローバル変数をコンテキストローカルに持つには gc 変数を同じように関数の引数で持ちまわるのが一番簡単だ。POSIX の外ではスレッドローカルストレージなど使えないからだ。一見して gc 変数と arena 変数の二つをミューテータ内の全関数の引数に追加するのは無駄に思えた。arena 変数の 1 メンバーとして gc 変数を持てば良いのではないか？ しかし調査をすすめてみるとそれほど簡単な問題ではないことがわかった。arena 変数の方は簡単だ。常にコンテキストに対して一つだけ存在し、そのアドレスは変化しない。しかし gc 変数はミュー

[*45] A garbage collector for C and C++ `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

テータ内のローカルスコープによってその具体的なアドレスは変動している。スコープや関数を抜ければ元の状態が復元されなければならない。このような復元をコンパイラパイプライン側で実現するのは不可能ではないが、煩雑だ。また、当然この gc 変数の復元にはコストがかかる。x86 のようにレジスタが少ないアーキティクチャはこれから淘汰されることを祈って、ぼくは gc と arena 両方の引数をミューテータに追加することにした。

arena 変数の構造体を変更しよう (図 23.27)。いままでグローバル変数で定義されていた管理構造体はできるだけ arena 変数の中に埋め込むべきだ。link は arena 変数を管理するリスト用メンバーだ。arena 変数は used_arenas か free_arenas どちらかのグローバルリストに繋がれる。全コンテキストにメッセージを送りたければ used_arenas を辿れば良いし、新しくコンテキストに arena 変数を割り当てたければ free_arenas から取り出せば良いわけだ。gc_stack_base メンバーはそのままこれまでの gc_stack_base グローバル変数の意味そのままだ。GC スタックの初期ポインタを保存している。array_caches と array_caches_atomic メンバーもこれまでのグローバル変数をそのまま arena 変数に埋め込んだものだ。force_gc_next_s_alloc というメンバーは “次回の s_alloc 時に強制的に GC をかける” というフラグだ。なぜこんなものが必要になるのかというと、hs_perform_gc という C 言語関数が Haskell 標準で決まっているためだ。この関数は Ajhc 側のコンテキストを全く知ることなく強制 GC を指示する。そのため、今回の実装ではランタイムをグローバルロックした後で、このフラグを立てる必要がある[*46] 。そして Haskell ヒープから領域を確保する s_alloc 関数の頭でこのフラグの判定して強制 GC を実行すればいい[*47] 。

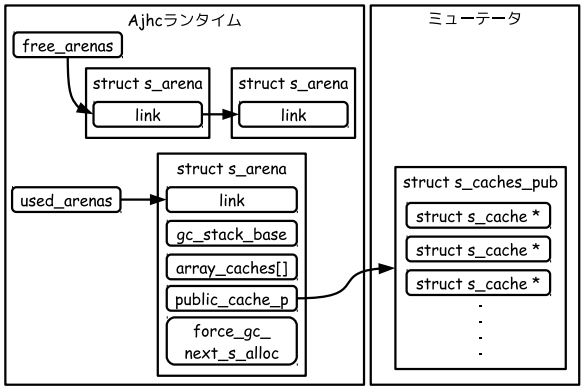


図 23.27: s_arena 構造体と GC スタックの構造と配置

arena 変数にはもう一つ public_caches_p というメンバーが追加されている。このメンバーはミューテータ側で宣言される cache 管理構造体へのポインタだ。s_arena 構造体の中身はミューテータ側から隠されているが、このミューテータが宣言した cache だけは見えてほしい。そのためにこんなややこしいしくみがどうしても必要だった。ミューテータは以下のようなコードで、コンテキスト起動時に public_caches_p メンバーを初期化する。

```
// File: main_code.c (ミューテータの例)
struct s_caches_pub {
    struct s_cache *cCJhc_Prim_Prim_$x3a;
    struct s_cache *cFR$__fJhc_Basics_$pp;
    struct s_cache *cCJhc_Type_Ptr_Ptr;
};
// --snip--
void
jhc_hs_init(gc_t gc,arena_t arena)
{
        alloc_public_caches(arena,sizeof(struct s_caches_pub));
        find_cache(&public_caches(arena)->cCJhc_Prim_Prim_$x3a,arena,
                   TO_BLOCKS(sizeof(struct sCJhc_Prim_Prim_$x3a)),2);
        find_cache(&public_caches(arena)->cFR$__fJhc_Basics_$pp,arena,
```

[*46] https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L826
[*47] https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L573

```
                TO_BLOCKS(sizeof(struct sFR$__fJhc_Basics_$pp)),3);
        find_cache(&public_caches(arena)->cCJhc_Type_Ptr_Ptr,arena,
                TO_BLOCKS(sizeof(struct sCJhc_Type_Ptr_Ptr)),0);
}

// File: ajhc/rts/rts/gc_jgc.c
void
alloc_public_caches(arena_t arena, size_t size) {
        if (arena->public_caches_p == NULL) {
                arena->public_caches_p = malloc(size);
        }
}
```

ちょっとややこしくなってきた。気分をかえて、ローカルヒープを採用した Ajhc でのコンテキストの一生を見てみよう。

Ajhc におけるコンテキストは C 言語からの関数呼び出しによって始まる。C 言語から Haskell コードの main 関数を呼び出す際には_amain 関数を C 言語から呼び出すのだった。

```
// File: main_code.c (ミューテータ)
void
_amain(void)
{
        arena_t arena;
        gc_t gc;
        gc = NULL;
        arena = NULL;
        jhc_alloc_init(&gc,&arena);
        jhc_hs_init(gc,arena);
        b__main(gc,arena);
        jhc_alloc_fini(gc,arena);
}
```

C 言語文脈にはなかった gc と arena という引数を二つの言語の界面である export 関数で初期化するように変更した (図 23.28)。jhc_alloc_init 関数では gc と arena をプールから確保して初期化する。その後 jhc_hs_init 関数を使ってミューテータで定義されている cache を初期化する。export 関数から呼び出される b__main 関数から先は gc と arena を引数によって引き回す Haskell コンテキストの世界だ。b__main 関数が完了したら、この gc と arena は解放される。もちろん free してしまうのでなく、ランタイム側内部でプールして次回使用時に使いまわす。

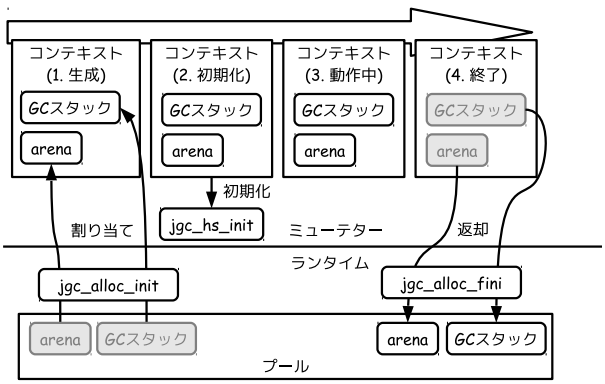


図 23.28: gc と arena 変数とコンテキストの一生

このような arena 引数を持ち回るしくみをミューテータに追加するには、もちろん Ajhc のコンパイルパイプラインに修正を加える必要がある。この arena 引数を追加する変更は grin=>C への変換

に細工をすれば実現できる (図 23.29)。

compileGrin 関数は Grin 型を C 言語のソースコードが入った ByteString 型に変換する。この関数の頭で C 言語ソースコードの構造が ans で示されている。Grin 型で定義されている内容を ans につめかえるのがこの変換の役目だ。ans の個々の要素に関して、修正箇所を見てみよう [*48] 。

最初に変更するべきは jgcs エントリだ。GC タイプとして jgc が選択された場合にのみ s_caches_pub 構造体にミューテータ側で定義するべきキャッシュを列挙することにしよう[*49] 。さらに jhc_hs_init 関数の宣言に arena 引数を加える。この header と body は ans のエントリだった。つまり C 言語の関数定義を generateC に通すとヘッダと本体が得られるのだ[*50] 。Haskell 由来の C 言語関数全てに第一引数に gc を第二引数に arena を取ることにしよう。jhc_hs_init 関数の中身は icaches で定義されている [*51] 。ミューテータ側で定義された cache は arena の下の s_caches_pub メンバーに配置することにしたので、そのように書き換えた。もし当該の Haskell 関数が FFI で export されていたら gc と arena の新規割り当てを行なうようにした[*52] 。

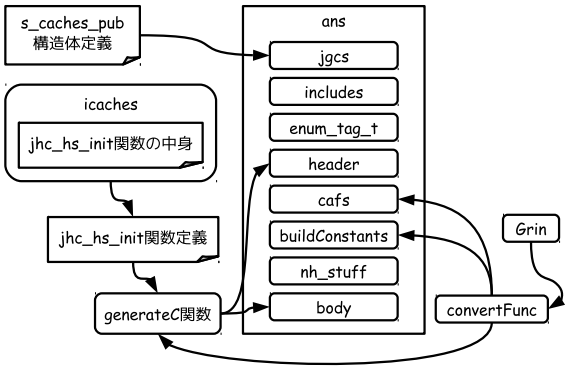


図 23.29: grin → C 変換

また場合によっては Haskell コードからランタイムの関数を import して使いたい時がある。この時にも gc と arena を渡す必要がある。そこで jhc_context という特殊な import 種別を作った [*53] [*54] [*55] 。

jhc_context 付きの import のおかげで、gc と arena 引数付きで C 言語の関数を呼び出せるようになった (図 23.30)。export にも同様のしくみが必要になりそうな気もしたが、いまのところは明確な用途が見あたらないので import 側のみの修正にとどめた。

例えばランタイムの gc_new_foreignptr 関数は、これまでは Haskell 側から gc 引数がわたってこないので、saved_gc というグローバル変数経由で GC スタックへのポインタを受け取っていた。jhc_context を使えば、Haskell 側から直に gc と arena 引数を任意の C 言語関数に渡すことができるようになった[*56] [*57] 。

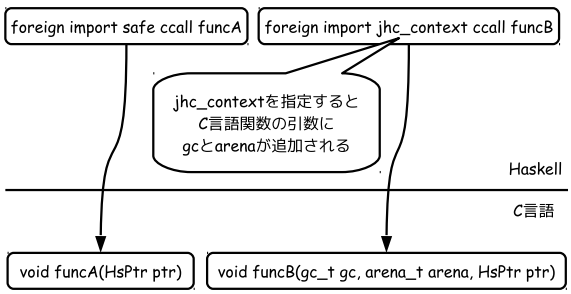


図 23.30: C 言語の関数呼び出しに gc と arena を追加する

[*48] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L129
[*49] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L150
[*50] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L157
[*51] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L158
[*52] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L214
[*53] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/Prims.hs#L22
[*54] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/FrontEnd/ParseUtils.hs#L410
[*55] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/C/FromGrin2.hs#L596
[*56] https://github.com/ajhc/ajhc/blob/v0.8.0.7/lib/jhc/Jhc/ForeignPtr.hs#L64
[*57] https://github.com/ajhc/ajhc/blob/v0.8.0.7/rts/rts/gc_jgc.c#L781

やれやれ。これであらかた GC スタックと arena 変数をコンテキストローカルにできたはずだ。あとは残ったクリティカルリージョンを mutex で保護するだけだ。様々なロック機構に対応するため、図 23.31 のように 3 種類の並列実行に大別してオプションを作った。将来はもっと増えるかもしれない。

また、ランタイムのグローバルロックのために mutex をグローバル変数で一つだけ定義した。ロックを細分化した方がいいかもしれないが、今は簡単な実装にしておきたい。

コンパイルオプション	-fnothread	-fpthread	-fcustomthread
ロックの初期化	ダミー実装	pthread実装	利用者が実装
ロックの獲得	ダミー実装	pthread実装	利用者が実装
ロックの開放	ダミー実装	pthread実装	利用者が実装
forkOS	単純関数実行	pthread実装	利用者が実装

図 23.31: コンパイルフラグによる mutex の実装差異

23.8.4 スレッドの実装

これでコンテキストとコンテキストの分離が実現できた。ということは割り込みコンテキストのような受動的なコンテキスト切り換えが可能になったわけだ。もちろん “jhc_mutex_t を適切に設計した場合は” というただし書きはついてしまうがこれは設計開始当初に受け入れた制約だ。

このままでもシングルコアのマイコンを制御するには当面十分だ。しかしどうせ再入可能にできたのであれば、POSIX 上でのスレッドもサポートしておきたい。継続的インテグレーション [*58] することを考えれば、マイコンよりも POSIX 上でテスト可能にした方がいい。またマルチコア環境でしか起きない不具合も検出できるかもしれない。

既にランタイムの mutex は pthread を使って実装済みなので、状態共有をしない並列実行であれば forkOS を pthread_create を使って作れば十分だろう。まずランタイム側で pthread_create を使ったスレッド生成の関数を作る。Haskell 側は GHC の forkOS 実装を真似て StablePtr 経由で IO () を渡すようにした (図 23.32)。どうも jhc では StablePtr は直接 FFI に渡すことはできず、Bang_という型で包む必要があるようだった。

この実装で動きそうに思えるが、なぜかコンパイルエラーだ……

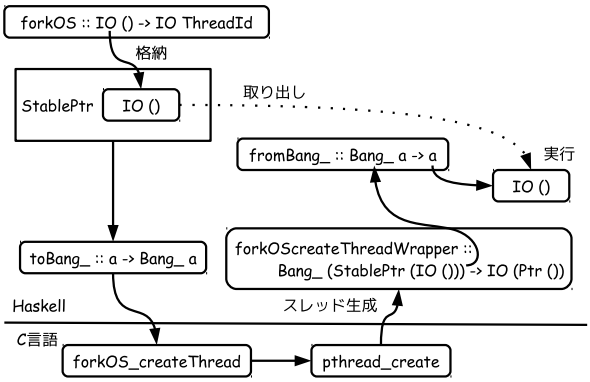


図 23.32: forkOS を pthread で実装

```
ajhc: Grin.FromE.compile'.ce in function: FE@.CCall.forkOScreateThreadWrapper
can't grok expression: <fromBang_ x108042543::IO ()> x62470112
```

エラーメッセージを読むかぎりではコンパイルパイプラインの中の Grin.FromE.compile'.ce 関数は fromBang_型を受け取ることを予期していないようだ。FromE での評価には ce 関数と cc 関数の二つの形式がある。今回のエラーが起きている ce 関数は正格評価で、cc 関数は遅延評価だ。fromBang_型を受け取るのは cc 関数のみで、ce 関数には当該のパターンマッチが存在しない。cc 関数でのパターンマッチの前に fromBang_

[*58] 執筆時点では Travis CI を使っている https://travis-ci.org/ajhc/ajhc

プリミティブを削除する関数をかませてしまうことにした[*59]。おそらく Bang_型は正格にするためのプリミティブだと思われるので ce 関数では無視してよさそうだが、多少不安だ。fromBang_プリミティブについては後付けでもう少し調査が必要そうだ。

さて、いくつかテストを書いてみよう。最初のテストは三つのスレッドから文字を吐き出すプログラムだ。これはさすがにうまくいった。

```
import Control.Monad
import Control.Concurrent

main :: IO ()
main = do
  l <- putStrLn "Type some string and enter." >> getLine
  forkOS $ (forever $ putChar '*')
  forkOS $ (forever $ putStr l)
  forever $ putChar '.'
```

次にマイコンと同じような動作をするテストを書いてみた (図 23.33)。つまり割り込みコンテキストの動作を pthread を使ってエミュレーションするのだ。まず C 言語側で Haskell コードに隠れてスレッドを生成しておく。この減算スレッドは Haskell で書かれた timingDelayDecrement 関数を定期的に呼び出して、TimingDelay グローバル変数が 0 になるまで減算する。Haskell 側の myDelay 関数は待ち時間が経過するのをビジーループで待ちあわせる。

このテスト実行すると、減算スレッドの sleep が解除されると同時に myDelay 関数の待ち合わせが解除された。これでマイコンの割り込みをスナッチする自信がついた。

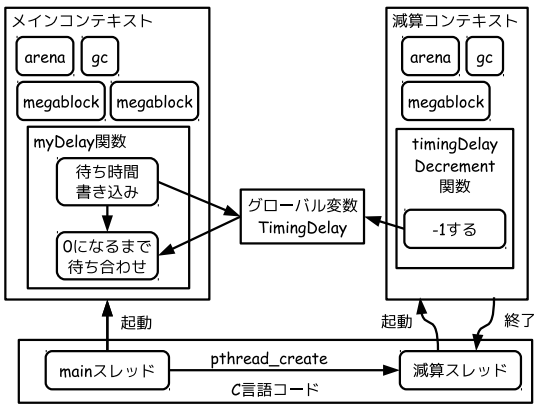


図 23.33: 割り込みハンドラの動作を pthread でエミュレーション

23.8.5 割り込みコンテキストのスナッチ

今回のスケッチで作った再入可能の機能を使って、Cortex-M4 マイコンの割り込みハンドラを Haskell でスナッチしてみよう。

図 23.34 は Haskell でスナッチしたクロック割り込みによる delay の実装だ。まずマイコンの電源が入ると C 言語の main 関数が Haskell のメインコンテキストを起動する。これまで出てきたように_amain 関数がメインコンテキストの入口だった。この時、Ajhc ランタイムは arena と gc をプールからメインコンテキストに割り当てる。メインコンテキストは何度かコンストラクタを呼び出すうちに、おそらくは、二つの megablock を手に入れるだろう。この megablock 達の中にメインコンテキストが作ったサンクが納

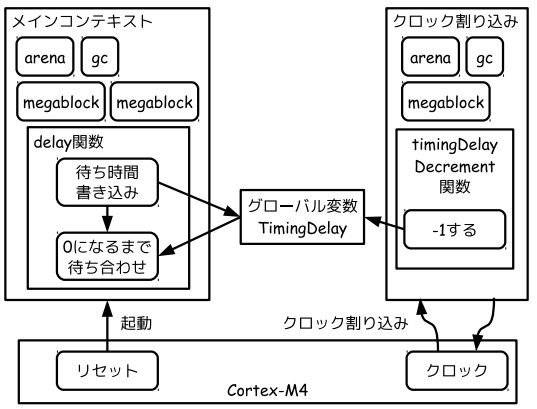


図 23.34: 割り込みハンドラを Haskell でスナッチ

[*59] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/Grin/FromE.hs#L365

めれれる。メインコンテキストは電源 OFF
までの間終了せず、ずっと滞留することに
なる。

一方 C 言語の main 関数がクロックの初期化をすると、メインコンテキストとは全く関係のないタイミングでクロック割り込みが発生するようになる。このクロック割り込みが発生するとクロック割り込みベクタに実行が移り、クロック割り込みベクタは Haskell で記述されている timingDelayDecrement 関数を起動する。この時、メインコンテキストとは別の Haskell コンテキストを起動したので、メインコンテキストとは別に arena と gc がランタイムによって割り当てられる。クロック割り込みコンテキストが終了すると、arena,gc,megablock はランタイムによって回収されて次回誰かが使うまでプールに保管する。もちろん次回このプールから arena 達を確保するすのは次のクロック割り込みだろうけれど。

こうしてコンテキストの分離が済んでしまえば、Haskell での割り込み実装は簡単だ。C 言語側でグローバル変数 TimingDelay を用意して、このグローバル変数を Ptr 型によってアクセスすればコンテキスト間通信ができる。

さて、長い今回のスケッチで結局 Cortex-M4 マイコン上で動作するプログラム[*60] のどこまで Haskell でスナッチすることができたのだろうか？ 実行バイナリ内に含まれるシンボルの中で、Ajhc ランタイム、ランタイム用ダミー関数、malloc、Haskell コード由来ではないものを列挙してみよう。

- 例外/割り込みハンドラ
- data/bss 初期化コード (アセンブラ)
- Haskell スレッド間通信用グローバル変数
- クロックの初期化コード
- GPIO の初期化コード
- LED の GPIO 初期化コード

割り込みハンドラは使っていないものは全部 abort だ。data と bss の初期化は C 言語を使う前にアセンブラで行なう必要があるため Haskell 化は不可能。Haskell スレッド間通信用グローバル変数については MVar のようなスレッド間状態共有のしくみを構築すれば解決するだろう。もちろんそれは非常に困難な課題だけれど。

上記以外の初期化コードは全てメモリへの読み書きによって作られている。ということは Ptr 型を使えば Haskell でスナッチできるということだ。つまり Cortex-M4 マイコン上で動作するソフトウェアに関しては、限界まで Haskell でスナッチする見込みがたったのだ!

23.9 突然の連絡

何度目かの Ajhc リリースノートを書いていると、携帯のアラームが鳴った。

> From J, at 2013 年 05 月 25 日 (土):
> **会社をやめることにしたよ。きっと jhc の開発に戻れると思う。**
> **いままでメーリングリストで返事ができなくてすまなかった。**
> **それもみんな会社が従業員に個人プロジェクトを持つことを許さなかったからだ。**
> **オレはもう一度 jhc のようなオープンソース活動をしてみたい。だからやめた。**

J、今まで疑っててごめん。本当はずっと気にかけていてくれたんだね。ぼくは最初から一人ぼっちじゃなかったんだ。

ふいにあの娘の声が聞こえたような気がする。
「ああ、わかっているさ。次のスケッチはもう見えている」
ぼくは思わず独り言をつぶやいた。

[*60] https://github.com/ajhc/demo-cortex-m3/tree/master/stm32f3-discovery

23.10　これからのこと

結局この一連のスケッチで勝ち取ったものはなんだったのだろう？ 一見なにも変化していないように思える。なにせシングルコアの小さなマイコンで動くソフトウェアを型付けできたにすぎない。しかしとにかく実際に使われているプログラムを型によって少しずつ再設計できる、ということは確かめられたようだ。これからの課題も山積みだが、何が必要なのかは今なら理解できる。

- MVar のような型を経由したスレッド間状態共有の実現
- ユーザー空間スレッドの実装
- Haskell Platform 相当のライブラリを移植
- cabal のようなパッケージ管理システム
- 大規模な Haskell プログラムのコンパイル実績
- さらなる応用例の提案

おぼろげながら見える。スケッチの糸が遠くデザインへと繋がっている。ふと目をあげると、まだ見慣れぬ優しい **アラフラの海** [*61] が広がっていた。

図 23.35: ”ビンタンビール”

23.11　参考文献

- 図 23.35: Sunset with Bintang - Copyright (C) 2010 Matteo Catanese All Rights Reserved. [*62] [*63]
- The NetBSD Project `http://www.netbsd.org/`
- Metasepi Project `http://metasepi.org/`
- Metasepi Project の近傍 `http://metasepi.org/map.html`
- Five-Year Plan for the Metasepi Union `http://metasepi.org/doc/20130508_5year_plan.pdf`
- Jhc Haskell Compiler `http://repetae.net/computer/jhc/`
- Ajhc - Haskell everywhere `http://ajhc.metasepi.org/`
- 「ファウンデーション」、アイザック・アシモフ著、岡部宏之訳、ハヤカワ文庫

[*61] デザイン Arafura `http://metasepi.org/posts/2013-01-09-design_arafura.html`
[*62] `http://www.flickr.com/photos/matteocatanese/5166690955/`
[*63] Licensed under a Creative Commons Attribution 2.0 Generic License. `http://creativecommons.org/licenses/by/2.0/`

第24章

jhcコピペ

— @master_q

24.1 jhcたんへの愛の歌

聞いてください！

日本語版

jhc！ jhc！ jhc！ jhcぅぅううわぁあああああああああああああああああああああああああああん！！！ あぁあああああ…ああ…あっあっっー！ あぁあああああああ！！ jhcjhcjhcぅううぁわぁあああああ！！！ あぁクンカクンカ！ クンカクンカ！ スーハースーハー！ スーハースーハー！ いい匂いだなぁ…くんくんんはぁっ！ jhcたんのオレンジの髪をクンカクンカしたいお！ クンカクンカ！ あぁあ！！ 間違えた！ モフモフしたいお！ モフモフ！ モフモフ！ 髪髪モフモフ！ カリカリモフモフ…きゅんきゅんきゅい！！ darcsのjhcたんかわいかったよぅ！！ あぁぁああ…あああ…あっあぁあああああ！！ ふぁぁああああんんっ！！ Parsec使えて良かったねjhcたん！ あぁあああ！ かわいい！ jhcたん！ かわいい！ あっああぁああ！

haskell2010化されつつあって嬉し…いやぁあああああああ！！！ にゃあああああああああん！！ ぎゃあああああああああ！！ ぐあああああああああ！！！ ソースコードなんて現実じゃない！！！！ あ…バイナリもよく考えたら…

j h c ちゃんは現実じゃない？ にゃあああああああああああああん！！ うぁあああああああああ！！ そんなぁあああああ！！ いやぁぁぁあああああああああ！！ はぁあああああああん！！ NetBSDぁあああ！！ この！ ちきしょー！ やめてやる！！ Haskellなんかやめ…て…え！？ 見…てる？ スレッド対応のjhcちゃんが僕を見てる？

STM搭載のjhcちゃんが僕を見てるぞ！ jhcちゃんが僕を見てるぞ！ 中断GC対応のjhcちゃんが僕を見てるぞ！！

コンパイラのjhcちゃんが僕に話しかけてるぞ！！！ よかった…世の中まだまだ捨てたモンじゃないんだねっ！ いやっほぉおおおお！！！ 僕にはjhcちゃんがいる！！ やったよJohn！ ひとりでできるもん！！！ あ、独自RTSのjhcちゃあああああああああああああん！！ いやぁあああああああああああああ！！！！ あっあんああっああんあgrin様ぁあ！！ リ、リージョン推論！！ superkindぉおおおおお！！！ superboxぅぅううう！！

ううっうぅうう！！ 俺の想いよjhcへ届け！！ サンタモニカのJohnへ届け！

In English

Jhc! Jhc! Jhccccccccccccccccccccccccccahhhhhhhhhhhhhhh!!! yarghh...uh...aaahah-! AaAAAAA!!! JHCJHCJHCaaaaAAAAaaa!!! Ae...sniffsniff! sniffsniff! sssssaaahssssaaah.. smells good.... sniff
gasp! I can haz sniffsniff orange-colored hair de jhc?! sniffsniff! Aah! No! I want fluffing! fluff! fluff! Hair hair fluff fluff! Scratch scratch fluff fluff... Kyunkyunnkyui!! Jhc-tan was so kawaii in the darcs repo!! AaaAA...AAA...AhAaAAA!! FaaAAAnng! Congrats on ready to use Parsec Jhc-tan! AaAAAAA! So cute! Jhc-tan! Kawaii! AaaAAA!
Compat haskell2010 were grea...nnNrAGGggghHH!! Nyaaaaaargh!! UGyaaaAAAAA!!! Nnnnnnyyyyuuuurrrrgggghhhh!!! Source code..... AREN'T reality!!!? ...what about the binary....
Jhc I S N' T R E A L????? AAAAArgggghhHH!!! Uwaaaaaaannn!!
NononononoNONONONONO! EEeeeeaAAAA!!!! HGGGGrrrrruuyynnnnnNN!! Halkeginiaaaaaa!! You! BASTARDS! I'm quitting! I'm QUITTING HASKEL...L....huh!? She's...looking? Threaded Jhc-chan on the cover is looking at me?
Jhc using STM IS looking at me! Jhc... at me! Jhc-chan in the interrupting GC are looking at me!!
Jhc-chan as compiler is talking to me! Phew... reality ain't so bad after all! Yesssssss! Wheeeee! YEAH!!! I have jhc!! I've done it John, I can do it alone!!! C..custom... RTS'ed JHC-CHAAAaaaaaAAaaAAAN!!!!!!!! HyaaAAAaaaAaaaAAaA!!!!! Ahahaah...ahahaaahhhhahaGrin! Re, Region-based Memory Management!! SuperkindaaaAAAAAAaAA!!! SuperboxaaAAAAA!!
u....uuuu..sniffsniffuuuUUU!!! May my love reach jhc!! May my love reach John at Santa Monica!

24.2 参考文献

- ルイズコピペ http://dic.pixiv.net/a/**ルイズコピペ**

第25章

侵略者と転校生とアイドルとイカが再帰を学ぶそうですよ！

— @nushio

要旨

最近レコードアクセスのために lens を使う機会が増えてきたので、ちゃんと勉強しようと思いたち、“Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire” を読むことにしました。そしたら、未定義・意味不明の記号があまりにたくさん使ってあってとても読めたものではなく、他の論文やブログ記事を参照し、ロゼッタストーン片手に古代文字を解読するような作業の結果、ようやく読めるありさまでした。先駆的論文としての価値は強調してしすぎることはありませんが、より新しい論文や、ライブラリで補完・改善されている点も見えてきました。そこで、この論文を読もうという人にも、そうでない人にも私の体験をたのしくお伝えすべく古文書の解読をめぐる冒険活劇に仕立てることにしました。本文中で「旧約甘蕉書」とされている文書のページ数や式番号は原論文のそれに準じています。原論文は Web からダウンロードできますので、印刷して片手に置きながらお読みください。また、本文に登場するソースコードは `https://github.com/nushio3/`で公開しますので、こちらもあわせてご覧ください。

ちなみに、甘蕉はバナナという意味です。

ちなみにちなみに、レコードアクセスのための lens と本稿で解説する lens は結局別物だったようです。ひどい話ですよね。本来知りたかった方の lens の記事は——たぶん明日の出来事だ。

序

「ブンブンブブーンブンブブーン、ブンブンブブーンブンブブーン、ブンブンブブーンブンブブーン」

少女はエンジン音の口真似をしながら薄明のヒガシ＝オオキイ＝メインストリートを駆けていく。と、体を傾け路地の一つに駆け込むや彼女の水色の髪の一部が触手めいて伸び、防水リュックから取り出した配達物を郵便受けに叩き込む。

「イヤーッ！」「イヤーッ！」「イヤーッ！」

叫び声の源は住宅街に紛れていった。キョート＝サン市街が活動を開始し、λ文字山から日が昇らんとしているのがわかる。ヒガシヤマ＝マウンテン連峰の向こう側、まだ暗い夜空に聳え立つ円柱状の構造物が水平線下からの太陽光を散乱して、いち早くキョート＝サン市街に光源を届けているからである。

これこそ、納紀元年に完成し、世界のなりたちを根本から変えることになった発明、余剰次元エンジンであった。

この物語はフィクションであり、冒頭から末尾まで、登場する人物・組織・団体・事象は実在および架空のそれらとは一切関係ありません。あなたがどこかで出会ったことがある誰かや何かが登場するかもしれませんが、それは他人の空似、同姓同名の別人、一卵性および二卵性双生児、生き別れの義理の妹、メイドインなんちゃらの模造品等でございます。あしからずご了承ください。

25.1　侵略者

「お断りします。必要ありません」

夜に起きつづけることも、昼過ぎまで寝つづけることも俺の自由だ。laziness の鍛錬に励む俺にとって、規則正しい生活というのは、ダサい設計をカバーするために重宝される ピー 言語のコーディング規約と同様の価値しかない。

そんな俺の住居をわざわざ昼間から訪ねてくる者はおおむね二種類に分類できる。俺がブラウザを介して発信した要求を非常な低コストで叶えてくれる組織に雇われた善意の配達員か、販路、あるいは収益源、あるいは信者を拡大しようとする悪の組織の末端構成員たちだ。信頼すべき匿名掲示板の情報によれば、後者は無料で言論バトルゲームの相手になってくれる場合があるらしく、俺は「当り」を引くのを楽しみにしては肩透かしを食らっていた。が、今回のは早朝からお越しになったために、ベッドから寝ぼけた頭のまま瞬時に玄関まで移動。そこで初見のインパクトがでかすぎて、俺は咄嗟に定型句を発するのがやっとだったのだ。

「無下にトビラを閉めなくてもいいじゃなイカ！ 人類は参照透明な海を穢した自らの罪をきちんと知るべきでゲソ！」

「うわ気持ち悪いなこれ、扉に挟まっても動くとかよく出来てる、中にワイヤーとか入ってんのか？ つか知ってるよ！ コスプレイヤーはコミケにお帰りください」

「これは地毛でゲソ！ それに問題はそこじゃないでゲソ！ 人類は闇の言語に毒されまくっているのでゲソ！ この本を読んで洗脳から解放され、参照透明な海を取り戻さなイカ？」

「だからそれ知ってる、俺もいっぱしの Haskeller だしな……あ？」

勢いよく扉を開けてしまった俺の視野には、イカ娘がいた。目を見開き、円弧状の眉、半開きの口を持つ萌え絵としてのイカ娘が立体物として存在したのだ。

「そんなこと言って、体よく断るつもりじゃなイカ？」

もちろん。その本の青と肌色の配色、あれは間違いなく業界では知らぬ物のないアイテム。日本文化との出会いがもたらした奇跡、『関数型言語の薄い本』ではないか。

「今夜のイベントに持ってくつもりだったから、玄関に用意してある。ほら。ちなみにこっちが俺のツイッターアーアイコン」俺はイカ娘が持ってきたのと同じ本に、自分のアイコンを載せてみせた。

「この意匠は確かに Haskell 公式ロゴインスパイア……使い手がこんな所にいたでゲソか。」

と、彼女はあの不敵な笑みを取り戻し、

「それは好都合でゲソ。が、今はまだ何も話すわけにはイカんでゲソ」

ワンピースをひるがえし、

「また会うでゲソーーー」

マンガめいた速さで去っていった。

俺は少しだけ後悔した。いくらリアルがクソゲーだからって、自分があまりにも画面の中の世界に親しみすぎていて、どうみてもレンダリングがおかしいオブジェクトと普通に会話してしまったことに。

「どうやら、ここはもう京都じゃないみたいだな」

人間は場面転換めいてベッドから玄関へ瞬時に移動したりはしない。ツイッターアイコンはデジタルデータであって名刺のようにハンドルできるものではない。なにより俺の家からは、山越しにあんな塔が見えたりしない。俺の知っている京都市は、建造物の高さに厳しい制限を課していたはずだ。高層ビル群の圧倒的質感を背に感じながら、

「二次元（フィクション）の世界に来て最初の会話、もっと楽しめばよかった……」

25.2　ネオオオサカ炎上

「ちょ www お前なんでここに来てるんだよ www おれの惜別返せ www」

「こういう場所は意識の高い人間が多いから布教に最適じゃなイカ」

俺たちは、Haskell 勉強会が開催されるネオオオサカのスレイプニル＝ピラーで、机を並べていた。イカ娘は新客がくるたびに例の本を売りつけようとしていたが、態度の問題か一向に売れない。

「で、その本はそんなに大量に、どうやって手に入れたんだ」

「それは……記憶にないのでゲソ。気がつくと、家に一人でいて、覚えていたのは、この書物に書かれてい

ることを世に広めねば、という使命感だけだったのでゲソ」
「やっぱお前、宗教の押し売り屋めいているぞ？」
　突然、ガラスが割れる音がし、何かが目で追えない速度で視界を横切り、部屋の間仕切りに突き刺さって止まった。
「なんだあれ？　ムカデ？」歪んだ金属板の隙間から棒やモーター、導線を露出させたそれは、30cm くらいの細長い体をよじり、もがいていた。
「アイエエエエ！」「モジモジシテキモイネー！！」「アイエエエエエ！」勉強会に参加しようと着座していた一般人が一斉にパニックを起こす！
「こっちでゲソ！」「アイエエエエ！」「再帰コワイ！　再帰ナンデ？」「アイエエエエエ！」「バグの温床だ、皆殺られるぞ！」「ついてくるでゲソ！」俺の手を引き、殺到する市民を掻き分け非常階段に達するや、イカ娘は俺の体を触手の 2 本で包み込み、残りの触手を使って階段中央の空間を猛然と登りはじめた！
「なんとか屋上まで逃げ延びたでゲソ」
「むしろお前の拉致行為に恐怖を感じたよ……つか外が大変なことになっている件について。下は火の海だし、なんか虫っぽいのいっぱい飛んでるしやばくね？」
　今いる屋上より高い建物も多く、周囲の様子はよくわからない。何者かが応戦しているらしく、ビルに区切られた窓を機銃掃射やビームが交錯しているが、天を覆う虫どもは刻一刻と数を増していく。
「思ったより早く、腹を割って話せる時が来たようでゲソね……私は海からの使者、イカ娘。そして、バグと戦う正義の組織、リ＝カージョンのメンバーなのでゲソ！」
「嘘だろ w」どこで吹き込まれた設定か知らないが、渾身のドヤ顔が痛いぞイカ娘。
　と、その背後、炎上するビルを舐めるように、巨大な鋼鉄の飛行甲板が、それを支えるトラス構造が、炎と煙をかきわけて丸みをおびた艦首が押し出されてくる！　激しくはためくイカ娘のワンピース。強烈な空気の滝が煙と炎を吹き払い、マンガめいた大きさのターボ・ファンを左右に備え飛行する装甲と砲塔の塊をあらわにした！
「嘘……だろ……」
　《おいお前ら、早うせい！》拡声器を通じた幼い女性の声。
「言われなくても、でゲソ！　M-1（mike one）、モーフィズム許可を求む！」イカ娘は鉄塊の方に向き直って叫ぶ。
　《M-1、モーフィズム承認》ふたたび鉄塊の方からこんどは、女性の落ち着いた声が答える。
「あと、VIP 一名様確保、回収を任せるでゲソ。さあ、行くでゲソよ！」

「スペシャ☆ライズ↑↑カタモーフィズム!!」

　かけ声とともに、イカ娘の足元から発した強烈な光条があたりを包み、どこからともなく勇壮ながら可愛い音楽が流れだした。
「なんだこれは、あの戦艦から出てるのか？」
　光の渦の中心にいるのはイカ娘だ。彼女のワンピースは最初の衝撃で四散してしまい、つまり彼女はいまイカ帽子を除いて全裸なわけだが、肝心の箇所だけは輪を作って高速回転する光の粒子に妨げられて見えない！　彼女が空中でくるくる旋回しながら舞い、体の部位を叩けば、光の粒子はイカ娘の指示に従うかのように右手に、左手に、そして両足に蒸着し、新しい衣装を形成した。仕上げとばかりに頭部と触手をまとめ上げた光がポップな効果音とともに弾けると、毛繊維の束というよりは弾性体の塊のように跳ねる、ボリュームたっぷりの新しい髪が出現する。
「って、中の人つながりかよ！」
　この黄色を基調とした短めのドレスに、たわわに実ったバナナの房にそっくりな髪型、あからさまにキュアピースなのだ！
　ふしぎなおどりを終えたイカ娘は、軽くジャンプすると、

「あたしの戦いっぷり、よく見ておくでゲソォォーーー」
　ドップラー効果がかった台詞とともに夜空を駆け上がっていった。俺の方はというと、
　《お前さんも来るのじゃ！》「▁▃▅▆▇▒▒▒('ω')▒▒▒▇▆▅▃▁うわあああああああ」

謎のビームにつまみあげられ、戦艦に開いた入り口へと吸い込まれていった。

＊　＊　＊

トンネルを抜けると戦艦のブリッジであった。といっても俺は戦艦など乗ったことはない。アニメをよく見ているからよく分かるのだ。

「航空戦艦グラスゴーへようこそ、じゃ。客人よ。」

頭上から、白い衣をまとった、というより白い布の塊に埋もれた幼女が話しかけてきた。こいつはロリババアキャラなのだろうか。高いところに座っているところを見るとたぶん艦長、あるいは高いところが好きな子なのだろう。あとのオペレータ娘たちはクローンなのか量産品のメイドロボか、互いの容貌は見分けがつかないほどよく似ており、一糸乱れぬ集中でコンソールに向かい仕事をこなしていた。

「余剰次元観測および空中管制システム、起動」

「M-1 の識別信号を確認、パイロットおよびスーツともに正常運用中」

艦の周囲を映していた全面ディスプレイに EMACS という頭文字がつかのま浮かび上がったかと思うと、いくつもの大きなウィンドウが展開。分割された画面のそれぞれに、炎に照らされた市街と立ち昇る煙、虫のように飛び回る無数の小型機「バグ」、夜空を圧して浮かぶ二体の影、そしてバナナめいた髪と衣装のフリルをはためかせ飛行するイカ娘などを映し出す。彼女の周囲にもいくつかのホログラム EMACS 画面が展開しているようだ。

《これが、魔法少女になった私の『スキル』でゲソ！》水平飛行中の彼女を横から捉えたカメラ映像の中で、イカちゃんがそう言いながら手を前に伸ばして EMACS を操作すると、艦橋側でもイカちゃんの映像にカーソルが当たり、短いコードによる説明が現れる。インターフェイスは双方向のようだ。

```
module Data.List.Catamorphism where

cata :: b -> (a -> b -> b) -> [a] -> b
cata b (<+>) = h where
  h []     = b
  h (a:as) = a <+> h as
```

隅の画面に映るのはネオオオサカ名物の蟹料理チェーン店。1 階の車庫から続々と多脚ロボットの群れが滑り出す。画面に字幕が踊る「メインバトルカニ MBK-Z04 マシンズワイガー」。降り注ぐビーム攻撃を横滑りで巧みにかわしながらバグに対して銃撃を加える。さらに蟹料理チェーン店の看板である巨大カニ像も爆砕ボルトを切り離し大地に立つ！ 巧妙に偽装された人類側の防衛戦力だったのだ！ 画面に字幕が踊る「メインバトルカニ改善 MBK-T05 マシンタラバー」。カニ型ロボットは多脚歩行にローラー移動を組み合わせることで舗装道路の高速走行と瓦礫等の踏破性能を両立した都市防衛用の新兵器なのだ！ しかも複合装甲材はカニ殻樹脂を再利用だから環境にも実際優しい！ 人類対バグの地対空戦闘が始まった！

小型のバグの中には撃墜されていくものもあるが、二体の巨大兵器からはなお無数のバグが吐き出され続けている。二体のうち上空を飛んでいる方は漆黒の胴体のあちこちに赤や緑の輝点の列を目まぐるしく明滅させている。下を飛んでいる方の機体は同じく漆黒色で、胴体下部を起点に多層のバリアを伴って移動している。人類側の猛攻撃はバリアの各層に無数の徒花を咲かすのみで、二つの巨体やその付近のバグには届いていない。と、バリアの 1 枚が急に収束し輝きを増したかと思うと、各画面が異なる角度から夜空を走る閃光を捉え、唯一ホワイトアウトが続いた隅の画面はやがて、一直線にえぐられた道路と、カニ型ロボット群の成れの果て—脚部が残らずひしゃげたり、地面に潰れたり、めくれ上がった弾薬庫から断続的に炎を上げる無残な姿を映しだした。ビルの陰から破壊を免れたロボットがすかさず滑り出し、パイロットの救出作業をかばいながら攻撃を再開したが、その射撃音は以前よりもずっと頼りなげに聞こえてしまった。

「マシンズワイガー・デルタ中隊戦闘不能！ マシンタラバー隊、ミドースジ＝ラインまで後退します！」オペ娘が矢継ぎ早に報告する。

「今ので大型標的の行動事例収集度が 23 パーセントに達しました」とオペレータメイド。また 1 つの EMACS 画面が開き、ログ文字列が流れ始める。俺の目にはかろうじて部分的な文字列が認識できた。

```
ghci> length []
0
ghci> let bullets = ["弾弾玉弾弾","弾弾弾 tama 弾弾弾","弾弾偶々弾"]
ghci> length bullets
3
ghci> map (filter 当たってもどうということはない) bullets
["玉","tama","偶々"]
ghci> let tanks = [mbkz,mbkz,mbkz,mbkt]
ghci> length tanks
4
ghci> filter (not . 敵性) tanks
[]
```

マトリックスめいた緑色ログ文字列の奔流を瞳に映していたもう一人のオペ子が宣言する。「標的の動力源（ソース）推定結果、出ます。識別仮称を設定、災機獣レングズ、および災機獣ヴィルダーン」

```
module Data.List.LengthFilter where
import Prelude hiding (length, filter)

length                  :: [a] -> Int
length l                =  len l 0
  where
    len :: [a] -> Int -> Int
    len []     n = n
    len (_:xs) n = len xs (n + 1)

filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
filter pred (a:as)
  | pred a         = a : filter pred as
  | otherwise      = filter pred as
```

「あの……、何してらっしゃるんですか？」急展開につぐ急展開に思わず敬語になる俺。魔法少女、戦艦、それに EMACS だって？

「陽動に対する反応から敵の行動パターンを分析しておるのじゃ」ロリババア艦長が察して説明セリフを補う。「災機獣に対して通常攻撃では時間稼ぎにしかならぬ。砲撃で空けた風穴が塞がるのもあっという間、たとえ群れのほとんどが倒された状況からも、一枚の装甲板でも残っておれば群れ全体が再生しおるのじゃ。災機獣のボディはすべて、ある核となる共通構造から構成されており、それが驚異的な汎化性能をもっておるからじゃ」

「災機獣を完全に葬るには、その共通構造を見ぬいて一撃で無力化するほかないのじゃ。魔法少女はその媒介。して魔法少女を要とする窓から実装力を投射するためのシステムが『イーマックス』、Extradimensional Monitoring and Airborne Control System というわけじゃ」

「ビームは通用しない相手、というわけか」

「わからんぞ？ 海の向こうでは武威派がブイブイ言わせているとも聞くがのう。エクメイド・クリスタよ、分析結果はどうじゃ」

呼ばれたオペレータが対応する。「レングズは敵性物体の数を正確に捉える感覚器を備え索敵を担当、ヴィルダーンは対象の属性ごとにそれを通さないバリアを張る能力を持ち、防御および攻撃手段としている模様です」

「厄介な組み合わせじゃな。遺本ライブラリに対処法の預言は？」

「検索結果、旧約甘蕉書の 3 ページに該当、表示します」また新しいウィンドウ。
「よし、戦術長、解読を頼む！ って、戦術長はどこじゃ？」無人の戦術座席に漫符が点滅し、マヌケな効果音が鳴る。
「戦術長なら長期休暇中です。お預かりしているメッセージは一件です：『人類の未来よりコミケの原稿の締切り重点。オタッシャデー』」
「なにいいい！ こんな大事な時にあやつは何をしておるのじゃ！ 人類が滅んでしまえばコミケも何もなかろうに！」
駄々をこね、ぶかぶかぬののふくを揺らした艦長はふと気づいてこっちを見る。
「のう客人、お主これを読めるか？」
「何を？」
「これじゃよ」艦長がジェスチャーすると、小さな指からホログラムの光線が艦橋空間を横切って『旧約甘蕉書の 3 ページ』の枠線を点滅させる。

りすと的**かたもるふぬずむ**を産すの事

祖あだむ＝にんじやは弓手に $b \in B$ を持ち馬手に $\oplus \in A \| B \to B$ を掲げ給ひて**りすとかたもるふぬずむ** $h \in A* \to B$ あるべしと宣せ給ふ

$$\begin{aligned} h\,\mathtt{Nil} &= b \\ h(\mathtt{Cons}(a, as)) &= a \oplus (h\ as) \end{aligned} \tag{1}$$

卽ち斯なりぬ◆このごろ都に流行る**はすける**語に於ては foldr とも稱すとぞ◆あだむ＝にんじや之を觀て善とし給ふ◆あだむ＝にんじや甘蕉形の喝鈷にて**かたもるふぬずむ**を表すの沙汰定めおかせ給ひぬ

$$h = (\!|b, \oplus|\!) \tag{2}$$

りすとを引數に取る關數**かたもるふぬずむ**にみな從ひて其數限りなし◆例を擧ぐるに先の征夷對象羣源**れんぐす**頼朝及び數々の名句を殘せし二天流宗家宮本**ふぬるたー**正志是皆**かたもるふぬずむ**の一族なり

$$\begin{aligned} \mathtt{length} &= (\!|0, \oplus|\!) \quad \text{where} \quad a \oplus n = 1 + n \\ \mathtt{filter}\ p &= (\!|\mathtt{Nil}, \oplus|\!) \\ &\quad \text{where} \quad a \oplus as \\ &\qquad\qquad \mid \ p\ a = \mathtt{Cons}(a, as) \\ &\qquad\qquad \mid \neg p\ a = as \end{aligned}$$

再歸關數を手ずからに實裝せずして**かたもるふぬずむ**に歸依するこそ**ばぐ**を除き災鬼を祓ふ術なれあなかしこ

ウィンドウには古びた本のページめいたものが映し出されている。艦橋の皆がこっちに注目している。時間の流れが止まったかのような錯覚。俺に何をしろと言うんだ？ 戦闘シーンを流す大画面、複数の EMACS ウィンドウに映し出された Haskell コードの断片、一つだけ場違いな古文書……この文書の言語は日本語を基調としているようだが、使われている記号の意味が、それ以前に文意が全くわからない。いや、待てよ、まさかこの状況——

cata という関数を使って、
length と filter を実装しなさい、というクイズ……!!

艦長がジェスチャーめいて手で合図すると、ホログラムのキーが飛来し、馴染んだキーボードの形を構成する。戸惑いつつも普段どおりに手を置くと、
「あっ自然に打てて、あっすっごい楽だ……これは持って帰りたい」
「生体認証と拡張現実を組み合わせた最新鋭の UI じゃ」

そうか。落ち着いて見比べれば、災機獣の動力源（ソース）、と船員たちが呼んでいる物は Haskell における `length` と `filter` の実装だ。行動事例（サンプル）はまさに動作のサンプル。そしてイカちゃんの『スキル』は、`b` 型の初期値と `(<+>) :: a -> b -> b` という二項演算を引数に取って、`a` 型のリストを `b` 型の値に畳み込む `cata` という関数の定義。引数の順序は変わっているが、`foldr :: (a -> b -> b) -> b -> [a] -> b` と同じだ。古文書のほう、言いたいことはさっぱりわからないが、(1) 式は関数 `cata` の定義に酷似している、そして (2) 式によれば `cata` のことは魔法少女イカちゃんのバナナ髪を形どった (|…|) という括弧で表現するらしい。後の部分は、そのバナナ括弧を使って `length` と `filter` を作る方法が謎めかして書いてある！ 古文書の (|…|) で囲まれた部分を `cata` の関数呼び出しに置き換え、残りは逐語訳すれば、何とか意味の通った Haskell になりそうだ。

```
module Data.List.LengthFilter.Catamorphic where

import Data.List.Catamorphism (cata)
import Prelude hiding (length, filter)

length :: [a] -> Int
length = cata 0 (<+>) where a <+> n = 1 + n

filter :: (a -> Bool) -> [a] -> [a]
filter pred = cata [] (<+>)
  where x <+> xs
          | pred x          = x : xs
          | otherwise       = xs
```

「うむ、さすがはイカ娘が見込んだ客人じゃ、見事ぞ。検証班！」
指名されたオペ子が、一瞬戸惑いをみせたのち、
「テストコード実装完了」
ガトリングめいた高周波タイプ音を 1 秒にも満たないつかの間響かせると、cat コマンドに匹敵する速度でソースを表示させた。

```
module ListCataSpec where

import Test.Hspec
import Test.Hspec.QuickCheck (prop)
import Test.QuickCheck.Function (apply)
import Test.QuickCheck.Poly (A)

import qualified Data.List.LengthFilter as X
import qualified Data.List.LengthFilter.Catamorphic as Y

spec :: Spec
spec = do
  describe "list catamorphism" $ do
    prop "can reproduce the length function" $ \xs ->
```

```
    X.length xs == Y.length xs &&
    Y.length xs == Prelude.length (xs::[A])

  prop "can reproduce the filter function" $ \fun xs ->
    let f :: A -> Bool
        f = apply fun in
    X.filter f xs == Y.filter f xs &&
    Y.filter f xs == Prelude.filter f (xs::[A])
```

「QuickSpeck テスト結果、Passed？ 本当にこれで行くんですか？」オペ娘はいぶかしげだ。
「魔法雷撃戦、パッチ用意！」艦長は号令をもって回答！
「「了解」」となればメイド達は即座に作業にかかる。艦橋メインディスプレイにめまぐるしくウィンドウが開閉し、満ちるタイプ音、飛び交う連絡、報告。《ビワコ１号機、ワーデンクリフ回路に接続完了》「送電を開始してください」暗雲垂れ込める空と山々を映し出していた監視カメラの一つの輝度が突如増大！ 余剰次元エンジンの特徴的なシルエットから、夜空に光の柱が立ち昇っているのだ。
《ゲソッ?!あの技は、あ、ちょっとやめなイカ！》
「何を言う、もう『予算はついた』のじゃ。あとは魔法少女であるお前がカロウシしてでも仕様書通りに実装せんと、無制御の余剰次元エネルギーが無駄になるんじゃぞ？ 成せば成る！」
《うぁあああああああん！ ヤバレカバレでゲソ～～！》
黄色イカ娘は涙目ながらも両手に V サインを作り天に掲げる！ 水平に１００ｋｍを隔て、電離層と大地とが形成する電路にそってイカ娘に落雷！
《くうぅぅぅ、どうせなら電気じゃなくて炭火で美味しくいただいてほしいでゲソ～～！》
膨大な電荷を蓄えたイカ娘は触手の１本１本が反発して球状に広がり、センジュカノンめいた形相を示している！ 雷を通じて流れ込む電気エネルギー量はなおも膨張！
《はやくレビューするでゲソ～～》イカ娘は電撃の痛みに痙攣しながら叫ぶ！
「パッチ内容 bugfix：レングズおよびヴィルダーンを revert、レビュー完了、現実ブランチにプッシュしてください」オペ子の一人が応答！
《よっしゃおらあああ、イカイカいかりん、これでも喰らえええええでゲソ！ **かたもる・ピー**スサンダー！！》
イカ娘は両手を前に突き出し、溜めに溜めた雷撃エネルギーを開放！ と、極太雷撃はまず上下に分裂、ついで無数の枝分かれと合体を繰り返しながら、画面奥、災機獣の方へと驀進してゆく！ その先頭部は次第に 2 体の巨獣、レングズとヴィルダーンの容姿をとっていくではないか！ ヴィルダーンは地対空攻撃に抗して下向きに張っていたバリアを俊敏に張り替えるが、雷撃側も即座に、いやまったく同じタイミングで雷をまとったバリアを生成する！
夜空を駆ける放電塊の黄色にゆらぐバリアが、天を圧する闇の巨体の薄桃色のバリアと接触したかと見るや、あれほど堅固そうに見えたバリア面を雷撃塊はまったく勢いをゆるめず貫通！ いや、ゴム袋のように裏返りながら、バリア面に張り付き、つづいて二体の災機獣をぴったりと覆ってゆくのだ！ さらに被覆が完成した瞬間、あれほどの巨体が、描き忘れたかのように忽然と姿を消す！
一瞬の静寂の後、災機獣たちが消え去った場所よりも奥の空間で爆発！ 対消滅エネルギーがしばらく余剰次元を走ったあと開放されたのか？ さらに爆雲から電荷が逃げ場を求めて次々に放電が発生！ 近くのバグを次々に巻き込んで連鎖爆発！
やがて一連の爆発により加熱された大気が放出された微粒子を巻き込んで黒々とした乱流雲が発達し、無数に飛び交っていたバグの編隊はあるいは落雷に串刺しにされ爆発、あるいは主を探して黒雲に何度も飛び込むうち吸気系をやられたのか、開口部から火花を散らし煙の尾を曳き、ネオオオサカ湾をハナビ大会めいて明るく染めながら破片の雨となって降り注ぎ消えていった。

＊　＊　＊

「ゲーソゲソゲソ！ 我は海からの使者、イカ娘！ 金色の衣をまといて、参照透明な海に降り立つ者なり！」痛みはすっかり去ったのか、航空戦艦グラスゴーの飛行甲板に誇らしげに立ちポーズを決めて、イカ娘が名

乗りを上げる。
「いろいろきわどい。その発言も、金色の衣とやらの破れ具合も……」
　焼け焦げた衣装が辛うじて包む彼女の肢体の、お尻とふとももに切り取られた空間から余剰次元エンジンの勇姿が覗いていた。
「ところでお主、断片的とはいえ旧約甘蕉書をいきなり解読してみせた機転、なかなかのものじゃった。それにイカ娘の報告によれば何かおもしろい物を持っておるそうじゃな。機内持ち込みは手荷物を改めさせてもらうぞ」

25.3　預言の書

　グレイス准将と名乗ったロリ艦長は、艦内の書庫とおぼしき部屋に俺とイカ娘を連れ込んだ。艦体の構造に合わせたものか横に長細い部屋で、入り口からみて左右には書物を満載した棚の列が闇へと続いており、正面には簡素な机と EMACS 画面があった。二人に席をすすめたあと自分も着座した艦長は、俺の取り出した本を見るや興奮をあらわにした。
「これはウ＝ス遺本！　しかも現代人の手になるコピー本などではなく、古代に作られた真正写本の一冊じゃ！」
「ウ＝ス遺本って、なんなんですか？」元の世界でも聞いたことがある言い回しだ。嫌な予感しかしない。
「『東の島』。ジパングの東に位置し、かつて世界をも創り出す力を具した神々が数十万柱と集い、冒涜的な狂宴を繰り広げたという伝説の島じゃ。宴は百代、千代と続いたそうじゃが、今の世界を創り出したアダム＝ニンジャによって島ごと海の底に没せしめられたという。これがファースト＝インパクトじゃ。古事記にもそう書いてある」
「まて、いろいろおかしい」「古事記にそう書いてあるなら仕方ないでゲソね」
「その東の島からサルベージされたお宝、それがウ＝ス遺本じゃ。ウ＝ス遺本という一つの名は、旧支配者階級によって記された書物の総称。すごい H な本から技術書まで、全ジャンルを網羅しておる。現代のオーバーテクノロジーは、すべてウ＝ス遺本の記述を元に復元された旧文明の技術なのじゃ」
「そう……なのか？」「ウ＝ス遺本に書いてあるなら仕方ないじゃなイカ」
「サルベージされるウ＝ス遺本には、腐っている本や原典が散逸してしまっている本も多い。そういう場合は、より後の時期の写本や、別の書物に現れる引用箇所などを手がかりに、原典を復元する文献学研究が欠かせないのじゃ。これまでに見つかってないバージョンの写本が出土したとなれば、この上もなく貴重な手がかりじゃ。とくにその本」
「アッハイ」「そうでゲソ！　私が布教しようとしている本でゲソ！」
「古典学の分類において『簡約聖書』と仮称されるアンソロジー、それに収められた一篇に間違いない。次元エンジンの完成による新時代、納紀の到来。人類の天敵『バグ』の侵略。すべては簡約聖書に預言されてあったことじゃ」
「λカ娘　(算) の伏線回収したぞおぃい？」
「なんのことでゲソ？」
　グレイス艦長は不意に黙り込み、何かをタイプしはじめる。ややあって俺の目の前 1 m ほどの空間にホログラム文字列が出現した。イカちゃんが無反応なところを見ると、この文字列は私だけに見えているらしい。
　《まずは安心してほしい。わしも『外の世界』からのまれびとじゃ。目的はこの世界の物理法則の理解。そして、もといた世界に帰ることじゃ》
　ハッとして、俺はグレイス艦長の顔を見つめる。こいつ、どういう意図があってこんな発言を？　俺の目線による問いかけに答える代わりに、艦長は自分のホロ・キーボードを操作して、1 つのウインドウを壁面ディスプレイに呼び出した。
「これは、論文の表紙？」
「『旧約甘蕉書』—経典本文の最初の単語 banana からそう仮称されておるウ＝ス遺本じゃ。災機獣に関する数多の預言書の言及をひもといてゆくと必ず行き着く文献ゆえに、災機獣創生の秘密を記した原典と見て間違いない。今後の展開もこのシナリオ通りに進む可能性が高かろう」

「見よ、旧約甘蕉書の 11 ページじゃ」

$$(\!|\varphi|\!)_{\mathrm{F}} = \mu(\varphi \overset{\mathrm{F}}{\leftarrow} \mathtt{out}) \tag{12}$$

$$[\![\psi]\!]_{\mathrm{F}} = \mu(\mathtt{in} \overset{\mathrm{F}}{\leftarrow} \psi) \tag{13}$$

$$[\![\varphi, \psi]\!]_{\mathrm{F}} = \mu(\varphi \overset{\mathrm{F}}{\leftarrow} \psi) \tag{14}$$

$$\{\!|\xi|\!\}_{\mathrm{F}} = \mu(\lambda \mathtt{f}.\xi \circ (\mathtt{id} \triangle \mathtt{f})_{\mathrm{F}} \circ \mathtt{out}) \tag{15}$$

「なるほど、全くわからん」「わからないでゲソ」

「わしにも読み方すらわからん。じゃが、周囲の文脈から推理するに、この四行詩こそ、四つの喝鈷（かっこ）**かたもるふゐずむ、あなもるふゐずむ、はいろもるふゐずむ、ぱらもるふゐずむ**を携えた四人の戦士が、四頭の災厄の獣を打ち倒す最終戦争『**すくい乞（こ）る**』、その経緯を記した旧約甘蕉書の核心とみて間違いない。この部分を早急に誰かが解読せねば、世界は破滅じゃろう」

「んな大袈裟な。ノーヒント、即死プレイっすか？」

「手がかりならあるぞ。四世紀ほど後代に著者の子孫が書いた新約甘蕉書、さらに後代になって書かれた注釈書『ロゼッタストーン』などが発掘されておる。それらと照らし合わせれば――」

その時だった、艦内放送で警報が鳴り響いたのは。

「なんじゃあ?!」

《余剰次元振動が観測されています》

「新手の災機獣でゲソ？」

《固有重力波形、マッチドフィルター・アーカイブに該当確認。識別名称……》――音声はそこで一旦途切れ、メイドさんにしては珍しく戸惑いを含んだ調子で続く――《レングズ、およびヴィルダーン?!》

「なん……じゃと!!」

「確かにこの手で倒したはずでゲソ！ 話が違うじゃなイカ！」

「総員、戦闘艦橋へ急ぐのじゃ！ M-1 はカタパルトへ！」

25.4　強敵・災鬼獣

艦橋ではすでにメイドたちが各種機器を立ち上げはじめており、EMACS モニターが先ほどの戦闘の傷跡も生々しいネオオオサカ市街を映し出している。グレイス艦長が艦長席への梯子を登る間にも、モニターはネオオオサカ湾、さきほど無数の破片が降り注いだ辺りにうごめく何かの姿を捉える。海面に波浪を拡げながらみるみる成長していくそれは、植物の蔓のようにも見える無数の触手の塊だ！ 触手塊は呼吸するように何度かの膨張と崩落を繰り返したのち、一気に天を衝いて伸び、二つの繭状物体を作り出す。繭は左右に分かれて回転しながら、地上に残した部分を巻きとってゆく。やがて回転が最高潮に達すると二つの繭はほぼ同時に破裂して破片を撒きちらし、横回転する二つの巨体を現す！ 巨体は横回転を制動すると、どっしりとした水平航行にうつった。宙を舞う生体破片も、巨体から垂れ落ちる汚滴も、スローモーションのように重力に抗い、やがて意志を持って飛行し始める。無数の突起や触手に被われ、生物めいた外観を呈してはいるが、そのシルエットは確かに、さきほど交戦したばかりの……

「「災機獣！！」」

「うろたえるな！」艦長が一喝。応えるかのように、さきほどから聞こえていた複数のエンジン音が一段と力強さを増す。ネオオオサカの街のそこかしこから、フグ屋の看板に偽装されていた武装飛行船、焼タコ屋の看板に偽装されていた回転翼機、オコノミ=ガレット屋の看板に偽装されていた円盤飛行機などが次々に飛び立っているのだ。「こんなこともあろうかと、何らかの異常が観測されたらすぐに航空部隊が発進する手筈にしておいたのじゃ」

「通常の兵器では歯が立たない設定じゃなかったのか？」

「大丈夫じゃ！ さっきの戦いは行動事例収集（サンプル）が足りんかっただけじゃろう。それに、奴等の特性について分かったことがある。奴等は一直線上に並んだ対象しか認識できない。無人機を複雑な編隊を組んで展開させ

れば充分な行動事例収集が可能じゃ！」

確かに、一見飛行特性すらバラバラの寄せ集めに見える航空部隊は、それゆえに同種の機体三、四機づつのまとまりがあるいはゆったりと、あるいは素早く災機獣をとえらたカメラの前を横切り、一種の撹乱効果をもたらしているかに見えた。が、その時！

ジグザグ編隊飛行していたオコノミヤキが一直線にならんだ瞬間、レングズの体表面から一本の触腕が猛然と伸びる！ その表面はびっしりと眼球で覆われてマジキモーイ！ そしてヴィルダーンの収束バリア攻撃がオコノミヤキ編隊を貫通！ EMACS 画面にログが表示される！

```
ghci> okonomiyakis
Cons 牛玉 (Cons 豚玉 (Cons 海鮮 (Cons ミックス Nil)))
ghci> length okonomiyakis
4
ghci> filter (not . 敵性) okonomiyakis
[]
```

さらにふわふわと浮遊するフグ、足を高速回転させながら斜め上昇に転じたタコ、水平飛行してきたイカ娘が密集したところを見逃さじと、レングズの体表面から触腕が伸び——分岐を繰り返しながらたちまち樹状生体構造を形成！ その樹状腕の末端に眼球がいっせいに見開く！ オニキモーイ！ そしてヴィルダーンのバリア攻撃が炸裂！ 今度のバリア攻撃は飛来しながら二分裂を繰り返し、空中に散開したそれぞれの機体に正確に命中！ EMACS 画面に表示されたログがそれを裏付ける！

```
ghci> fish
Branch (Branch (Leaf tako) (Leaf tako))
       (Branch (Branch (Leaf ikmsm) (Leaf fugu)) (Leaf fugu))
ghci> length fish
5
ghci> filter (not . 敵性) fish
[]
```

イカ娘は大きく吹き飛ばされながらも、バナナ型の髪を一杯に拡げ、空気ブレーキで踏みとどまり、
「よくも海の仲間たちを！ 二回目なので必殺技バンクは省略するでゲソ！ **かたもる**・ピースサンダー！！」

果敢にも怪物に立ち向かっていくが、鋭い直線状の雷撃は甲高い音とともにあらぬ方向に弾かれ、傷ひとつつけることができない！ EMACS モニターに戦況分析が表示され、担当官が要約して読み上げる！

```
Couldn't match type 'Rec f0'
              with 'a0 -> (f0 a1 -> a1) -> f0 a1 -> a1'
Expected type: (f0 a1 -> a1)
               -> (a0 -> (f0 a1 -> a1) -> f0 a1 -> a1) -> [a0] -> f0 a1 -> a1
  Actual type: (f0 a1 -> a1) -> Rec f0 -> a1
In the expression: cata
```

「型エラーです、**かたもる**・ピースサンダーが通用しません！」

あっという間に此方の航空戦力を一掃してしまった新レングズは、体中のいたるところフキデモノめいて不規則に並んだ眼窩を盛んにぎょろつかせ全周囲を睥睨！ 新ヴィルダーンは、イカ娘の攻撃など痛くも痒くもないと言いたげに、フィールドを収束させると、ようやく反撃を開始したネオオオサカ市街に飛び込ませる！ だが見よ、今度のバリア攻撃は、大通りを一掃するのみならず、交差点に至るたびに分裂し、路地にも侵入していくではないか！ 艦橋カメラは次々に切り替わり、吹き飛ばされていくカニ型ロボット、巻き込まれる乗用車やトラック、なぎ倒される街路樹を執拗なまでに映し出す！

「マシンズワイガー隊、マシンタラバー隊全滅！ 威力と精度が違いすぎます！」

「ここまでの進化を遂げるとは……これはもはや災機獣などではなく、災鬼獣とでも呼称すべき存在じゃ。活路は一つ、かたもるふゐずむ喝鈷（かっこ）の撃ち手たるイカ娘の、レベルを上げて殴ればいい！」

cata を拡張することで List でも Tree でも引数に取れるような length や filter を作りさいというクイズ……!!

「そんな無茶な！ 旧約甘蕉書は読めたものじゃないし、リストもツリーも扱える再帰関数なんてどう書いたらいいのか想像もつかない。リストとツリーの値コンストラクタは全く別物じゃないか！」
「ちょうどデータベース化作業が完了した新約甘蕉書にも該当箇所があります。表示します」モニタにウ＝ス遺本のページであろうものが映し出される。
「だから無理……って、新約？ これはだいぶ Haskell っぽい言語で書いてあるな、読める！ 読めるぞ！」
　俺の意識はコードに強く誘引され、眼前の EMACS 画面以外のあらゆる感覚が彼方に飛んでいく。艦長の台詞がフェードアウトしていった。

「でかした！ 事態は一刻を争う、必要な箇所 だ け か い つ ま ん で 読　む　 ん ……

＊　＊　＊

「まずは新しい length や filter が引数にとっていた値の型を調べよう。

```
ghci> :t okonomiyakis
list :: List Object
ghci> :info List
type List a = Rec (L a)
```

```
ghci> :t fish
tree :: Tree Object
ghci> :info Tree
type Tree a = Rec (T a)
```

　なんだこれは？ List と Tree は、それぞれ L や T というものを基に、Rec から生み出されているのか。Rec、L、T はそれぞれ型引数を一つづつ取る型コンストラクタなのかな？ kind を調べてみよう。

```
ghci> :k Rec
Rec :: (* -> *) -> *
ghci> :k Rec (T Object)
Rec (T Object) :: *
```

```
ghci> :k L
L :: * -> * -> *
ghci> :k L Object
L Object :: * -> *
```

```
ghci> :k T
T :: * -> * -> *
ghci> :k T Object
T Object :: * -> *
```

　いや違う！ L や T は二つ型引数をとる型関数。L Object や T Object はそれを部分適応したもので、あと一つ型引数を取る型関数だ。そして Rec は『一つ型引数を取る型関数』をとって具体型を帰す……一体どうなっているんだ」
「新約甘蕉書によれば、リストやツリーは通常、たとえばこのように定義される：

```
data List a
   = Nil
   | Cons a (List a)
```

```
data Tree a
   = Prune
   | Leaf a
   | Branch (Tree a) (Tree a)
```

　が、以下のような定義も可能らしい：

```
data Rec f = In (f (Rec f))

type List a = Rec (L a)
data L a x
  = Nil
  | Cons a x
```

```

type Tree a = Rec (T a)
data T a x
  = Prune
  | Leaf a
  | Branch x x
```

「新約甘蕉書が言うには、L は確かに元の List の定義に似ているけど、L の定義の右辺には L が登場しないから、L は再帰的データ型ではない、このように再帰データ型を「再帰を使わないデータ」と「Rec による再帰」に分割して書くことができる、と。——って、L や T は、従来の定義のなかで被定義型自身を再帰的に呼び出していた箇所を x に置き換えただけじゃないか。何だか騙されてるような気がするな。いづれにせよこの Rec という構造が鍵であることは間違いない」
「それにしてもこの二つの定義はどうして等価なんだ？ 厳密さ軽視で書き換えてみるか。型シノニムの定義から始めて：

```
  List a                    | Tree a
= Rec (L a)                 | = Rec (T a)
```

Rec のデータ型定義を展開して：

```
  List a                    | Tree a
= In (L a (Rec (L a)))      | = In (T a (Rec (T a)))
```

最初の L や T の引数が二つ揃ったので展開して：

```
  List a                    | Tree a
= In ( Nil                  | = In ( Prune
     | Cons a (Rec (L a)))  |      | Leaf a
                           |      | Branch (Rec (T a)) (Rec (T a)))
```

Rec (L a) を List a と等価、Rec (L a) を List a と等価と思って置換すると：

```
  List a                    | Tree a
= In ( Nil                  | = In ( Prune
     | Cons a (List a))     |      | Leaf a
                           |      | Branch (Tree a) (Tree a))
```

「これは確かに再帰版のリストやツリーの定義に見えるぞ！ Rec (L a) を作った時、型コンストラクタ L の定義 L a x における二つの型引数のうち、第一引数 a はリストの中身の型となり、第二引数 x は再帰の起点の役割を担って消失するのか。
「改めて、Rec (L a) は a を要素に持つリストのことで、Rec (T a) は a を要素に持つツリーのことだと思って Rec を利用した定義を眺めると、

```
data Rec  f    = In (f   (Rec  f   ))   | Rec  f    = In (f   (Rec  f   ))
     Rec (L a) = In (L a (Rec (L a)))   | Rec (T a) = In (T a (Rec (T a)))
      List  a  = In (L a ( List  a ))   |  Tree  a  = In (T a ( Tree  a ))
```

値コンストラクタ In :: f (Rec f) -> Rec f が、Rec に渡されたファンクタ f を『一度利用して再帰データ構造の一段階を作り』『再帰の続きの段に渡す』という二通りの使い方をしていることがよくわかるな。だから Rec の定義の右辺には f、Rec、f がこの順で出てくる、ということなんだろう。」
「少しだけ分かって来たぞ。旧約甘蕉書の 10 ページで：

$$X_L = \mathbf{1} \mid A \| X$$

と書かれている部分は、ロゼッタストーンによれば | が和集合、‖が積集合を表すことから推測するに、全体を代数データ型の宣言と読み、| で区切られた二つの部分をそれぞれ値コンストラクタとし、その中で‖で区切られている要素を値コンストラクタの引数と読めばよい。適当に名前を補うと、こうなる！

```
data L a x = Nil | Cons a x
```

これは新約甘蕉書における L のことだ！ 同じく、旧約甘蕉書の 10 ページで直後にあたる

$$X_T = \mathbf{1} \mid A \mid X \| X$$

の部分は、| で区切られた三つのパート $\mathbf{1}$、A、$X \| X$ をそれぞれ Prune、Leaf、Branch とでも名づけた値

コンストラクタにすれば

```
data T a x = Prune | Leaf a | Branch x x
```

という代数データ型宣言になる。さらに旧約甘蕉書で Rec は μ と表記されているから、$(A*, \mathtt{in}) = \mu_L$ という記述は、Rec を用いてコンテナの型 $A*$=Rec (L a) と、その型に特化した型コンストラクタ In :: L a (Rec (L a)) -> Rec (L a) を作ることを意味する……一つのタプルの中に型と値が同居しているとは！旧約甘蕉書の時代にはまだ型という概念が未分化だったのか、型は基本的に集合、その型の値はその要素として表現されている。これも読みにくさの一因か」

「さて、cata の型を見てみよう。

```
cata :: Functor f => (f a -> a) -> (Rec f -> a)
cata phi (In x) = phi (fmap (cata phi) x)
```

cata の実装が何をしているのかはさっぱりわからないな。だが、ファンクタ f を L a や T a に特殊化すればいいことだけは分かっている。リストやツリーに特化した cata の型は、こうなる：

```
cata :: (L a b -> b) -> (Rec (L a) -> b)
cata :: (T a b -> b) -> (Rec (T a) -> b)
```

「今のイカちゃんのスキルである [a] に対する cata と、こんどの (Rec (L a)) に対する cata は、何が違うんだ？ 並べてみよう。

```
cata :: b -> (a -> b -> b) -> [a]        -> b  -- [a] の cata
cata :: (L a b       -> b) -> (Rec (L a) -> b) -- (Rec (L a)) の cata
```

形の上で、L a b の定義を展開してみよう。

```
cata ::  b   -> (a -> b -> b) -> [a]        -> b  -- [a] の cata
cata :: (Nil | Cons a b -> b) -> (Rec (L a) -> b) -- (Rec (L a)) の cata
```

Nil | Cons a b -> b という型シグネチャは— Nil | Cons a b -> b というのは Haskell の型じゃない、正確を期すなら Either () (a,b) -> b という型は —Nil が来ても Cons a b が来ても b 型の値を返せる関数を表している。これは二つの関数 Nil -> b と Cons a b -> b の組と等価だ。

```
cata ::  b       -> (a -> b -> b) -> [a]        -> b  -- [a] の cata
cata :: (Nil -> b, Cons a b -> b) -> (Rec (L a) -> b) -- (Rec (L a)) の cata
```

さらに Nil -> b は唯一の入力に対する答えを知っていればいいので、ただの b とみなしてよいだろう。

```
cata ::  b -> (a -> b -> b) -> [a]        -> b  -- [a] の cata
cata :: (b,   (a ,  b)-> b) -> (Rec (L a) -> b) -- (Rec (L a)) の cata
```

これは——！ [a] の cata のほうを uncurry 化すれば、二つの cata は等価になるじゃないか！

```
cata :: (b, (a, b) -> b) -> [a]        -> b  -- [a] の cata
cata :: (b, (a, b) -> b) -> (Rec (L a) -> b) -- (Rec (L a)) の cata
```

ついに分かった気がする。cata、あるいは Haskell の foldr は b0 :: b と (<+>) :: a -> b -> b を引数に取るが、これは [a] 型の値を取って b 型の値を返すような畳み込みを**一歩進める**のに必要十分な要素だったんだ。値 Nil が来たら値 b0 を返せばいいし、あるいは値 Cons a1 b1 が来たら、**自分の段以降の再帰は既に誰かが済ませて答えを** b1 **に格納してくれた**と思って、a1 <+> b1 を返せばいい。そして、その**再帰を済ませてくれる誰か** こそ cata に他ならない！」

「この『畳み込みを一歩進める道具』を、リスト専用の型 (b, (a, b) -> b) から、Rec で作れる一般のデータ構造に拡張したものこそが f a -> b！ そして

型シグネチャは言っている、
`cata :: Functor f => (f a -> a) -> (Rec f -> a)`
我に畳み込みを一歩進める関数 `(f a -> a)` を与えよ、
されば再帰データ型全体を畳み込む関数 `(Rec f -> a)` を返さん

——と。

これはまさに、`foldr` の一般化だ！」
「よし、何とか `length` と `filter` を書いてみよう。ぶっつけだ！

```
module Data.Rec.LengthFilter where

import Data.Rec (Rec(..), cata)
import Data.Rec.List
import Data.Rec.Tree
```

「もともとリストに対する length の型が `length :: [a] -> Int` であったこと、`[a]` が `Rec (L a)` と書き換えられることから類推するに、一般的な再帰データ構造 `Rec f` に対する `length` の型は `Rec f -> Int` になるはずだ：

```
length :: (Functor f, Lengthable f) => Rec f -> Int
length = cata lengthBase
```

「`Rec f` のベースとなるファンクタ `f` に、length の実装に必要となる再帰のパーツ `lengthBase` を提供してもらうんだ。`cata :: Functor f => (f a -> a) -> (Rec f -> a)` を `lengthBase` に適用した結果が `Rec f -> Int` になるには、`lengthBase` の型は `f Int -> Int` でなければならない。このような `lengthBase` を提供できる `f` の型クラスを `Lengthable` と名づけよう：

```
class Lengthable f where
  lengthBase :: f Int -> Int
```

「`L a` に対する `lengthBase` のインスタンス宣言は：

```
instance Lengthable (L a) where
  lengthBase  Nil       = 0
  lengthBase (Cons _ n) = 1 + n
```

「また、`T a` に対する `lengthBase` のインスタンス宣言は：

```
instance Lengthable (T a) where
  lengthBase  Prune       = 0
  lengthBase (Leaf _)     = 1
  lengthBase (Branch m n) = m + n
```

「`lengthBase :: L a Int -> Int` にしろ `lengthBase :: T a Int -> Int` にしろ、`L` や `T` の第二引数 `x` だったところに `Int` として、部分構造の length の集計結果をはめ込んだものを受け取ったなら、それを元に自分の length を返すような関数になっている。だか、`lengthBase` 自身は決して再帰関数ではない」
「次に、`filter :: (a -> Bool) -> [a] -> [a]` を一般化しよう。再帰的データ構造の中身 `a` を参照する述語関数 `a -> Bool` を扱う都合上、`Filterable` は `L a` や `T a` ではなく、`L` や `T` に対する型クラスとする必要がある。`[a]` を `Rec (t a)` に置き換えると、`filter` の型はこうなりそうだ：

```
filter :: (Functor (t a), Filterable t) => (a -> Bool) -> Rec (t a) -> Rec (t a)
filter p = cata (filterBase p)
```

「述語 p :: (a -> Bool) は filterBase に渡してやらないとフィルター処理の詳細が実装できないだろう。そうなると、(filterBase p) に cata を適用して望みの型になるためには、filterBase の型は、こうだ：

```
class Filterable t where
  filterBase :: (a -> Bool) -> t a (Rec (t a)) -> Rec (t a)
```

「filterBase は再帰データ型パーツ t a x の再帰起点 x に Rec (t a) を埋め込んだもの—t a (Rec (t a)) を取り Rec (t a) を返す。つまり、部分的にフィルター済みのデータ構造が x に埋め込まれたものを受け取り、自分自身が担当する箇所までのフィルター済みデータを構築することで、フィルター処理を一歩だけ進める。L の場合には：

```
instance Filterable L where
  filterBase _  Nil = In Nil
  filterBase p (Cons a as)
    | p a       = In (Cons a as)
    | otherwise = as
```

「ここで、Cons の第一引数 a :: a は述語で判定する必要があるが、as :: Rec (L a) はフィルター済みなのでそのまま結合してよい。同様に：

```
instance Filterable T where
  filterBase _ Prune = In Prune
  filterBase p (Leaf x)
    | p x                  = In (Leaf x)
    | otherwise            = In  Prune
  filterBase _ (Branch a b) = In (Branch a b)
```

「においても、述語を使う必要があるのは Leaf を処理する場合だけだ」

* * *

…… な 箇 所 だ け か い つ ま んで読むんじゃ！」

ふと気づくと、俺は何者かに操られるかのようにコードを書き上げていた。聴覚が戻ってくる。戦艦グラスゴーの艦橋。空に陸に、人類側の圧倒的な劣勢を写しだすスクリーン。こんなに集中したのは久しぶりだ。シラフではありえないハスケルカを発揮していたような気がする。時間すら経過していたという感覚がない。

テスト担当メイドオペ子がすぐさま後を引き継ぎ、高周波タイプ音が唸りをあげる。

「ひとまずテストを行うには Arbitrary のインスタンスが必要ですねえー、あと Show も。Eq と Show あたりを derive しときますか。Rec (L a) と [a] が同型であることは lens の Iso を使って表現すればいいでしょう、length の性質を表現するにはリストの中身の和を取りたいですねそのためには Rec (L a) が Foldable だと便利ですあっ Rec (L a) は正当な型コンストラクタの姿をしていませんからラッパでくるみましょう Rec (L a) と等価な WrappedRec L a 型を作れば WrappedRec L を Functor, Foldable, Traversable にできますね fmap や traverse の中では L や T の第一引数にアクセスする必要がありますから L や T を Bifunctor にしましょうなるほどこれらの再帰基底型 p について Bifunctor p, Bifoldable p, Bitraversable p インスタンスから Functor (WrappedRec p), Foldable (WrappedRec p), Traversable (WrappedRec p) インスタンスが出てくるわけですね。respectively。美しいです」

切れ目のない独り言とともに俺のコードが未知の道具によって大改修されていく。俺はただ唖然としてそれを傍観するしかなかった。

「それではテストコードを実装します。まずはモジュールのインポートから：

```
{-# LANGUAGE FlexibleContexts, ScopedTypeVariables #-}
module GeneralCataSpec where
```

```
import Control.Lens (iso, Iso, Simple, set, view, (^.), to, over)
import Test.Hspec
import Test.Hspec.QuickCheck (prop)
import Test.QuickCheck.Arbitrary (Arbitrary)
import Test.QuickCheck.Function (apply, Fun)
import Test.QuickCheck.Poly (A)
import Data.Traversable
import Data.Foldable (all, sum)
import Data.List.Catamorphism as L
import Data.Rec as R
import Data.Rec.List as R
import Data.Rec.LengthFilter as R
import Data.Rec.Tree as R
import Data.Rec.Wrapped as R
import Prelude hiding (all, sum)
```

[a] と (List a) との間の同型写像 Iso を定義します

```
listIso :: Simple Iso [a] (List a)
listIso = iso f g
  where
    f :: [a] -> List a
    f = L.cata
          (In Nil)
          (\x xs -> In (Cons x xs))
    g :: List a -> [a]
    g = R.cata $ \lx -> case lx of
          Nil -> []
          Cons x xs -> x:xs
```

テスト本文を書きます

```
spec :: Spec
spec = do
```

再帰スキームを使って定義されたリストは

```
  describe "List defined using recursion scheme" $ do
```

Prelude のリストと同型であってほしいです

```
    prop "is isomorphic to Prelude list" $ \xs ->
      over listIso id xs == (xs :: [Int])
```

また、cata を使って作った length 関数は Prelude の length と同一であってほしいです

```
    prop "can reproduce the length function" $ \xs ->
      xs ^. listIso . to R.length == Prelude.length (xs::[A])
```

List も Tree も length や filter に関して同一の性質のセットを満たしてほしいですね。その共通の性質のセットを containerProps と名づけましょう。

```
  describe "List defined using recursion scheme" $ do
    containerProps (undefined :: R.List Int)

  describe "Tree defined using recursion scheme" $ do
    containerProps (undefined :: R.Tree Int)
```

containerProps を定義します

```
containerProps :: forall t a.
  (Lengthable (t a), Filterable t, Functor (t a), Traversable (WrappedRec t),
   Arbitrary (Rec (t a)), Arbitrary (Fun a Bool),
   Show a, Show (Rec (t a)),  Num a)
  => Rec (t a) -> Spec
containerProps _ = do
```

length は要素をすべて 1 に置換して総和を取ったものと等しくなってほしいです

```
  prop "length is the number of elements in it" $ \xs ->
    R.length (xs :: Rec (t a)) == (sum $ fmap (const 1) $ WrapRec xs)
```

述語 p で filter した結果はすべての要素がその述語を満たしていてほしいです

```
  prop "filter filters true elements" $ \xs predGen ->
    let p :: a -> Bool
        p = apply predGen in
      all p $ WrapRec $ R.filter p (xs :: Rec (t a))
```

「実装できました。テストを実行します：

```
$ ./dist/build/spec/spec

GeneralCata
  List defined using recursion scheme
    - is isomorphic to Prelude list
    - can reproduce the length function

  List defined using recursion scheme
    - length is the number of elements in it
    - filter filters true elements
```

```
Tree defined using recursion scheme
  - length is the number of elements in it
  - filter filters true elements
```

テスト結果 Passed——行けます！」わずかに感情を覗かせるメイドさん。

「リストとツリー、まったく種を異にするデータ構造に対して `length` や `filter` が定義できるばかりか、それらの性質のテストまで共通化できるとは……お前、いったい何者なんだ」俺が思わず漏らした言葉は、メイドさんに対してだったか、あるいは再帰スキームに対してか。

「私は圏論習得支援啓発インターフェイスデバイス (Enlightenment of Category Masters Aid Interface Device)、エクメイドです。圏論の実世界における効用を最大化することによって存在意義を達成します」

「Siri に定型文を入力するような遊びをしとる場合ではないぞ！ ワーデンクリフ回路閉じよ！ 改善パッチ用意じゃ！」

《ゲソ！》すでにコスチュームは半壊、バグの連続ビーム攻撃に空中ガード一択でジリー・プアーに追い込まれていたイカ娘が力強く叫ぶ！

圏論よ、私たちに力を!

「ジェネラ☆ライズ↑↑カタモーフィズム!!」

ファンファーレとともに形成される光の舞台。高らかに希望を奏でる弦楽器と勇ましいビートの中で、イカ娘が舞い、光の粒子がコスチュームを、身体の傷を癒してゆく。光の粒子は踊り手の動きを追って幾筋もの流線を成し、イカ娘が決断的に手を、足を振り出すと、ぴたりとフィットした純白の手袋を、ヒールの高いブーツを形成する。リン、と一つ、鈴の音とともに登場した衣装もまた純白で、胸元の大きなリボンや豪華なスカートはおよそ戦闘員というよりは花嫁にふさわしいデザインだ。透けながら織り重ねられた幾層もの生地は、それぞれ異なる繰り返し模様を示し、あまねく再帰構造を司るイカ娘の新たなる力を象徴していた。

災鬼獣レングスが待ちわびたとばかり大小あらゆる触手を噴出させ、軟体が重力に従う間も惜しく一斉に眼球を見開かせる。災鬼獣ヴィルダーンの周囲には次々にバリアが収束しては次々に射ち出され、最初の一つは残像を残しながら一直線に、次のは二分裂を繰り返しながら夥しい数で、次のはトリッキーな曲芸飛行を描きながら、圧倒的な数でイカ娘に殺到！

イカ娘は臆することなく、両手を背後に廻すと二振りの刀をゆっくりと取り出す。峰は円弧を描き、刃の側は直線だ。イカ娘が二刀を頭上に掲げ、はっしと打ち合わせれば巨大な鋏の形！ そこに膨大な雷エネルギーが注入される！

「**かたもる・ピー**スサンダーハリケーン！！」

解放された雷撃は一直線に飛んできた巨大バリアに向かい、受け止める！

解放された雷撃は無数に分裂しながら、殺到するバリアのすべてを正確に受け止める！

一瞬の均衡の後、雷撃はバリアが通過してきた軌道を正確に逆にたどってヴィルダーンに集中！ ヴィルダーンを覆い尽くした後、見えない空中情報伝達経路を辿ってレングズをも飲み込む！

イカ娘は不敵な表情を浮かべてこちらに振り向き、

《勝ったな、でゲソ！》

刀を収める音が響くのと同時に、二体の災鬼獣は無数のバグとともに忽然と姿を消し。

今度こそ根絶に成功したのか、いつまで経っても爆発も何もなかった。

次回予告

「あの、その……暁美ほむら……です。よろしくお願い……します……ぅぅ」

「矢を放った犯人は……この中にいるでゲソ！」

「我がグンマの前途は無限無窮！」

「AdS-CFT 対応！ 余剰次元内で充分な数密度に達した宇宙ヒモ同士の衝突が発生したとしたら、準安定状態に閉じ込められていた真空が解放され、光の速さで我々の平面宇宙を上書きしてしまう！ みんな、みんな死んでしまうんだ！」

「緊急メカニカルシャッター閉鎖、対光学防御！」
「あるわ。次元の壁を打ち破る、方法が。」

信じることは、生きること

「さあ、唱えましょう。神の世界への扉を開く魔法を！」
「俺はようやく理解しはじめたばかりだからな……この果てしなく長い歴史をもつ再帰スキームをよ！」

災機獣とは何だったのか？ 奴等は何処から、何故地球を狙って来るのか？ 人類の未来よりも原稿をとった戦術長は締切りに間に合ったのか？ 余剰次元に隠された驚くべき秘密とは？『萌える再帰スキーム』第一部・完!! `@nushio` 先生の次回作にご期待ください!!

参考文献

この記事の執筆にあたり、以下の諸創作作品からキャラクターや設定を拝借いたしました。素晴らしい作品を発表されている原作者様各位に敬意を表明いたします。勝手なキャラクターや設定の拝借および改変については何卒ご寛恕下さいますよう願い申し上げます。

ネタ元

- 安部真弘著・原作「侵略！ イカ娘」
- 庵野秀明監督・脚本「新世紀エヴァンゲリオン」
- ブラッドレー・ボンド、フィリップ・ニンジャ・モーゼズ「ニンジャスレイヤー」
  `http://d.hatena.ne.jp/NinjaHeads/`
- @dif_engine 蓮物語、殺物語
- オケアノス原作「翠星のガルガンティア」
- 橘公司著・原作「デート・ア・ライブ」
- TEAM VIVID 原作・高村和宏監督「ビビッドレッド・オペレーション」

また、執筆にあたって以下の文献を参考にいたしました。

参考にした文献およびプログラム

[1] **[旧約甘蕉書]** E. Meijer, M. Fokkinga and R. Paterson (1991) “Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire,” Lecture Notes in Computer Science Volume 523, 1991, pp 124-144
Lens について一度きちんと勉強しときたくて原典に当たろうとした、というのが本記事の執筆動機です。ところが、執筆途中で Lens 違いであったことが発覚し……

[2] **[新約甘蕉書]** E. Meijer and G. Hutton (1995) “Bananas in space: extending fold and unfold to exponential types,” Proceeding of FPCA ’95 Proceedings of the seventh international conference on Functional programming languages and computer architecture, pp 324-333
旧約から 4 年。記法が Haskell に近づき、ずいぶん読みやすくなってます。

[3] **[ロゼッタストーン]** `http://blog.ezyang.com/2010/05/bananas` E. Z. Yang (2010)“Bananas, Lenses, Envelopes and Barbed Wire – A Translation Guide”
上記論文の数学記法を Haskell に翻訳してくれているブログ記事です。これのおかげで旧約が解読できました。

[4] `http://hackage.haskell.org/package/recursion-schemes`
ekmett による recursion scheme の実装です。

[5] `http://hackage.haskell.org/package/pointless-haskell`
Alcino と Hugo による recursion scheme の実装で、著者が見た中では一番多種の morphism をカバーしています。

第26章

殺物語

— @dif_engine

物語の概要と目的

この章では、モナドについて数学的とプログラミングの関係を意識しながら説明します。例えば関数型プログラミング言語の一つである Haskell では、モナドは入出力を始めとする多くの標準的なライブラリで採用されています。モナドは圏論（けんろん）という抽象的な数学の枠組みで物事を捉えたときに認識された概念です。

単に幾つかのライブラリの使い方を習得したいだけならばサンプルコードの学習で十分であり、圏論の学習は必要ないのかもしれません。しかしながら、圏論におけるモナドの意味や、モナドとプログラミングとの関係を理解すれば、モナドというデザインの有効性と限界を理論的に把握することができるでしょう。
以下では Haskell をすでにある程度習得している読者を想定し、

- Haskell と圏がどのように関わっているか
- モナドはどのようなプログラミングのアイデアを抽象化しているか
- モナド則をどのように理解するか

という順序で解説します。本記事のストーリー部分は「簡約！？ λカ娘（算）」の「蓮物語（はすものがたり）」の続編になっていますが、数学やプログラミングについての内容に関する限りにおいては「蓮物語」を前提としておらず、独立した記事としてお読み頂けます。

26.1 プロローグ

26.1.1 モナドって何でそんなにすごいんですか？

９月になっても暑い日が続いていたが、秋の気配は深まっていた。
そんなある日のこと、私の部屋には忍野扇（おしのおうぎ）ちゃんがいた。
「――それで翼（つばさ）さん、Haskell のモナドって何でそんなにすごいんですか？」
ほぼ初対面に近い下級生が私の家を訪問――アポ無しで――してモナドについて質問をする。
なんだか奇妙な構図だった。
奇妙ということなら、そもそも扇ちゃん自身にも少し奇妙なところがある。
といっても奇妙なのは彼女の性格のことではなく――いや、性格も少し変わってるかもしれないが――一連の怪異がらみの事件との関係だ。

26.1.2 忍野扇

その日の朝、空にはうろこ雲が一面に広がっていた。
空の東の端を朝日が燃え立つようなオレンジ色に染め上げていた。空を埋め尽くすうろこ雲はオレンジから白へ、そして白から青灰色へと色を変えながら夜闇の名残をとどめる西の空に溶け込んでいる。整然と並ぶ一つ一つの雲の小片の東側は曙光を受けて金色に輝いていた。オレンジ、青灰色、濃紺によって染め分けら

れた空に散りばめられた金彩——その荘厳な空を見上げているうちに、自分が突然異世界に飛ばされてしまったような不思議な気持ちになった。
そんなわけで、私が映画『The Wizard of Oz』のドロシーのセリフ
　「トト、ここはもうカンザスじゃないみたいだね」
を何気なく呟いたのは自然なことだったと思うし、そのときには深刻な意味を込めていたわけではなかった。
しかし、振り返ってみるとこれはその日私が体験した奇妙な事の先触れだったのかもしれない——。
こじつけかもしれないが、そんな風に思えてならないのだ。

　早朝の散歩を終えて家に戻ると、扇ちゃんが家の前にいた。
当たり前のような顔をして。
忍野扇ちゃんは高校一年生で、最近になって阿良々木くんや私の通う私立直江津高校に転校してきた。
扇ちゃんは忍野メメさんの姪だと自称している。
そこからして違和感がある。はっきりと本人の口から聞いたわけではないが、忍野メメさんには天涯孤独の印象が強い。親類縁者がいるとはとても思えないのだ。
それに、彼女はなにやら怪異についての事情通であることを匂わせる発言をいくつかしている。
状況から判断して、彼女がこの町で起きている一連の怪異と何らかの関係を持っているということはいかにもありそうなことだった。それどころか彼女は一連の事件の黒幕のであるかのようなそぶりすら見せていた。
しかし、これらの憶測を裏付ける確かな証拠と言えるようなものが何もないのも事実だった。
怪しいと言えば怪しいが、もし本物の黒幕ならば自分が黒幕であることをこれみよがしに匂わせたりするだろうか。もちろん、露骨に怪しいからむしろ黒幕候補から除外してよいだろう——そう思わせておいて実は本当に黒幕だったという可能性だってないとは言えない。または、彼女が真の黒幕を庇うためにあえて怪しい振る舞いをしているのだと想像することだってできる。
——だが根拠がない以上これらの憶測は疑心暗鬼を生ず、の域を出ない。
そんなわけで、繰り返しになるが彼女が何者なのか断定できずにいる。
推定無罪——というわけで、「たまたま通りかかって」私に質問したいと言った扇ちゃんの台詞を信じた風を装って、私は扇ちゃんにモナドの事を説明することにしたのだ。

　同じ高校とは言え学年も違うし、顔と名前を知っているだけで、扇ちゃんとは事実上初対面に近かった。
だけど、扇ちゃんはなんというか全然物怖じしない子だった。

家の前で彼女が挨拶したときには確か「羽川（はねかわ）さん」だったはずだが部屋に案内したころには私を「翼（つばさ）さん」と呼んでいた。
　「Haskell についてはどの程度知ってるの？」
　「いや、それが全然……というのは冗談で、『すごい H な Haskell 楽しい！』って本でひと通り勉強しました」
寒いギャグを平気で投げてくる辺り、空気を読まない子だなと思う一方で——他人の部屋でくつろいでいる自由闊達さに羨望を覚えた。新学期早々に起きた事件のあと、私は初めて自分の部屋を持つようになったのだった。自分の部屋があるという感覚にまだ馴染めない——。
それだからこそ、この部屋でくつろいでいる扇ちゃんの姿を見て、あと半年たらずしかいないであろうこの部屋が本当に自分の部屋なのだと、急に実感をもって迫ってきたのだった。
　「——んん、それは『すごい Haskell 楽しく学ぼう！』だよね？」
私の返しも少しぎこちない。
　「あ！　そうだったかも知れませんね」
まっすぐに語らず、意図的に間違えたりそれを指摘したりする、それが無駄な会話の楽しみというものなのだろう。そういう会話の楽しさを理解できるようになったのは最近になってからだ。

　そんな無駄な会話を続けながら、私は「教えること」について考えていた——。

26.1.3　教えること

やってみれば分かることだが、教えるということは簡単なことではない。
周囲から頭脳明晰な優等生だと思われている私は、友達に勉強のことで質問を受ける事がある。
自分がどこをどのようにわかっていないか――それを把握した上で質問してくる人は非常に少ない。これは当然の事だ。自分が何を理解していないのかをきちんと把握できるほどの子なら私に質問するまでもなく自力で解決できるはずだ。

したがって、「**ここ**がわからないのだけど」と相談してくる子が実際にわかっていないのは**そこ**ではない、という事態を想定しながら話を聞き、本当にその子がわかっていないのは**どこ**なのかを対話しながら上手く汲み取ってやらないといけない。それができないと、的を外した教え方をしてしまうことになる。

例えば、単位円の方程式がなぜ $x^2+y^2=1$ なのか？ と聞く子は、ピタゴラスの定理がわかっていない場合もあるが、「およそ方程式というのは１つか２つの解しか持たない」という固定観念にとらわれているために「方程式の無限個の解が図形を表すように並ぶという事が理解できていない」場合もある。
そんな場合、噛み砕いて丁寧に説明するのが正解とも言えない。数学的なセンスはあるのにちょっとした事でつまづいている子に対しては、むしろ一般化した形で
「n 変数の多項式 $F(x_1,...,x_n)$ があったとすると、方程式 $F(x_1,...,x_n)=0$ は $n-1$ 次元の超曲面

$$\left\{(x_1,x_2,...,x_n)\in\mathbb{R}^n \;\middle|\; F(x_1,x_2,...,x_n)=0\right\}$$

を定義してると考えられるよね？　今の場合はこれの $n=2$ の場合になってるでしょ」
と説明すれば一発で分かってもらえる場合がある。このような説明は明らかに高校の教科書のどこにも書いていない事に基づいているし、$G(x_1,x_2)=x_1^2+x_2^2+1$ のように零点の超曲面（今の場合は曲線だが）が空集合になってしまう[*1]多項式も存在するので完全に正しい説明でもないが、それでもこのような「変数の数と方程式が定める解の集合の次元」についての直感は数学的に自然だ。そして、誰に教えられるでもなく数学的な直感を発達させている子にはこのような説明の方が理解しやすいこともあるのだ。

今までに友達から受けた質問に対して、私はうまく答えられていただろうか。
私が質問の背景を正しく汲み取れずに見当違いの解説をしてしまったことは、振り返ってみると一度や二度ではなかったはずだ。私が的はずれな説明をしたためによく理解できなかったとき、相手が「羽川翼は優秀だから理解できるけれど自分はそうではなかった」と落胆し諦めてしまった事はなかっただろうか。

受験生にとって勉強がわからないというのは大変なプレッシャーのはずだ。大げさな言い方かもしれないが、質問をしに来た子は私に助けを求めていたのだ。私はその子たちの期待に充分応えてやれたのだろうか。相手の理解度を正しく汲み取れずに的外れの説明をしたことで、結果として助けを求める手を振り払うような事をしてしまった――そんなことがなかったと言えるだろうか――。

わかっている――そんな責任は私にはないのだと。頭脳明晰な優等生だろうがなんだろうが、たかが一生徒に同級生の勉強の面倒を見る責任などあるはずがない。
でも、私は知ってしまった――。世の中には文字通りの他人を助けるために自分の命を投げ出すほどに、他人に責任を感じて行動する人達がいるのだということを。
「君が勝手に助かるだけだよ、委員長ちゃん」
なんて突き放した言い方をしながらこの街の怪異がらみの事件をひと通り解決して、ふらりと姿を消したあの人もそうだ。そして阿良々木くん――。

[*1] 厳密にはどの体の上で考えるかに依存する。複素数体 $\mathbb{C}$ 上で考えると、G は例えば $(x_1,x_2)=(i,0)$ を解として持ち、したがって集合

$$\left\{(x_1,x_2)\in\mathbb{C}^2 \;\middle|\; G(x_1,x_2)=0\right\}$$

は空集合ではない。

阿良々木くんに言わせれば私は彼の命の恩人という事になるらしい。
だが、私にとっても阿良々木くんは命の恩人だ。私は阿良々木くんが好きだ。
自分の命を助けてくれた男の子のことを意識するようになり、気がついたらその男の子を好きになっていた。
そんなふうにして私は阿良々木くんを好きになり、紆余曲折の末、彼には戦場ヶ原さんという素敵な恋人がいることを知りながら図々しくも告白し、そして振られた。
ありふれた話だ。
そんな事があった後でも、いまでも私は阿良々木くんが好きだ。
阿良々木くんは、誰のことも見捨てようとしない。それを優柔不断と断ずることもできるが、その彼の優しさで助けられた人達がいる——私もその内の一人だ。
そんな優しい彼の事を私は好きになったのだ。
私も、誰かを助けられる人になりたい——世界を救うなんて大げさな話ではなく、身近な誰かを見捨てずに助けられる人になりたい——。
いつしか、そんな風に私は思うようになった。

そんなわけで、質問を抱えてわざわざ私の家にきた扇ちゃんに対しても——彼女には色々不審な点があることは承知のうえで——自分に出来る範囲で頑張って解説してみようという気になったのだった。

26.2　準備：集合と関数

さて、どんな流れで説明すれば良いのだろうか。
「——質問は『Haskell のモナドはなぜそれほどすごいのか』だったよね」
「はい」
「つまり、『モナドの使い方』とかじゃなくて、Haskell でモナドが重要だと思われてる理由についての質問ってことかな」
「はい。幾つかサンプルコードなんか見てるうちに、全然互いに関連なさそうなものがモナドのライブラリとして実装されてるので、なんでだろうって気になっちゃって」
扇ちゃんの話を聞きながら頭の中で大まかな方針を立てた——。

モナドは圏論（けんろん）の概念であり、Haskell はプログラミング言語だ。
両者の関わりを示すのなら、どちらかに引き寄せて説明することになるだろう。いま求められているのは概念的な説明だから、様々な概念を形式化する便利さを考えて圏論に引き寄せて考えることにしよう。したがって、Haskell における概念を圏の言葉に置き換える説明をする必要が出てくる。念の為に確認してみよう——。
「扇ちゃんは圏論って知ってる？」
「そういうのがあるってことだけは知ってますが、中身は全く知らないです」
なるほど、そういうことなら圏論を前提として話をするわけにはいかないが、モナドの数学的な定義を説明するためには圏論が必要なので、身近な例から圏論を導入することから始めてみよう。
具体的には、比較的簡単な「集合と関数」あたりを材料にして圏論を導入してみよう。
「さて、モナドをきちんと説明するためには圏論の説明が必要なんだけど、いきなり圏論の説明をしてもわかりにくいと思うんだ」
「なるほど、そういうものなのですか」
「うん、それでね、まず集合と関数の基本のおさらいをして、圏の例を作ってみることから始めようと思う」
「はい、お願いします」
扇ちゃんは食卓から持ってきた椅子に行儀よく座っている。

26.2.1　関数の基礎用語

「扇ちゃんは『関数』ってわかるかな」

「関数ってあれですよね、x に対して $f(x)$ が一つ決まってるみたいなやつですよね？」
「関数の**定義域**と**値域**については説明できる？」
「えーと、集合 X の元 x に対して $f(x)$ が集合 Y に属してるとき、

- 関数 f の定義域は X
- 関数 f の値域は Y

です。記号で書くときは

$$f\colon X \longrightarrow Y$$

とか

$$X \xrightarrow{f} Y$$

ですね。あ、個々の x に対する行き先を書きたいときは

$$f\colon x \longmapsto f(x)$$

って書いたりします。まあこの例だとあんまし意味ないですけどね」
「じゃあ**関数の合成**については説明できるかな？」
「はい……関数 f が X から Y に行くやつで、関数 g が Y から Z に行くやつだとするとき、集合 X の元 x に対して $g(f(x))$ を返す関数を f と g の合成関数って言います。記号だと $g \circ f$ って書くことが多いです。要するに

$$g \circ f(x) = g(f(x))$$

ということになります。値域と定義域を意識して書くなら

$$X \xrightarrow{f} Y \text{ かつ } Y \xrightarrow{g} Z \text{ ならば } X \xrightarrow{g \circ f} Z$$

となります」
うん、悪くはない。これぐらいの理解でも充分な場面も多い。でも、数学的にモナドの説明を行うためにもう少し基礎を確認してから進むことにしよう。

26.2.2 関数を集合とみなすこと

「関数を集合とみなせる事は知ってる？」
「集合とみなす……んですか？　どうやるんでしょう？」
この辺はいわば定石のような話だから答えを先に与えて一緒に考えてみる方針で話を進めてみよう。
「集合 X を定義域とし、Y を値域とする関数 f のすべての情報は、集合

$$\left\{ \langle x, f(x) \rangle \,\middle|\, x \in X \right\} \ (\subseteq X \times Y)$$

に集約できるよね」
「えーと……$\langle x, f(x) \rangle$ が x とその関数値 $f(x)$ のペアになっていて、そのペアを全部持ってる集合を考えるんだから……なるほど、確かに『座標平面上の関数のグラフ』とぴったし同じ情報がこの集合に含まれてますね！」
良かった、集合の基本的な扱いには充分慣れてるみたいだ。
扇ちゃんが続ける。
「——でもこの集合は先に関数が与えられてないと定義できないのだから、このままでは**関数を集合として定義した**ことにはなってないですよね？」
こういう質問が出るということはちゃんと話に付いてきてくれてるということだ。
「そうだね。でも、もう少し工夫すれば関数を集合として定義できるよ。こんな風に——」

そう言って私は手元のノートに書きつけた――。

（集合としての）関数の定義

X, Y を集合とするとき、$f \subseteq X \times Y$ が**関数**であるとは、次の条件 **(func)** が満たされることである：

$$\textbf{(func)} \quad \forall x \in X \ \exists! y \in Y \ \ \langle x, y \rangle \in f.$$

扇ちゃんが不思議そうにノートを見て

「これガンダムですか――？」

と言って「∀」を指さした。ガンダムなら私も一応どんな形かぐらい知ってるけど、∀ってガンダムのツノの AA として使われてたりするのかな？

「やだなぁ翼さん、『げっこうちょう』ですよ『げっこうちょう』！」

扇ちゃんの謎のノリも絶好調だが、雑談に引きずられずに話を戻そう。

「この記号の意味はわからない？」

「えーと『全て』という意味なんでしたっけ？　でも他にもわからない記号あるし、こうやって並んだときにどう読んだらいいのか教えて下さい！」

「そうだね、一番楽な読み方はこれが英語だと思って読む方法だと思う。こんな感じで――」

$$\underset{\text{for all}}{\forall} \ x \ \underset{\text{in}}{\in} \ X \qquad \underset{\text{uniquely exists}}{\exists!} \ y \ \underset{\text{in}}{\in} \ Y \qquad \underset{\text{such that}}{} \ \langle x, y \rangle \ \underset{\text{in}}{\in} \ f.$$

「あ、なんか漢文を読むときみたいな読み方ですね！　記号をそのまま英語に置き換えてみるとこんな感じですか――」

$$\text{For all } x \text{ in } X, \text{ uniquely exists } y \text{ in } Y \text{ such that } \langle x, y \rangle \text{ in } f.$$

「意味は取れるかな？」

「……なんか英語を知らない人が辞書引きながら作ったようなひどい文だし、特に ”such that” 以降は英語として完全におかしいですけど、言ってることは分かる気がします」

「うん、じゃあそれを日本語にすると――？」

「ええと……この ”for all x in X” は『X の全部の元 x について』だから x は X 全体を動くけど、y の方は『全部の x に』じゃなくて『個別の x に』対応して『ただ一つ存在する』んですよね？」

この子はすごく察しがいいみたいだ。――でも理解のレベルをもう少し詳しく確認しておこう。

「ちゃんと文章にするとどうなるかな？」

「えっと、こんな感じですかね」

扇ちゃんは次のようにノートに書いた：

x に対して、Y の元 y が 1 つだけ存在して、$\langle x, y \rangle \in f$ となる

↑

X の全ての元 x について！

……これはちゃんとした日本語の文章じゃないよね――と言いたくなったが、よく見るときちんと理解している事を伺わせる書き方だった。折り目正しい日本語に直していると国語の時間になってしまう――これでいいことにして先に進もう。

「日本語としてはどうかなと思うけど扇ちゃんは理解できてるみたいだね。じゃあ次の質問だけど、これで『集合として関数を定義』したことになってることは納得できる？」

「えーと、条件 **(func)** が満たされていたら確かにそれは私たちのよく知ってる関数だという事を確かめればいいんですかね。それなら簡単です。x に対して必ず y が 1 個だけ存在してるんだから、その y が x の関数値なんだと思えば、集合 f は確かにさっきの『関数のグラフ』になってます！

——そこまではわかるんですが翼さん、どうして『関数』という概念をわざわざ集合として扱いたいんですか？」

「確かに関数ってのは要するに『x に対して $f(x)$ を対応させる規則』なんだけど、
もっと明確に『集合』としての具体的な形を与えておくと抽象的な議論をするときに無用の混乱を避けられる事があるんだ」

「ふうん……それってどんな場合でしょうか？」

26.2.3 関数たちの集合 $\mathrm{Map}(X,Y)$

「そうね——たとえば『関数の集合』を考えたいときかな」

「関数の集合——ですか……？」

扇ちゃんが考えこんでいた。少し助け舟を出したほうが良さそうだった。

「例えば sin や cos は関数だよね。だとするならば

$$\boxed{\{\ \sin, \cos\ \}}$$

は関数の集合だよね」

「えーと……ああ！『関数 f が x において取る値を $f(x)$ と書く』ルールだったから、関数本体の名前は $\sin(x)$ じゃなくて sin なんだ！」

論理的に一貫した記号法というのは整然としているが、慣習的な記号法や名称と衝突することもある。そのため、整然としているのにもかかわらず意味を汲み取りづらくなることも起こる。

「……話を戻すと、『関数』を『集合』として表すことができたから、『関数の集合』のようなものを明確に議論できるようになったんだよ」

「なるほど、『関数』を『集合』として扱う利点は一応わかった気がします」

「さて、定義域が集合 X で値域が集合 Y になっている関数全部を集めた『関数の集合』を $\mathrm{Map}(X,Y)$ と書くことにしよう」

「つまり

$$\boxed{\mathrm{Map}(X,Y) := \{\ X \text{を定義域とし} Y \text{を値域とする関数}\ \}}$$

ってことですね」

「うん、そうだね」

「なるほど、確かに『関数』を集合っていう『もの』として定式化してなかったら $\mathrm{Map}(X,Y)$ みたいなものを考えるのは難しい感じですね」

26.2.4 $\mathrm{Map}(X,Y)$ と関数合成

「ついでだけど、$\mathrm{Map}(X,Y)$ を使ってみると関数合成の条件はこんなふうにも書けるよね——」

$$\boxed{f \in \mathrm{Map}(X,Y),\ g \in \mathrm{Map}(Y,Z) \implies g \circ f \in \mathrm{Map}(X,Z).}$$

「ええと、$f \in \mathrm{Map}(X,Y)$ は $X \xrightarrow{f} Y$ の言い換えだと思っていいから、確かにそうなりますね」

「そんなわけで、関数の合成『$\circ$』を関数とみなすならば、

$$\boxed{\begin{array}{ccccc} \circ\colon \mathrm{Map}(Y,Z) & \times & \mathrm{Map}(X,Y) & \longrightarrow & \mathrm{Map}(X,Z) \\ \langle g,f\rangle & & & \longmapsto & g\circ f \end{array}}$$

みたいな感じになるね」

「なるほど、関数の合成というのを一段上から眺めるとこんなふうに見える感じですかね」

26.2.5 $\mathrm{Map}(\varnothing,Y) = ?$

「理解のテストを兼ねて、集合 Y に対して、関数の集合 $\mathrm{Map}(\varnothing,Y)$ がどうなるか考えてみると面白いよ」

「ええと、$\varnothing$ は空集合——何も含まれてない集合——ですよね」

「うん」

「$\mathrm{Map}(\varnothing, Y) = \varnothing$ なんじゃないですかね？」

「つまり、空集合 $\varnothing$ を定義域として、集合 Y を値域とする関数は世の中に一つも存在しない、ってのが扇ちゃんの主張かな」

「えーと……はい、そういうことです」

「そうなる理由をちゃんと考えた？」

「いやあ、そうなるよねっていうか要するに直感なんですけどね、多分大体あたってます」

「じゃあ今度は直感じゃなくて、きちんと論理的に考えてみて」

「論理的に考えても $\mathrm{Map}(\varnothing, Y) = \varnothing$ な気がするんですけど、まあやってみますか。まず、

$$f \in \mathrm{Map}(\varnothing, Y) \Rightarrow f \subseteq \varnothing \times Y$$

となりますが、最後に出てきた $\varnothing \times Y$ は空集合になります」

「それが空集合になるのはどうしてかな？」

「これも直感でそうかなって思ったんですけど……真面目に考えてみます。ええと、背理法を使うことにします。要するに、$t \in \varnothing \times Y$ と仮定して矛盾を導ければいいわけです。明らかに

$t \in \varnothing \times Y$
$\Leftrightarrow$ なんかの t_1 と t_2 によって $t = \langle t_1, t_2 \rangle$ と表せて、しかも $\langle t_1, t_2 \rangle \in \varnothing \times Y$ となってる

となりますが、順序対の性質から、このとき $t_1 \in \varnothing$ が成り立ってないといけないです。空集合の定義からこれはあり得ないので矛盾です。よって、『どんな t も $\varnothing \times Y$ に属さない』事が言えました」

「いまの結論をさっき説明した記号を使って書いてみると

$$\forall t \ \ [t \notin \varnothing \times Y]$$

になるね」

「なるほど、言葉でごちゃごちゃ書くよりすっきりしますね――。で、話を戻すと、どんな t も属さないような集合なんだからそれは空集合です。つまり、$\varnothing \times Y = \varnothing$ ということです」

「ここまでで

$$f \in \mathrm{Map}(\varnothing, Y) \Rightarrow f \subseteq \varnothing \times Y = \varnothing$$

が言えたことになるね」

「はい、そして $f \subseteq \varnothing$ だから $f = \varnothing$ じゃなきゃいけないから、結局

$$\mathrm{Map}(\varnothing, Y) \subseteq \{\varnothing\}$$

が言えました。そして、明らかに $\varnothing \notin \mathrm{Map}(\varnothing, Y)$ だから、確かに $\mathrm{Map}(\varnothing, Y) = \varnothing$ が示せました」

「いまの『明らかに』ってところがあやしい」

「あれ、明らかだと思うんですが間違ってますか。考えてみますね。定義 **(func)**(p.364) から」

$$\varnothing \in \mathrm{Map}(\varnothing, Y) \Leftrightarrow \forall x \in \varnothing \ \ \exists! y \in Y \ \ \langle x, y \rangle \in \varnothing$$

となります。それで、右側の論理式が『$\forall x \in \varnothing \ \cdots$』の形をしていますが、空集合の元について何か主張していても成り立たないに決まってるから右側の論理式はウソになります。だから $\varnothing \notin \mathrm{Map}(\varnothing, Y)$ が言えると思うんですけど……」

なるほど――扇ちゃんは、空集合について誤解をしているようだ。でも、これは私が話を少し急ぎすぎたせいかもしれない。誤解を正しておかねばならないが――誤解を正す、というのは単に「正解を与える」ことだけでは達成できない。そうではなく「正解に至る道筋を自分で辿れるように導く」ことが必要だ。さて、ではこの扇ちゃんの誤解を正す道筋をどうやって示したら良いのだろうか――。

「残念だけど、結論から言うとやっぱり間違ってるよ。そうだね、じゃあ背理法で $\varnothing \in \mathrm{Map}(\varnothing, Y)$ を示してみようか――。

$$\varnothing \notin \mathrm{Map}(\varnothing, Y). \qquad \textbf{(背理法の仮定)}$$

――確認なんだけど扇ちゃん、$\varnothing \in \mathrm{Map}(\varnothing, Y)$ を関数の定義にもとづいて変形すれば、この仮定は

$$\neg[\forall x \in \varnothing \ \exists! y \in Y \ \ \langle x, y\rangle \in \varnothing]$$

と同値だよね」

「はい、そうですね」

「さて、一般に $\neg[\forall x \in X \ \ A(x)]$ が成立するのは、『$A(x)$ が成立しないような $x \in X$ が少なくとも一つ存在するとき』だよね」

「この A というのは X の元についての『述語』、つまり

$$A\colon X \longrightarrow \{\ \text{真}, \text{偽}\ \}$$

だと思えばいいんですよね」

「うん」

「ええと、待ってください、考えてみます。『すべてのカラスは黒い』を否定したかったら……ああ、そうですね、黄色でもピンクでもいいからとにかく黒くないカラスを一羽見つけてくれば否定できますね。たった一羽見つければいいんだからお安い御用じゃないですかね？」

「ということは、今の場合だと、空集合 $\varnothing$ の中に、『$\exists! y \in Y \ \langle x, y\rangle \in \varnothing$』 の反例となるような x が最低一つ存在しなきゃいけないよね？」

「はい……あれ？　でも空集合の中には『なにもはいってない』んですからそれは全然お安くないですね。というか頑張っても絶対無理です」

「つまり、$\neg[\forall x \in \varnothing \exists! y \in Y \ \langle x, y\rangle \in \varnothing]$ という仮定から矛盾が導かれたわけだね。よって背理法により

$$\forall x \in \varnothing \exists! y \in Y \ \langle x, y\rangle \in \varnothing$$

が導けるし、これは

$$\varnothing \in \mathrm{Map}(\varnothing, Y)$$

と同値だね」

「おお……！」

「今までの結果をまとめると

$$\varnothing \ \in \ \mathrm{Map}(\varnothing, Y) \ \subseteq \{\varnothing\}$$

となるし、これをさらに整理すれば

$$\mathrm{Map}(\varnothing, Y) \ = \ \{\varnothing\}$$

ということだよね」

「ええっ空集合 $\varnothing$ って関数なんですか！？」

「うん、なんかちょっと意外かもしれないね」

扇ちゃんは食い入るようにこれまでのノートを見返してからこう言った――。

「翼さん、これってやっぱり少し変じゃないですか？　だって、今の論法を真似すると 任意の 述語 $A(x)$ について

$$\forall x \in \varnothing \quad A(x)$$

が示せちゃいますよね？」

「そうだね」

「だとするとですね、いま $A(x)$ はどんな命題でも良かったから、$A(x)$ にはなんかの述語 $B(x)$ を入れてもいいし、その述語の否定 $\neg B(x)$ を入れてもいいことになります。だとすると結局

$$\forall x \in \varnothing \quad B(x)$$

も

$$\boxed{\forall x \in \varnothing \quad [\neg B(x)]}$$

どっちも正しいことになっちゃいますよね? これって気持ち悪くないですか？」

「確かに気持ち悪いかもしれないけど、さっきの議論みたいに『$\forall x \in \varnothing\ B(x)$』みたいな命題を否定しようとすると結局反例が空集合 $\varnothing$ から取り出せないといけないからそれは無理ってことになるよね」

「なるほど、『論理』と『空っぽの集合』を認めるなら、『空集合の元についてはどんなことも成り立つ』も飲まなきゃいけないわけですね」

扇ちゃんはそう言いながら両腕で輪を作ってちょんちょんと両手の人差し指を突き合わせてみせた。ふふ、可愛いジェスチャーだ。

「そうだね、こういうのには段々に慣れていけばいいと思うよ」

「それにしても翼さん、『どんな途方も無い夢も叶う場所』が空集合しかないってのは何か重大な事を示唆してる気がしませんか？」

「いや別に……だっていまは数学の話してるんだし……」

「それはともかく、『関数』を集合として定義したからこそ $\mathrm{Map}(\varnothing, Y) = \varnothing$ って断言できる様になったんですね。多分、関数を『何かに何かを対応させる規則』みたいに漠然とした形で定義していたらこういう事をハッキリと言うのは難しかったとおもいます」

集合と関数についての最低限の確認は済んだのでいよいよ圏の話に進もう。

26.3 圏の導入

圏論の難しさというのは、群論の難しさと少し似ていると思う。
どちらも定義と初歩的な理論における記号操作やルール適用自体の難易度はそれほど高くない——
にも関わらずどちらも入門者泣かせで知られている。
この難しさは、「何をイメージしたら良いか」がわからないことから来ていると思う。
知らない場所——知り合いがおらず、言葉も通じず、周囲の動植物も見慣れない——そんな場所を旅行する場合には「ガイド」が必要だ。
数学の新しい分野を学ぶとき、脳は丁度見知らぬ場所を旅行するときのようなプレッシャーを受けている。
そんなときの「ガイド」になってくれるのが個々の具体例だ。群論でも圏論でも、最近は最初のほうで充分に具体例を扱っている教科書が増えてる気がする。

26.3.1 集合と関数がなす圏

——というわけで、ここでも具体的な圏の例をまず導入し、充分に準備をしてから抽象的な圏の定義を述べることにした。こうするといくらかの場所で似たような議論を繰り返すことになるかもしれないがそれは仕方ない。

「さて、集合と関数の話はこの辺で終わりにするね。モナドは圏論の概念だから準備として圏の話をしようと思うんだけど、圏の定義を扱う前に具体的な圏の例を扱うことにしたいんだ」

「はい、お願いします」

「たとえば集合族(しゅうごうぞく)

$$\boxed{V = \{X_i \mid i \in I\}}$$

があったとして、それをもとに

$$\boxed{E := \bigcup_{i,j \in I} \mathrm{Map}(X_i, X_j)}$$

という集合 E を考えることにしようか——」

「あの、質問なんですが『しゅうごうぞく』ってなんですか？」

「『族』は『集合の集合』、つまり全ての元が集合となってるような集合を表す言葉だよ」
「ふーん……じゃあ『集合の集合の集合』――『族の集合』ってことですけど――これにも何か名前があったりするんですか？」
「それは聞いたことないな――あまりそういうのを扱う必要がないからかな？　『集合の集合』も集合には違いないんだから『集合族』という言葉はどちらか気分的な言葉かもしれないね」
「なるほど――ところで、いま気づいたんですが『集合族』と『暴走族』って音も意味も似てませんか？」
「……『族』しか共通点がないと思うんだけど……」
「ほら、群れてないと『族』になれない辺り、意味が似てるんですよ！」
「……それってやっぱり『族』しか共通点がないってことだよね……」
「あはは、言われてみたらその通りですね！」
うーん……この子のノリ結構疲れる……。しかし、気を取り直して先に進むことにしよう。

26.3.2　集合と関数：圏による抽象化

「さて、そろそろ圏の定義に進もう――」
そう言って私は圏の定義をノートに書き出した――。

圏の定義

集合 O, A と写像

$$\begin{array}{lll} A \xrightarrow{\mathrm{dom}} O, & A \xrightarrow{\mathrm{cod}} O, & O \xrightarrow{\mathrm{id}} A \\ f \longmapsto \mathrm{dom}(f), & f \longmapsto \mathrm{cod}(f), & X \longmapsto \mathrm{id}_X \end{array}$$

と、写像

$$\begin{array}{ccc} \mathrm{Comp}(A) & \xrightarrow{\circ} & A \\ \langle g, f \rangle & \longmapsto & g \circ f \end{array}$$

が与えられたとする。ただし、

$$\mathrm{Comp}(A) := \left\{ \langle g, f \rangle \in A \times A \;\middle|\; \mathrm{dom}(g) = \mathrm{cod}(f) \right\}$$

とする。

このとき、$\mathbf{C} := \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ \rangle$ が**圏(category)** であるとは **(cat1)-(cat5)** が成立することである：

(cat1)	$\forall X \in O$	$\mathrm{cod}(\mathrm{id}_X) = \mathrm{dom}(\mathrm{id}_X) = X.$
(cat2)	$\forall \langle g, f \rangle \in \mathrm{Comp}(A)$	$\mathrm{dom}(g \circ f) = \mathrm{dom}(f).$
(cat3)	$\forall \langle g, f \rangle \in \mathrm{Comp}(A)$	$\mathrm{cod}(g \circ f) = \mathrm{cod}(g).$
(cat4)	$\forall f, g, h \in A$	$\langle h, g \rangle, \langle g, f \rangle \in \mathrm{Comp}(A) \implies (h \circ g) \circ f = h \circ (g \circ f).$
(cat5)	$\forall f \in A$	$f \circ \mathrm{id}_X = \mathrm{id}_Y \circ f = f,$ ただし $X := \mathrm{dom}(f), Y := \mathrm{cod}(f).$

圏 $\mathbf{C} = \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ \rangle$ が与えられたとき、

$$\begin{aligned} \mathrm{Ob}(\mathbf{C}) &:= O, \\ \mathrm{Arr}(\mathbf{C}) &:= A \end{aligned}$$

とそれぞれ呼ぶことにする。
集合 $\mathrm{Ob}(\mathbf{C})$ の元を**対象(object)** と呼ぶ。集合 $\mathrm{Arr}(\mathbf{C})$ の元を**射(arrow)** と呼ぶ。
対象 X に対して定まる射 id_X を**恒等射(identity arrow)** と呼ぶ。
射 f と g から $g \circ f$ を作る操作を**合成 (composition)** と呼ぶ。

射 f に対し、$X := \mathrm{dom}(f)$ そして $Y := \mathrm{cod}(f)$ とする。このとき f, X, Y の関係を簡潔に

$$f: X \to Y \quad \text{または} \quad X \xrightarrow{f} Y$$

と表記する。この表記を使えば、

$$\forall X, Y, Z \in O \quad X \xrightarrow{f} Y,\ Y \xrightarrow{g} Z \implies X \xrightarrow{g \circ f} Z$$

となることが確認できる。
対象 X と Y に対し、

$$\mathrm{Hom}(X, Y) := \left\{ f \in \mathrm{Arr}(\mathbf{C}) \;\middle|\; \mathrm{dom}(f) = X \text{ かつ } \mathrm{cod}(f) = Y \right\}$$

と書く。$\mathrm{Hom}(X, Y)$ を対象 X と Y が定める **hom（ホム）集合**と呼ぶ。

「翼さん、『ドム』『コッド』『イド』と並べるとなんだかモビルスーツの名前みたいですよね！」
「扇ちゃん、ガンダムの話はもういいから」
「――ところで『写像』って言葉がさりげなく出てきてますがこれは『関数』とは違うんですか？ よく似た言葉だと思うんですが」
「私は、『写像』という言葉を『関数』と同じような意味で使ったんだ。つまりどちらの言葉も、『あるもの』に別の『なにか』を対応させる規則という意味で使ってる」
「ふむふむ、そうすると使い分けは気分的にしてる感じですか？」
「そうだね、もう少しその気分を説明すると、『関数』といったときには今はその規則を集合として明確に捉えられていることを強調する気分かな」[*2]
「なるほど、逆に『写像』はあまりその規則そのものがどんな形をしてるかとか議論したくない気分って感じですかね」
「そうなるかな。さて、早速だけど、

$$\boxed{\mathrm{Ob}(\mathbf{C}) = V,\ \ \mathrm{Arr}(\mathbf{C}) = E}$$

となる圏はどうしたら作れるかわかるかな――？」
「えーと、つまり

$$\boxed{\mathrm{Ob}(\mathbf{C}) := V,\ \ \mathrm{Arr}(\mathbf{C}) := E}$$

としたとき、残りの dom, cod, id, ∘ を**どんなふうに定義したら**圏になってくれるかを考えればいいんですよね――。集合 E の元は関数ですから、この『∘』はそのまま関数合成の記号だと定義しちゃっていいのかな――それに合わせて、関数の集合 $\mathrm{Comp}(E)$ は『合成可能な関数のペアの集合』なんだと思ってみることにします。
次に dom と cod について考えてみますが、一般に $f: X \to Y_1$ で $g: Y_2 \to Z$ のとき、もし $Y_2 = Y_1$ ならば合成関数 $g \circ f$ を考えていいわけです。そんな風に考えると、

$$\boxed{g \text{ の定義域が } f \text{ の値域と等しい}}$$

という条件が合成可能であるための十分条件ですね。この条件を集合 $\langle g, f \rangle \in \mathrm{Comp}(E)$ となるための条件

$$\boxed{\mathrm{dom}(g) = \mathrm{cod}(f)}$$

と同じ事だということにすればうまくいきそうです――要するに dom は定義域を教えてくれる写像だとして、cod は値域を教えてくれる写像ってことにすればよさそうです」
「$\mathrm{Comp}(E)$ に属するペア $\langle g, f \rangle$ を合成してできた $g \circ f$ が E に属してることも簡単にわかるね。じゃあ残ってるのは id の定義かな」

[*2] この原稿における「関数」「写像」の気分使い分けの基準は、一般的なものではないかもしれません。

「これはどう考えたらいいんですかね——あーなんか思いつきました。まず、条件 **(cat1)** があるから id_X は関数として

$$\boxed{\mathrm{id}_X : X \to X}$$

となってるはずですね。次に、条件 **(cat5)** を考えると、関数 $f: X \to Y$ があったときに

$$\boxed{\forall x \in X \quad f\big(\mathrm{id}_X(x)\big) = \mathrm{id}_Y\big(f(x)\big) = f(x)}$$

となってなきゃいけないです。そうなると

$$\boxed{\begin{aligned} \mathrm{id}_X : X &\longrightarrow X \\ x &\longmapsto x \end{aligned}}$$

として定義してやれば良さそうですね」

「よく出来ました。さて、少し長くなったから $\mathrm{Ob}(\mathbf{C}) := V,\ \mathrm{Arr}(\mathbf{C}) := E$ としたときの圏について表を書いて整理しようか——」

「ところで、その『$\mathrm{Ob}(\mathbf{C}) := V,\ \mathrm{Arr}(\mathbf{C}) := E$ としたときの圏』って無駄に長いですね」

「そうだね、この圏は $\mathbf{Ens}_V$ と呼ばれる事がある[*3] から、私たちもそう呼ぶことにしましょう」

私はノートに次のような表を書いた (表 26.1)：

圏 $\mathbf{Ens}_V$ の言葉	集合論の言葉
対象 X	集合 $X\,(\in V)$
射 $f: X \to Y$	関数 $f \in \mathrm{Map}(X, Y)$
対象 X に対応する恒等射 id_X	恒等関数 $\mathrm{id}_X = \{\langle x, x\rangle \mid x \in X\}$
$f: X \to Y$ と $g: Y \to Z$ の合成射 $g \circ f$	合成関数 $g \circ f: X \to Z$
対象 X と Y が定める hom 集合 $\mathrm{Hom}(X, Y)$	関数集合 $\mathrm{Map}(X, Y)$

表 26.1: 圏 $\mathbf{Ens}_V$ と集合論の言葉の対応

「とりあえず、$\mathbf{Ens}_V$ というのが『圏』の具体例であることはこれでわかりました。ただ、いまのところあまりピンとこないです。なんというか大げさっていうか、こういう話ならそのまんま集合と写像でいいんじゃないかなって思っちゃうんです」

非常に理解しやすい具体例から説明したので、こういう印象を持つのも当然かもしれない。もう少し先で「関手」を対象とし、「自然変換」を射とする圏の話をすれば印象も変わってくると思う。とはいうものの、あまり急いで話を進めてしまうと多分扇ちゃんは混乱してしまうだろう——。

「確かに $\mathbf{Ens}_V$ は圏の例としては平凡なんだけど、こんなに身近に転がっているものが圏になっているということ自体が圏論の適用範囲の広さの一例なんだよ」

——ここではこんなふうにとりあえずの説明しておこう。

「なるほど、そういうものですか」

26.3.3 型を対象とし、関数を射とする Haskell の圏

「さて、扇ちゃんは Haskell について基礎的な事はわかってるんだっけ」

「はい、そのつもりです。入門書を一冊なんとなく通読した程度ですけどね」

「ついさっき $\mathbf{Ens}_V$ を扱ったけど、そこからの類推で『型を対象とし、関数を射とする Haskell の圏』もわかるかな」

「えーと、『型』ってのは `Int` や `String` みたいな『値の集合』だと思っておけばいいですか？」

「それでいいと思うよ」

Haskell に限らず、プログラミング言語における**型 (type)** について詳しく語り出したら非常に難しくなってし

[*3] CWM p.12, (訳書では p.14)

まう。言語の実装や型推論を話題にしてるわけではないので、ここは簡単に「型とは値の集合のこと」ぐらいの理解で良いだろう。

「だったら、その『型を対象とし、関数を射とする Haskell の圏』のことは一応わかってると思います」

「この圏は **Hask** と呼ばれることがあるので、私たちもそう呼びましょう」

「はい」

「早速だけど、圏 **Hask** における概念と Haskell の概念の対応表を書いてもらおうかな」

扇ちゃんはさっき私が書いた表を真似て次のように書いた (表 26.2)：

圏 **Hask** の言葉	Haskell の言葉
対象 X	型 `X`
射 $f\colon X \to Y$	関数 `f :: X -> Y`
対象 X に対応する恒等射 id_X	関数 `id :: X -> X`
$f\colon X \to Y$ と $g\colon Y \to Z$ の合成射 $g \circ f$	合成関数 `g.f :: X -> Z`
対象 X と Y が定める hom 集合 $\mathrm{Hom}(X, Y)$	型 `(X -> Y)`

表 26.2: Haskell の圏 **Hask** と Haskell の言葉の対応 (1)

「できました」

「恒等射 id_X に対応する Haskell の言葉が `id :: X -> X` になってるね」

「はい、最初は `id` のまま書こうと思いましたが、`id` は多相関数なので、普通の意味での関数とは考えられないから、型 `X` に制限した形 `id :: X -> X` にしました」

「……さて、いま考えてる圏 **Hask** と、さっき考えた $\mathbf{Ens}_V$ の一番大きな違いはなんだと思う？」

「んー……？」

扇ちゃんの様子を見て「しまった」と私は思った。「違う」とか「同じ」というのは視点の取り方で変わってくる話だ。扇ちゃんと私では知識と理解の前提が異なるのだから、こういう質問は漠然としすぎてる。

「ごめん、いまの質問は取り消しね。本当は、$\mathrm{Ob}(\mathbf{Ens}_V)$ は有限集合かもしれないけど $\mathrm{Ob}(\mathbf{Hask})$ は常に無限集合になるところが違うね、という事を話したかったんだ」

「なるほど……。ええと——でも $\mathrm{Ob}(\mathbf{Hask})$ が無限集合になる、というのはどうしてでしょうか？」

「具体例で考えると簡単にわかるよ。例えば `Int` から出発して考えると

$$
\begin{aligned}
&\texttt{Int} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->Int)} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->(Int->Int))} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->(Int->(Int->Int)))} \in \mathrm{Ob}(\mathbf{Hask})\\
&\dots
\end{aligned}
$$

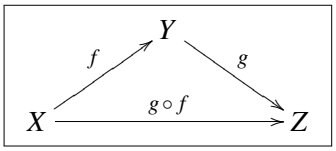
という型の系列があって、この系列だけでも、すでに無限個の型が含まれてるよね」

「なるほど……正直、圏 **Hask** の話って圏 $\mathbf{Ens}_V$ と本質的に同じじゃん、なんで無駄にページ稼いでるんだよって思ってキレそうだったんですが、こう言われてみると結構本質的なところで違うように思えてきました」

26.3.4 図式、特に可換図式

そろそろ図式の説明をしておこう。圏論において図式はあまりにも重要だ。言葉と論理だけで追うのが大変な議論でも、図式を使うことにより大幅に簡略化できることがある。図式の持つ、このような力こそが圏論を発展させた原動力の一つだったと言えるだろう。

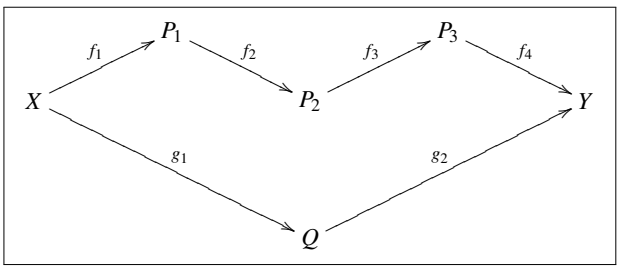
「さて、例えば射の合成の様子を書くときに射の矢印が斜めでもいいことにすれば



みたいに書けて、わかりやすいよね。射の矢印をこういう風に平面的に配置した図のことを**図式 (diagram)** って言うんだ。要するに複数の射を並べて書くってだけなんだけど、こうやって自由に配置してみると見通しよく議論できる事が多いんだよ」

「なるほど」

「——特に大事なのは、



こんなふうに、図式に含まれる対象 X から対象 Y まで射を乗り継ぐみたいに合成していく方法が二通り以上あったときに『どっちの経路で乗り継いで合成しても同じ射になる』ような場合だね」

「いまのこの図式で言えばつまり $f_4 \circ f_3 \circ f_2 \circ f_1 = g_2 \circ g_1$ となる場合ですね」

「うん、そうだね。それで、図式 D があったとき、D の中の対象のペア $\langle X, Y \rangle$ をどんな風にとっても、対象 X から Y への二つ以上の『射の経路』が『合成すると同じもの』になってしまうとき『図式 D は**可換 (commutative)** である』と言うんだ」[*4]

「かかん、ですか」

「可換図式を使った議論は圏論ですごく大事なポイントの一つだよ。理屈の上では、可換図式を使わずに議論することだってできるよね？　たとえば上の図式は

$$f_1 : X \to P_1,\ \ f_2 : P_1 \to P_2,\ \ f_3 : P_2 \to P_3,\ \ f_4 : P_3 \to Y,$$
$$g_1 : X \to Q,\quad g_2 : Q \to Y.$$

と書くことができるし、その可換性についてはさっき扇ちゃんが言ったみたいに

$$f_4 \circ f_3 \circ f_2 \circ f_1 = g_2 \circ g_1$$

と書ける」

「あー……でも、こうやって条件を列挙しちゃうとめっちゃわかりにくいですね」

「——というわけで、私たちもどんどん可換図式を使っていきましょう」

「はい、わたしもそれがいいです」

「圏論は、可換図式という武器があったからここまで発展したんじゃないかなって思うんだ。ある意味、代数学に記号を持ち込んだヴィエト[*5] の工夫に匹敵すると言っても構わないと思う」

「ふうん、誰でも思いつきそうな感じがするけど、そこまですごいですか」

「変数に記号を使うというアイデアだって、知ったあとでなら誰でも思いつきそうに感じるけど、実際に

[*4] 本章における図式とその可換性の定義はやや曖昧かもしれません。圏 $\mathbf{C}$ を、対象を頂点とし、射を有向辺とする有向グラフ $\langle \mathrm{Ob}(\mathbf{C}), \mathrm{Arr}(\mathbf{C}) \rangle$ とみなすとき、図式はその部分グラフとして定義できます。グラフ理論の言葉を借りれば、図式の可換性をもっと明確に定義することができます。たとえば
J. L. Bell, "Toposes and Local Set Theories", Oxford University Press (1988)
ではそのような扱いがされています。

[*5] フランソワ・ヴィエト（1540 年〜1603 年）`http://ja.wikipedia.org/wiki/`フランソワ・ビエト

はヴィエトが導入するまでは非常に不自由なまま数学をやっていたんだよね。説明されたら誰にもわかるぐらい平易——本当の本当にすごいアイデアってのはそういうものなんだと思う」

「なるほどなぁ」

26.4 関手

26.4.1 関手の定義

「関手 (functor) というのは、Haskell の `Functor` の名前の由来になってるものなんだ」

「`Maybe` なんかがそうですね。`Maybe` は型 `X` を型 `Maybe X` に移すから、『関数』じゃないけど『関数っぽい』から関数を思わせる名前になったのかなと気になってました」

「圏論における関手というのは圏から圏への写像で、『可換図式を壊さない』ものなんだ。定義をちゃんと書くならば——」

私は次のようにノートに書いた——。

関手の定義 (object-part と arrow-part を分けて記述したもの)

圏 $\mathbf{C}, \mathbf{D}$ と二つの写像

$$F_O : \mathrm{Ob}(\mathbf{C}) \longrightarrow \mathrm{Ob}(\mathbf{D}),\ \ F_A : \mathrm{Arr}(\mathbf{C}) \longrightarrow \mathrm{Arr}(\mathbf{D})$$

が与えられているとする。このとき、$F = \langle F_O, F_A \rangle$ が $\mathbf{C}$ から $\mathbf{D}$ への関手 (functor) であるとは次の **(func1)**-**(func3)** が成立することである：

(func1) $\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad F_A(f) \in \mathrm{Hom}\big(F_O(X), F_O(Y)\big).$

(func2) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad F_A(\mathrm{id}_X) = \mathrm{id}_{F_O(X)}.$

(func3) $\forall \langle g, f \rangle \in \mathrm{Comp}\big(\mathrm{Arr}(\mathbf{C})\big) \qquad F_A(g \circ f) = F_A(g) \circ F_A(f).$

「ええと、ここにでてくる $\mathrm{Comp}\big(\mathrm{Arr}(\mathbf{C})\big)$ ってのは……ああ、p.369 で定義されてたやつですね。要するに

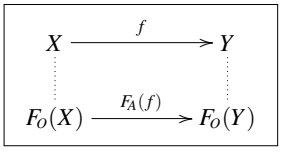
$$\langle g, f \rangle \in \mathrm{Comp}\big(\mathrm{Arr}(\mathbf{C})\big) \iff g \circ f \text{ が定義できる}$$

ということか。えーと、でもこれがなんで可換図式と関係あるんですかね」

「とりあえず図式にしてみたらいいんじゃないかな」

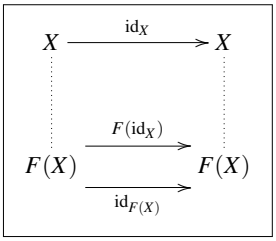
「なるほど、習うより慣れろってわけですね。じゃあまず条件 **(func1)** からやってみますか」

そう言って扇ちゃんはこんな図式を書いた：



「関手の object-part F_O と arrow-part F_A の記号を区別しないで両方 F と書くことにすれば条件 **(func2)** はこんなふうになるね——」

そう言って私は次のように書いた：



「ええと……あの、F_O と F_A は別のものなのに同じ記号を使ったら混乱しないでしょうか？」

「とりあえず、対象は大文字で、射は小文字で書くようにすれば混乱はないと思うよ。でも、混乱しそうなときは F_O や F_A を使うようにすればいいね」

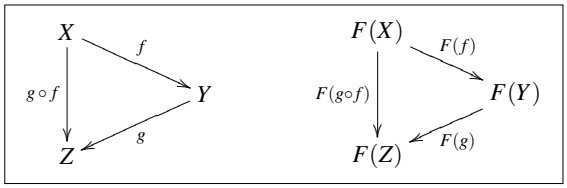
「なるほど、話を戻すと条件 **(func2)** は、今の図式が可換になる事だと言い換えられるわけですね！」

「そうだね――。」

今の図式のように一箇所も合成が入らないものを可換図式と呼ばない流儀もある。しかし、可換性の定義を明確には教えていなかったから、細かい話はやめておくことにした。ここではこの例のように一箇所も合成を含まないような可換図式も許すことにしておこう。

「さて、条件 **(func3)** はこんな感じですかね」

そう言って扇ちゃんはこんな図式を書いた：

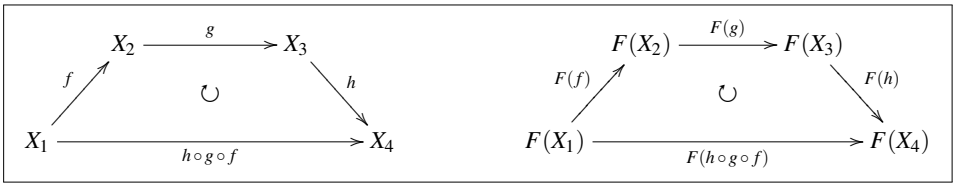


「射の等式 $F(g)\circ F(f) = F(g\circ F)$ が成立する事が右の図式の可換性として表現されてるね。今の図式だと、右側の可換図式は『左側の可換図式を関手 F で移したもの』と思えるよね」

「……なるほど、確かにそうですね。でもこんな簡単な奴だけじゃなくて、もっと大きな可換図式でも関手 F で『移せる』んですか？」

「そうだね、例えばちょっとだけ大きくした例だとこんな風になるかな」

そう言って私は次のような図式を書いた：



「あの――、このくるっとした矢印はなんですか？」

扇ちゃんが『↻』のあたりを指さした。

「ああ、これはこの『丸まった矢印』を取り巻く射で作られた部分図式が可換である事を意味する符丁だね。あとで出てくるけど、もう少し大きな図式を使った証明をしてると『この部分が可換だということはわかってるけれど他の部分が可換かどうかはまだわかってない』という場面によく出くわすの。この記号を可換性がすでに成り立ってる場所を書いておくと証明の見通しが良くなるんだ」

「なるほど。じゃあ、右側の図式の可換性をチェックしてみますね。対象 $F(X_1)$ から $F(X_2)$ へ向かう、下側の射が $F(h\circ g\circ f)$ で、上側の射が左から順番に $F(f),F(g),F(h)$ だな。んで、これを全部合成すると

$$\begin{aligned} F(h)\circ F(g)\circ F(f) &= (F(h)\circ F(g))\circ F(f) \\ &= F(h\circ g)\circ F(f) && \text{：条件 \textbf{(func3)} により} \\ &= F((h\circ g)\circ f) && \text{：条件 \textbf{(func3)} により} \\ &= F(h\circ g\circ f) \end{aligned}$$

となるから、なるほど確かに上を通っても下を通っても『同じこと』になりますね」

「Haskell の型クラス `Functor` との関係はわかるかな？」

「ええと、例えば `Maybe` の場合だと次の法則が成り立ってました：

(Maybe-1) `f :: X -> Y ⇒ fmap f :: Maybe X -> Maybe Y`

(Maybe-2) `fmap id = id`

(Maybe-3) `fmap (f . g) = (fmap f) . (fmap g)`

Haskell では `id` が多相関数になっちゃってるので少し対応が見づらいですが、`fmap` が関手の『射を移すやつ』で、`Maybe` が関手の『対象を移すやつ』だと思えばいいみたいですね。実際、上みたいに書いてみると

丁度関手の定義 (p.374) の条件 **(func1)**-**(func3)** と対応してます」
「そうだね。まとめるとこんなふうになるかな」
そう言って私は次のような表を書いた (表 26.3)：

圏 **Hask** の言葉	Haskell の言葉
Hask から **Hask** への関手 $F = \langle F_O, F_A \rangle$	なし
Hask から **Hask** への関手の object-part F_O	`Functor` インスタンスの型コンストラクタ
Hask から **Hask** への関手の arrow-part F_A	`Functor` インスタンスについて定義される `fmap`

表 26.3: Haskell の圏 **Hask** と Haskell の言葉の対応 (2)

「圏 **Hask** における関手に対応する『Haskell の言葉』が『なし』というのはちょっと意外です」
「ここでは関手 F というのは、対象を対象に移す写像 F_O と射を射に移す写像 F_A の対 $\langle F_O, F_A \rangle$ のことだと決めたから、それに厳密に従えばそういうことになるよね」
「じゃあ、うかつに『`Maybe` は圏論における関手』とか言わないほうが良いんですか？」
「話をしてる人の間で、何をどんなふうに省略するかについての合意がとれているなら気にしなくていいと思うよ」
「なるほど、合意、ですか」
「というわけで、合意してもらえるなら『Maybe 関手』という言葉使いをしましょう」
「なるほど、じゃあ合意しますってことで」
「付け加えるとしたら、圏論での議論では写像 F_O と写像 F_A に同じ記号を使う事が多いから、圏論の議論を Haskell に翻訳したいときには、自分が翻訳しようとしている関手の記号が『対象を移す写像』なのか『射を移す写像』なのか注意しないといけないとは思う」
「あ、なんか実際それ気にしないと即死フラグになりそうですね」

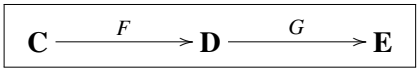
26.4.2 自己関手の簡単な性質

「今の Maybe 関手の場合は、出発と行き先が同じ圏 **Hask** になってるよね。一般に、出発と行き先が同じ圏 **C** であるような関手を **C** 上の**自己関手 (endofunctor)** と言うんだ。自己関手はモナドを説明するときにも必要だから、簡単な性質についてまとめておこうね。まずは合成の話から――」
そう言って私は次のようにノートに書きだした：

自己関手の合成
圏 **C** 上の自己関手 F, G を考える。このとき、F と G の合成関手 $G \cdot F = \langle (G \cdot F)_O, (G \cdot F)_A \rangle$ を次の条件 **(fcomp1)(fcomp2)** を満たすものとして定義する:

(fcomp1) $(G \cdot F)_O : X \longmapsto (G_O) \circ (F_O)\ X.$
(fcomp2) $(G \cdot F)_A : f \longmapsto (G_A) \circ (F_A)\ f.$

「見てすぐ思ったんですけど、この合成関手ってのは自己関手じゃなくても定義できそうですね？」
「そうだね、一般に圏 **C**, **D**, **E** があったとき、この圏たちがこんなふうに関手で結ばれていたとすると――」
私はノートに圏を頂点とし、関手を辺とするグラフを書いた：



「――この場合にも合成関手 $G \cdot F$ が考えられるよ。自己関手の合成はこれの特殊な場合、つまり全部の圏が同じ場合だね」
ものを教えるときに、あまりにも一般的に教えてしまうと生徒は置いてけぼりにされてしまうが、最小限に絞って教えても生徒の余計な疑問を招いてしまうことがある。この辺のバランスは難しい。
「さて、関手の合成についても結合法則が成り立つんだよ」

自己関手の合成の結合法則
圏 **C** 上の自己関手 F, G, H を考える。このとき、関手の合成に関して結合法則

$$\textbf{(assoc)} \quad (H \cdot G) \cdot F = H \cdot (G \cdot F)$$

が成り立つ。

「これはわかります。合成関手ってのは対象と射の写像をそれぞれ合成したものから作ったペアとして定義したんだから、写像の合成の結合法則からこれは簡単に言えますね」
どうやら丁寧に証明を書き下すまでもなさそうだ。先に進もう。
「そんなわけで、こんなふうに自己関手 F のべき乗 F^n を定義すると……

$$\textbf{(funcexp1)} \quad F^1 := F.$$
$$\textbf{(funcexp2)} \quad F^{n+1} := F \cdot F^n \qquad (^\forall n > 0).$$

——こんなことが言えるよ：

$$\forall m, n \in \mathbb{N} \quad F^m \cdot F^n = F^{m+n}.$$

「……ええと、これっていかにもそうなりそうな気もしますが、どうやって証明するんですか？」
一般に、集合 X 上の二項演算子 $\star$ が結合法則を持つ場合 $a, b, c, d, e \in X$ から作った項たち、例えば

$$a\star(b\star(c\star(d\star e))), (a\star b)\star(c\star(d\star e)), ((a\star b)\star c)\star(d\star e)$$

は全て同じ物を表している事が手計算で示せる。このような集合 X の上ではさらに「一般の結合法則」と呼ばれるものが成り立つ事を示せる。いま問題にしている $F^m \cdot F^n = F^{m+n}$ の証明も、自己関手の集合の上の合成演算 $(\cdot)$ に関する「一般の結合法則」に帰着できる。一般の結合法則は（普通の）結合法則を数学的帰納法と組み合わせれば示せるのだが、任意の順序で組み合わされた項をどう扱うか、が工夫を要するポイントだ。分かってしまえば簡単なのだが、説明されて簡単にわかるくらいの子なら、そもそもこんな質問はしないだろう。うん、一般の結合法則に帰着して示すのはやめておこう。
代わりに、私はいま問題にしている累乗の形に特化した方法で示すことにした。
「示そうとしてる命題を

$$\forall m \in \mathbb{N} \left[\forall n \in \mathbb{N} \quad F^m \cdot F^n = F^{m+n} \right].$$

という形に整理[*6]してみると、m についての数学的帰納法で簡単に示せるよ」
「そうなんですか？　ちょっとやってみますね。まず、$m = 1$ の場合、示そうとする命題は

$$\forall n \in \mathbb{N} \quad F^1 \cdot F^n = F^{1+n}$$

ってなるけど、**(funcexp1)** より $F^1 = F$ だったから **(funcexp2)** から、確かに等号は成立しますね。
次に、ある $m_0 \geq 1$ で成り立ってると仮定する（帰納法の仮定）と、つまり

$$\forall n \in \mathbb{N} \quad F^{m_0} \cdot F^n = F^{m_0+n}$$

が成り立ってると仮定したことになりますけど、これを使えば

[*6] 本章では、0 は自然数の集合 $\mathbb{N}$ の元ではないと約束しておく。

$$
\begin{aligned}
F^{m_0+1}\cdot F^n &= (F\cdot F^{m_0})\cdot F^n && \text{:(funcexp2) より}\\
&= F\cdot (F^{m_0}\cdot F^n) && \text{:(assoc) より}\\
&= F\cdot F^{m_0+n} && \text{: 帰納法の仮定より}\\
&= F^{m_0+n+1} && \text{:(funcexp2) より}\\
&= F^{(m_0+1)+n}.
\end{aligned}
$$

となるから、確かに『m_0+1 の場合』にも等号が成り立ちますね。なんか m,n と出てきたから数学的帰納法を二重に組み合わせたり複雑な事をしないといけないのかとおもって思考停止してたんですけど意外と簡単でしたね」

ふふ、誘導してあげるまで全然自信なさそうだったのに「意外と簡単だ」なんて言い出してるあたりが図々しくて可愛い。さて、モナドの説明に必要だから恒等関手の話もしておこう。

「さて、ところで自己関手 F に対して F^0 はなんだと思う？」

「なんだって言われても、まだ定義してないじゃないですか……」

関手 F の「累乗」F^n は、$n\geq 1$ についてだけ定義したので扇ちゃんの言っていることは正しい。しかし――。

「じゃあ、『もし F^0 があったら』って考えてみると――？」

「もしあったら、ですか。そうですね……さっきの性質 $F^m\cdot F^n=F^{m+n}$ が $m=0$ や $n=0$ の場合に成り立ってくれてる事を期待して言えば F^0 はこんな感じの性質を持っていて欲しいですね：

$$\forall n\in\mathbb{N}\qquad F^0\cdot F^n = F^n\cdot F^0 = F^n.$$

「そんな関手 $I=\langle I_O, I_A\rangle$ は、例えばこんなふうに作れるよね――」

(id-functor1) $I_O : X \longmapsto X.$
(id-functor2) $I_A : f \longmapsto f.$

「んー、なるほど、何もしないやつですか。これなら F^0 とみなすのに丁度いいですね……って F 関係なくないですか？」

「多項式を考えるときも x に何を代入するかと関係なく $x^0=1$ とする、みたいなものだと思えばいいと思うよ」

「なるほど。ところでこれにもなんか名前あるんですか？」

「**恒等関手**っていう名前があるよ」

「こうとうかんしゅ、ですか。こうとう、は『恒等式』の『恒等』ですかね」

「そうだね。さて、F^0 まで自然に考えられるようになったので、さっきの定理をちょっとだけ拡張してこんなふうに述べられるね」

$$\forall m,n\in\mathbb{N}_0\quad F^m\cdot F^n \;=\; F^{m+n}.$$

「下にゼロがついたこれですけど、$\mathbb{N}_0 := \{0\}\cup\mathbb{N}$ ということですか？」

「そうだね――ごめんね説明省いちゃって。さて、もう一つ重要な公式を書いておくよ――

$$\forall X\in \mathrm{Ob}(\mathbf{C})\quad \forall m,n\in\mathbb{N}_0\quad \left(F^m\right)_A\left(\mathrm{id}_{F^n(X)}\right) \;=\; \mathrm{id}_{F^{m+n}(X)}.$$

――ただし、『射を移す写像』だけ $_A$ をつけておいたよ」

「よく考えれば $_A$ がついてなくてもどっちがどっちかはわかりますけど、この方が確かに私にはわかりやすいです」

「証明はできるかな？」

「えーと、あれ……これってさっきと同じパターンで行けるのでは？　ちょっとやってみますね。対象 X は固定されてるから省略することにして、さらに示そうとしてる命題の『$\forall$』を m と n で分けて

$$\forall m \in \mathbb{N}_0 \left[\forall n \in \mathbb{N}_0 \quad \left(F^m\right)_A \left(\mathrm{id}_{F^n(X)}\right) = \mathrm{id}_{F^{m+n}(X)} \right].$$

と、まず書いてみます。そして、m についての数学的帰納法で示せないか考えてみますね。まず、$m=0$ の場合は、

$$\begin{aligned}
&(m=0 \text{ のときの左辺}) = \left(F^0\right)_A \left(\mathrm{id}_{F^n(X)}\right) = I_A \left(\mathrm{id}_{F^n(X)}\right) = \mathrm{id}_{F^n(X)}. \\
&(m=0 \text{ のときの右辺}) = \mathrm{id}_{F^{0+n}(X)} = \mathrm{id}_{F^n(X)}.
\end{aligned}$$

となりますから、$m=0$ のときの等号は余裕で成立します」

「次は $m=m_0\,(\geq 0)$ のときに成立したと仮定すると、$m=m_0+1$ のときにも成立する、という部分だね」

「はい、そっちもやってみます――。

$$\begin{aligned}
&(m=m_0+1 \text{ のときの左辺}) \\
&= \left(F^{m_0+1}\right)_A \left(\mathrm{id}_{F^n(X)}\right) \\
&= \left(F^{m_0} \cdot F^1\right)_A \left(\mathrm{id}_{F^n(X)}\right) \\
&= \left(\left(F^{m_0}\right)_A \circ \left(F^1\right)_A\right) \left(\mathrm{id}_{F^n(X)}\right) && : \textbf{(fcomp2)} \text{ より} \\
&= \left(F^{m_0}\right)_A \left(\left(F^1\right)_A \left(\mathrm{id}_{F^n(X)}\right)\right) \\
&= \left(F^{m_0}\right)_A \left(\mathrm{id}_{F^1 \cdot F^n(X)}\right) && : \textbf{(func2)} \text{ より} \\
&= \left(F^{m_0}\right)_A \left(\mathrm{id}_{F^{n+1}(X)}\right) \\
&= \mathrm{id}_{F^{m_0+(n+1)}} = \mathrm{id}_{F^{(m_0+1)+n}} && : \text{帰納法の仮定より} \\
&= (m=m_0+1 \text{ のときの右辺})
\end{aligned}$$

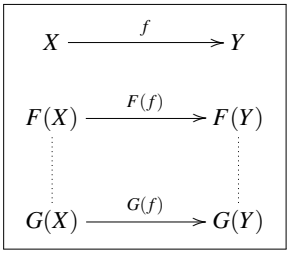
となって確かに示せました」

「よくできました」

26.5　自然変換

「さて、関手というのは可換図式を可換図式に移してくれるものだったね。今から紹介する**自然変換**は、二つの関手を結んでくれる何か、と言えるかもしれない。例えば圏 **C** から **D** への関手 F と G があったとして、圏 **C** の対象 X,Y に対してこんな図式を書いてみましょう」

そう言って私は次のように書いた：



「図式、ってだけだからまだ可換性は要求してないわけですね」

「そうだね――そして、いま点線で書いたところを上から下に射で結んで 4 つの対象 $F(X),F(Y),G(X),G(Y)$ を結ぶ可換図式ができるようにしてくれるものを『関手 F から関手 G への**自然変換 (natural transform)**』というんだ」

「なんかあやふやな話に思えるんですが、そもそも、そういうのっていつでも存在するんですか？」

やや疑わしげな声で扇ちゃんが尋ねた。

「ううん、いつでもあるわけじゃない――『もしそういうものがあったらそう呼びます』という話。さて、自然変換のきちんとした定義を書いておこうね」

自然変換の定義

C と **D** を圏とし、F と G を圏 **C** から圏 **D** への関手とする。このとき、写像

$$\theta\colon \mathrm{Ob}(\mathbf{C}) \to \mathrm{Arr}(\mathbf{D})$$

が関手 F から関手 G への**自然変換 (natural transform)** であるとは、次の **(natx1) (natx2)** が成立することである：

(natx1) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad \theta_X \in \mathrm{Hom}\big(F_O(X), G_O(X)\big).$

(natx2) $\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad \theta_Y \circ F_A(f) = G_A(f) \circ \theta_X.$

写像 θ が関手 F から関手 G への自然変換であることを記号で

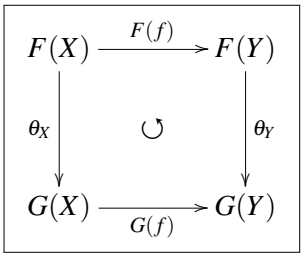
$$\theta\colon F \xrightarrow{\bullet} G$$

と表す。

「ええと、条件 **(natx1)** は要するに各 $X \in \mathrm{Ob}(\mathbf{C})$ について

$$F(X) \xrightarrow{\theta_X} G(X)$$

ってことですね。条件 **(natx2)** は少し複雑そうなので図式にしてみますね」
そう言って扇ちゃんは次のような図式を書いた：

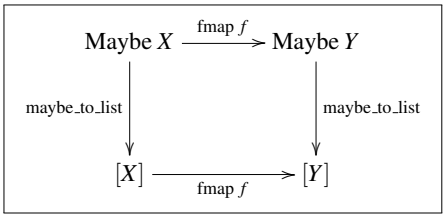


「——書いてはみましたが、なんのこっちゃという感じがします。そもそもなんでこんなものを考えたいのでしょう？」

「例えば Haskell でこんな型シグネチャを持つ多相関数があったとき——

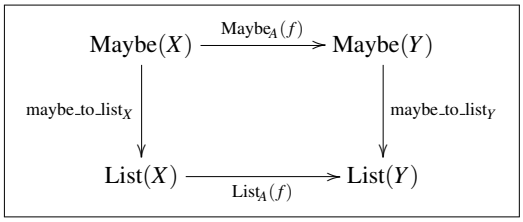
```
maybe_to_list :: Maybe a -> [a]
```

——この関数が圏 **Hask** における、Maybe 関手からリスト関手への自然変換であるということは次の図式の可換性が成立することを意味するわ」



「この書き方だと射についてるラベルが Haskell での呼び名に近くなってますけど、もっと圏論ぽく書けばこんなふうになる[*7]んですよね？」

[*7] 圏 **Hask** におけるリスト関手の名称を List にしてあります。



「うん、そうだね」

「この可換性を Haskell の言葉に戻すと

```
maybe_to_list . (fmap f) = (fmap f) . maybe_to_list
```

ですね。なるほど、こういう関係が成り立ってると便利な場面はあるかもしれません」

「今の例は圏 **Hask** から取ったけど、どんな圏にもある身近な自然変換の例は id かな——ほら！」

(id:natx1) $\forall X \in \mathrm{Ob}(\mathbf{C}) \quad \mathrm{id}_X \in \mathrm{Hom}(X,X)$.
(id:natx2) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y) \quad \mathrm{id}_Y \circ f = f \circ \mathrm{id}_X$.

「えーと、これって圏の定義の id に関係する公理を繰り返しただけじゃないですか？」

「そうだけど、恒等関手と自然変換の定義を念頭に見直すと

$$\mathrm{id}: I \xrightarrow{\bullet} I$$

であることが読み取れるでしょ」

「ええと、対象 X は恒等関手 I で移された $I_O(X)$ だとみなせて、射 f は恒等関手 I で移された $I_A(f)$ だと見なせるから……なるほど、確かに id は恒等関手 I から I 自身への自然変換になってますね」

26.6 計算の抽象化と修飾された型

「さて、いままで圏の定義、関手、自然変換まで話してからモナドの話をする準備は整ったんだけど、その前に Haskell とモナドの関係を説明するために『関手を使った計算の抽象化』の話をしておこうね」

「なるほど」

「たとえば、関数 $f: \mathbb{R} \to \mathbb{R}$ を受け取って、その零点[*8]を返してくれるアルゴリズムを Haskell で書く[*9]としたら

```
getSomeNullPoint :: (Real -> Real) -> Real
```

という型シグネチャを持つ関数 `getSomeNullPoint` を考えてもいいけど、入力される関数 f が $\lambda x \mapsto (x^2+1)$ みたいに零点を持たない関数かもしれないときは

```
getSomeNullPointMaybe :: (Real -> Real) -> Maybe Real
```

という型シグネチャを持つ関数 `getSomeNullPointMaybe` を考えるのが自然だし、複数個の零点を列挙して取り出したい場合は

```
getNullPointList :: (Real -> Real) -> [Real]
```

[*8] 実数 x が関数 $f: \mathbb{R} \to \mathbb{R}$ の零点であるとは、$f(x) = 0$ が成立することを意味します。

[*9] ここでは実装の詳細は問題にしていません。

という型シグネチャを持つ関数 `getNullPointList` を考える方が自然だよね」

「なるほど、そうかもしれません」

「今の例では Maybe もリストも圏 **Hask** の上の自己関手だったから、これを一般化すれば自己関手 F を使って

$$\boxed{X \to F(Y)}$$

として計算を抽象化できる」

「なるほど、それはそうかもしれませんし自己関手 F が Maybe やリストの場合には納得しますが、もう少し気の利いた例はないですか？」

「気の利いた例——かどうかはわからないけど、状態を持つ関数なんかも関手を使ってこのタイプの計算だとみなせるよ」

「あ、それ知りたいです」

「まず、状態を持つ関数は C++ ではこんな感じに書けるよね」

```
Y  stateful_func(X x){
    static S state = s_0;
    //変数 state は関数の内部状態を表現している。初期状態を s_0 にする。
    S next_s = new_state(x, state);
    Y ret = return_val(x, state);
    state = next_s; //状態の更新をする。
    return ret;
}
```

「Haskell の話してたのに、なんで突然 C++ が……って思ったけど確かに Haskell は参照透明だからこういう『呼ぶ度に違う値を返す関数』は素直に書けないんですね」

「うん。上の擬似コードでは、状態を持つ型 S を使って状態を内部に保持してるけど、これを外に出してしまえば Haskell でも状態を扱う関数をこんな型シグネチャを持つ関数として書ける——」

```
function_with_state :: (X, S) -> (Y, S)
```

「なるほど。確かにそれはそうですが、内部状態の変数を使う人が管理するようになるから、あまり便利そうじゃないですね」

「さて、今の関数を数学的な記号で書けば

$$\boxed{\text{function\_with\_state}\colon X \times S \to Y \times S}$$

となるけど、カリー化

$$\boxed{\begin{array}{ccc} \text{curry}\colon \mathrm{Map}(A \times B, C) & \longrightarrow & \mathrm{Map}\big(A, \mathrm{Map}(B, C)\big) \\ f & \longmapsto & \text{curry}(f) \end{array}}$$

を適用したものはこんな形になるよね」

$$\boxed{\text{curry}(\text{function\_with\_state})\colon X \to \mathrm{Map}\big(S, Y \times S\big)}$$

「Haskell の関数だと思ったときの型シグネチャを書くと

```
    function_with_state_curried :: X -> (S -> (Y, S))
```

ですね。えーと……ああ、もしかしてこれの後ろ側って関手なんですか？」

「状態を表す Haskell の型を S をパラメータとする関手 $\mathrm{State}^S = \langle \mathrm{State}^S_O, \mathrm{State}^S_A \rangle$ を

$$\mathrm{State}^S_O(X) := \mathrm{Map}(S, X \times S),$$
$$\mathrm{State}^S_A(f)\colon\ \mathrm{Map}(S, X \times S) \longrightarrow \mathrm{Map}(S, Y \times S)$$

ただし

$$\mathrm{State}^S_A(f)(\varphi) := \lambda s \mapsto \langle f(\varphi_1(s)), \varphi_2(s)\rangle .$$

ここで、$\varphi(s) = \langle \varphi_1(s), \varphi_2(s)\rangle$ とした。

——こんなふうに定義できるよ」

「ぱっと見たとき、射を移す写像 State^S_A が複雑に思えたんですけど $f\colon X \to Y$ を使う縛りで考えれば、自然にこの形にたどり着きますね」

「実際、関手 F を定義するときに対象を移す写像 F_O の定義だけ与えて、F_A は考えればわかるはずだから省略します、という場合も結構あるよ」

「なるほど。ところで——State^S が実際に関手であることの証明がまだですよね——？」

「そうだね、面倒だけど難しくないから、あとで自分でやってみたら？」

「わかりました。あとでやってみます。とりあえず、$X \to \mathrm{State}^S(Y)$ の形で状態を持つ関数を表せることは納得しました」

なんでも細かく教えてしまったら、いつか自分の頭で考えることもできなくなってしまう。もちろん、最初から何も教えないのもかわいそうだ。バランスが大事だというクリシェよりましな一般論は残念ながら存在しない。

「例は少なかったけど、とりあえず $X \to F(Y)$ みたいな形で色々なタイプの計算が表せそうだということは納得してもらえた？」

「はい」

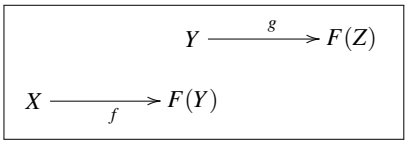
26.6.1 抽象化された計算とその合成

「さて、関手 F を使って $X \to F(Y)$ という形で計算を抽象化してみると、その『合成』を考えたくなる」

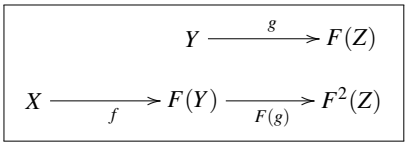
「例えばどんな場合ですか？」

「そうだね——たとえば、状態の関手 State^S を使った計算 $X \to \mathrm{State}^S(Y)$ と $Y \to \mathrm{State}^S(Z)$ を『合成』して $X \to \mathrm{State}^S(Z)$ にしたい場合かな」

「それって、できたら便利そうだけど実際難しそうですね。だって、関手 F で図式を書いてみると



となって、合成のとっかかりがありません。あ——でもなんか思いつきました。F は関手なんだから今の図式の g を F で移せて、

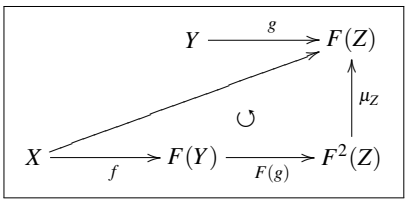


みたいのができます。あれ……これを合成しても $X \to F^2(Z)$ にしかならないですね。うまく行きそうだとおもったんですけど」

「いや、その方向でいいんだよ。今の話に足りないのは、自然変換

$$\mu\colon F^2 \xrightarrow{\bullet} F$$

だよ。もしこんな自然変換 μ があったとすれば、こんな図式を作れるよね――」



こうすれば、圏 **Hask** の射 $f\colon X \to F(Y)$ と射 $g\colon Y \to F(Z)$ の『合成』として

$$\boxed{\mu_Z \circ F(g) \circ f\colon X \to F(Z)}$$

が使えそうだよね」

「ええと、それっぽく聞こえる話ですが、そういう自然変換 μ がないといけないし、それが合成らしくちゃんと結合法則を満たすんだろうか、とか気になります」

「この後でちゃんとした定式化をするけど、『関手 F がモナドである』というのは、$X \to F(Y)$ の形の射が

$$\boxed{\begin{array}{ccc} \mathrm{Hom}(Y,F(Z)) \times \mathrm{Hom}(X,F(Y)) & \longrightarrow & \mathrm{Hom}(X,F(Z)) \\ \langle g,f\rangle & \longmapsto & \mu_Z \circ F(g) \circ f \end{array}}$$

を合成とする圏をなす事として定義できるんだ」

「自己関手 F がモナドになる条件って自然変換 $\mu\colon F^2 \xrightarrow{\bullet} F$ の存在だけじゃ足りないんですか？」

「結論から言えば足りないよ。例えば、$X \to F(Y)$ の形の射を基礎にするならば、恒等関手に相当するものは $X \to F(X)$ の形をしているはずで、このことから自然変換

$$\boxed{\eta\colon I \xrightarrow{\bullet} F}$$

も欲しくなるよね。でも、これ以上話を進める前に定理の形で整理して条件を述べましょう」

Kleisli 圏とモナド則

$\mathbf{C}$ を圏とし、F を $\mathbf{C}$ 上の自己関手とする。さらに、二つの写像 $\eta, \mu\colon \mathrm{Ob}(\mathbf{C}) \to \mathrm{Arr}(\mathbf{C})$ が存在し、次の **(eta-nat-1) (mu-nat-1)** が成立しているものとする：

$$\begin{aligned} &\textbf{(eta-nat-1)}\ \forall X \in \mathrm{Ob}(\mathbf{C}) \quad \eta_X \in \mathrm{Hom}\left(X, F(X)\right), \\ &\textbf{(mu-nat-1)}\ \forall X \in \mathrm{Ob}(\mathbf{C}) \quad \mu_X \in \mathrm{Hom}\left(F^2(X), F(X)\right). \end{aligned}$$

このとき、集合 $\overline{O}, \overline{A}$ を次のように定義する：

$$\begin{aligned} &\overline{O} := \mathrm{Ob}(\mathbf{C}), \\ &\overline{A} := \bigcup_{X,Y \in \mathrm{Ob}(\mathbf{C})} \mathrm{Hom}(X,F(Y)) \\ &\qquad \left(= \left\{ f \,\middle|\, \exists X, Y \in \mathrm{Ob}(\mathbf{C})\ \ f \in \mathrm{Hom}(X,F(Y)) \right\}\right). \end{aligned}$$

また、$f\colon X \to F(Y)$ に対し、

$$\overline{\mathrm{dom}}(f) := X, \quad \overline{\mathrm{cod}}(f) := Y.$$

と定義する。さらに、集合

$$\begin{aligned} \mathrm{Comp}(\overline{A}) := &\bigcup_{X,Y,Z \in \mathrm{Ob}(\mathbf{C})} \mathrm{Hom}\big(Y,F(Z)\big) \times \mathrm{Hom}\big(X,F(Y)\big) \\ &\left(= \left\{ \langle g,f\rangle \in \overline{A} \times \overline{A} \;\middle|\; \overline{\mathrm{dom}}(g) = \overline{\mathrm{cod}}(f) \right\}\right) \end{aligned}$$

を定義域とする写像

$$\bar{\circ}\colon \mathrm{Comp}\big(\overline{A}\big) \to \overline{A}$$

を、次のように定義する：

$$\begin{array}{ccc} \mathrm{Hom}\big(Y,F(Z)\big) \times \mathrm{Hom}\big(X,F(Y)\big) & \longrightarrow & \mathrm{Hom}\big(X,F(Z)\big) \\ \langle g,f\rangle & \longmapsto & \mu_Z \circ F(g) \circ f \end{array}$$

このとき、次のことが言える：

定理：条件 **(K1)-(K5)** が全て成立する $\iff$ 条件 **(M1)-(M5)** が全て成立する

ただし、条件 **(K1)-(K5)** と条件 **(M1)-(M5)** は次のとおり：

(K1) $\eta : I \xrightarrow{\bullet} F$　(η が自然変換であること)

(K2) $\mu : F^2 \xrightarrow{\bullet} F$　(μ が自然変換であること)

(K3) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,F(Y))$　　$\eta_Y \ \bar{\circ}\ f = f$.　($\eta_Y$ の左単位条件)

(K4) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,F(Y))$　　$f \ \bar{\circ}\ \eta_X = f$.　($\eta_X$ の右単位条件)

(K5) $\forall \langle h,g\rangle, \langle g,f\rangle \in \mathrm{Comp}\big(\overline{A}\big)$　　$h \ \bar{\circ}\ (g \ \bar{\circ}\ f) = (h \ \bar{\circ}\ g) \ \bar{\circ}\ f$.　($\bar{\circ}$ の結合法則)

(M1) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y)$　　$F(f) \circ \eta_X = \eta_Y \circ f$

(M2) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y)$　　$F(f) \circ \mu_X = \mu_Y \circ F^2(f)$

(M3) $\forall X \in \mathrm{Ob}(\mathbf{C})$　　$\mu_X \circ F(\eta_X) = \mathrm{id}_{F(X)}$.

(M4) $\forall X \in \mathrm{Ob}(\mathbf{C})$　　$\mu_X \circ \eta_{F(X)} = \mathrm{id}_{F(X)}$.

(M5) $\forall X \in \mathrm{Ob}(\mathbf{C})$　　$\mu_X \circ F(\mu_X) = \mu_X \circ \mu_{F(X)}$.

自己関手 F が条件 **(K1)-(K5)** を全て満たすとき $\langle F,\eta,\mu\rangle$ を**モナド (monad)** と呼ぶ。（しかしながら、自然変換 η と μ が議論の文脈から明らかな場合は「F はモナドである」という略式の言葉使いをする場合もある。）

$\langle F,\eta,\mu\rangle$ がモナドのとき、$\mathrm{Kleisli}(\mathbf{C}) := \big\langle \overline{O},\overline{A},\overline{\mathrm{dom}},\overline{\mathrm{cod}},\eta,\bar{\circ}\big\rangle$ は明らかに圏になる。この圏 $\mathrm{Kleisli}(\mathbf{C})$ を**モナド $\langle F,\eta,\mu\rangle$ が定義する Kleisli 圏**と呼ぶことにする。

「あ……しゃべっていいですか？」

「うん、いいよ」

「最初の 5 つの条件 **(K1)-(K5)** は Kleisli 圏の合成『$\bar{\circ}$』について述べられていて、次の条件 **(M1)-(M5)** は圏 **C** 本来の合成『$\circ$』について述べられてるんですね」

「そうだね、そういう意味では Kleisli 圏の条件を圏 **C** の言葉に翻訳したのが条件 **(M1)-(M5)** という見方ができるね」

「条件 **(M1)-(M5)** は Haskell の Wikibooks に出てるのとおなじ奴ですか？」

「うん、条件の順番や記号は一部変えてあるけど実質的には同じものだよ」

「条件 **(M1)-(M5)** を Haskell っぽくかきなおしてみるとこんな感じですね」

```
(M1')  (fmap f) . eta  = eta . f
(M2')  (fmap f) . mu   = mu . (fmap (fmap f) )
(M3')  mu . (fmap eta)  = id
(M4')  mu . eta  = id
(M5')  mu . (fmap mu)  = mu . mu
```

「じゃあまず定理を証明しちゃいましょう」

「ちょっと証明が長くなりそうなので、証明の概略というか流れというかそんなのを先に教えていただけ

ると助かります」
なるほど、たしかにそれは大事かもしれない。
「まず、条件 **(K1)-(K5)** から条件 **(M1)-(M5)** を示すときの流れはこんなふうになるよ」

(K1) $\Longrightarrow$ **(M1)**.
(K2) $\Longrightarrow$ **(M2)**.
(K3) $\Longrightarrow$ **(M3)**.
(K4) $\Longrightarrow$ **(M4)**.
(K5) $\Longrightarrow$ **(M5)**.

「えらくストレートですが、逆ももしかして同じですか？」
「後ろの方だけ違うかな。そっち向きの証明の流れはこんなふうになるよ」

(M1) $\Longrightarrow$ **(K1)**.
(M2) $\Longrightarrow$ **(K2)**.
(M3) $\Longrightarrow$ **(K3)**.
(M4), **(M1)** $\Longrightarrow$ **(K4)**.
(M5), **(M2)** $\Longrightarrow$ **(K5)**.

「じゃあ条件 **(K1)-(K5)** から条件 **(M1)-(M5)** を示すところを順番に片づけましょう」

26.6.2 条件 (K1) が (M1) を導くこと

「条件 **(K1)** は $\eta : I \xrightarrow{\bullet} F$ だから、自然変換の条件 **(natx2)** から

$$\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad \eta_Y \circ f = F(f) \circ \eta_X$$

であることがわかるね。よって条件 **(K1)** から条件 **(M1)** が導けたね」

26.6.3 条件 (K2) が (M2) を導くこと

「条件 **(K2)** も自然変換であるという条件ですから、いまのと同じ風にできそうですね。実際、μ が自然変換である条件とすると、さっきと同じように条件 **(natx2)** から

$$\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad \mu_Y \circ F^2(f) = F(f) \circ \mu_X$$

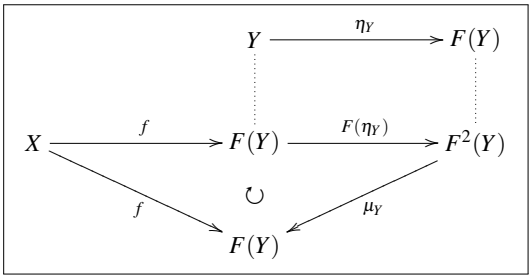
が出てきて、確かに条件 **(K2)** から条件 **(M2)** が導けます。あれ……なんかひょっとしてすごく簡単？」

26.6.4 条件 (K3) が (M3) を導くこと

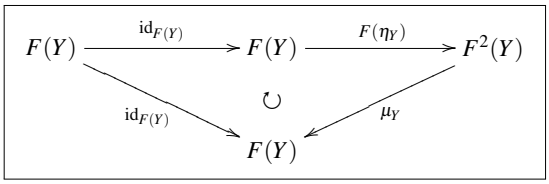
「ふふ、確かに今のところ結構簡単だね。じゃあ条件 **(K3)** から条件 **(M3)** を導くところをやってみましょう。まず、もう一度条件 **(K3)** を書いてみると

$$\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X, F(Y)) \qquad \eta_Y \ \bar{\circ}\ f = f$$

だけど、これを可換図式で表すと



となるね。いま、$X,Y \in \mathrm{Ob}(\mathbf{C})$ は任意、つまりどうとってもいいので $X := F(Y),\ f := \mathrm{id}_{F(Y)}$ としてやると、こんな図式になるよね：



こうして出来た図式の可換性の条件を書いてみると任意の $Y \in \mathrm{Ob}(\mathbf{C})$ について成立する次の関係式が得られる：

$$\boxed{\mu_Y \circ F(\eta_Y) \circ \mathrm{id}_{F(Y)} = \mathrm{id}_{F(Y)}.}$$

ここで、左辺に出てくる $\mathrm{id}_{F(Y)}$ は合成の単位元だから消してやれば

$$\boxed{\mu_Y \circ F(\eta_Y) = \mathrm{id}_{F(Y)}}$$

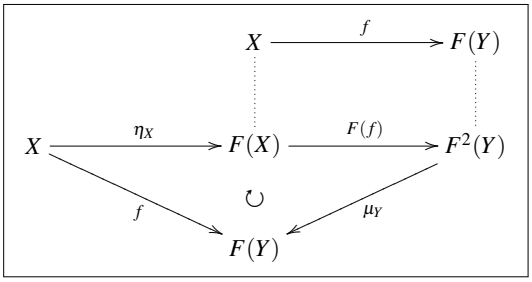
となる。よって、確かに条件 **(K3)** から条件 **(M3)** が導けたね」

26.6.5　条件 (K4) が (M4) を導くこと

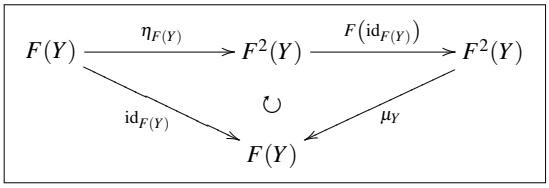
「条件 **(K4)** から条件 **(M4)** を導くところをやってみましょう。まず、もう一度条件 **(K4)** を書いてみると

$$\boxed{\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,F(Y)) \qquad f \ \bar{\circ}\ \eta_X = f}$$

だけど、これをさっきみたいに可換図式で表すと



今度は $X := F(Y),\ f := \mathrm{id}_{F(Y)}$ とするとこんな図式になるね：



こうして出来た図式の可換性の条件を書いてみると任意の $Y \in \mathrm{Ob}(\mathbf{C})$ について成立する次の関係式が得られるね：

$$\boxed{\forall Y \in \mathrm{Ob}(\mathbf{C}) \quad \mu_Y \circ F\left(\mathrm{id}_{F(Y)}\right) \circ \eta_{F(Y)} = \mathrm{id}_{F(Y)}.}$$

自己関手の性質 (p.378) より $F\left(\mathrm{id}_{F(Y)}\right) = \mathrm{id}_{F^2(Y)}$ なので結局次が言える：

$$\boxed{\forall Y \in \mathrm{Ob}(\mathbf{C}) \quad \mu_Y \circ \eta_{F(Y)} = \mathrm{id}_{F(Y)}.}$$

よって、確かに条件 **(K4)** から条件 **(M4)** が導けたね」

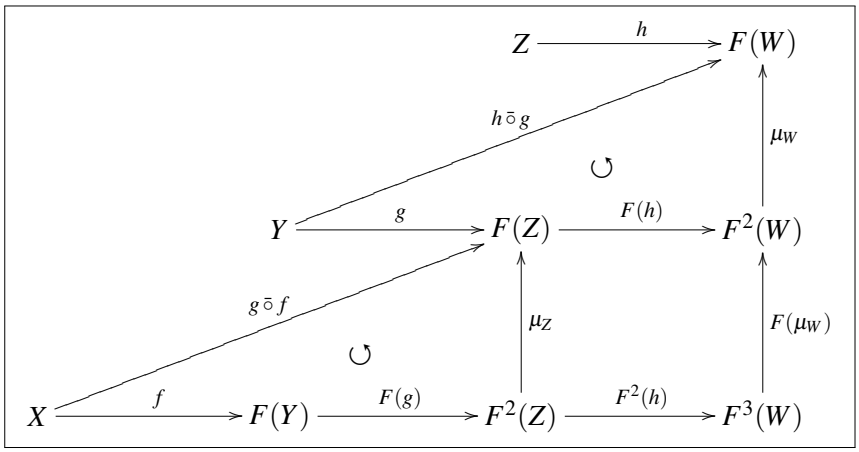
26.6.6 条件 (K5) が (M5) を導くこと

「いよいよ最後ですね」

「これだけはちょっと大変かもしれないね。条件 **(K5)** から条件 **(M5)** を導くところをやってみましょう。まず、もう一度条件 **(K5)** を書いてみると

$$\boxed{\forall \langle h,g\rangle, \langle g,f\rangle \in \mathrm{Comp}(\overline{A}) \qquad h \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} f) = (h \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} f.}$$

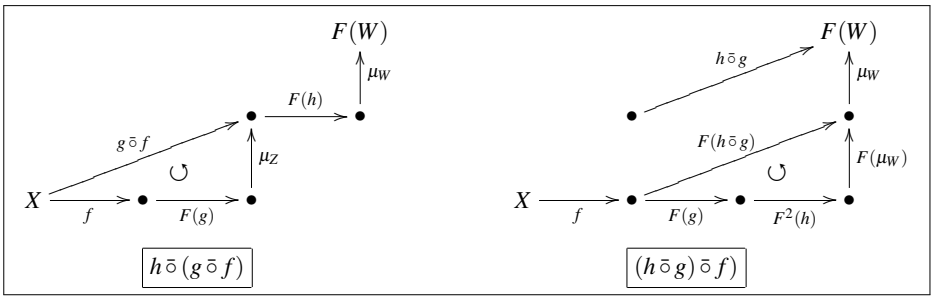
だけど、この条件は次の図式の中に埋め込まれてる」



「あ、今度のはちょっとでかい図式ですね」

「今書いた図式は少し複雑だけど、$h \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} f)$ と $(h \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} f$ の部分を取り出せばこんなふうになるよ」

そう言って私は次のような図を書いた：



「翼さん、黒丸が沢山ありますが――？」

「射が合成される経路に注意を向けやすくするように X と $F(W)$ の間の対象を全部 $\bullet$ に置き換えて書いたよ」

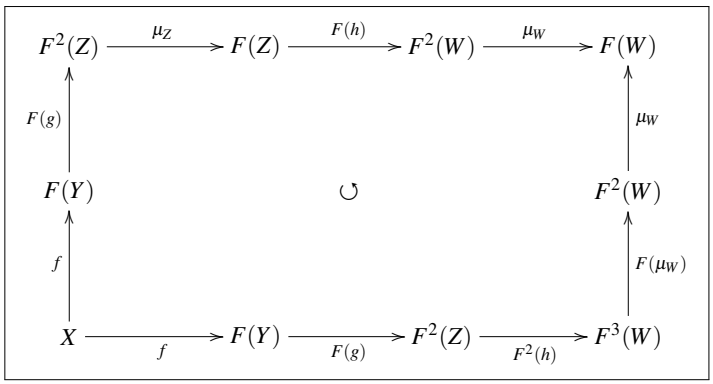
「なるほど、全体の図式はちゃんと見せてもらったし、確かに経路がわかりやすい気がします」

「ここで、条件 **(K5)** から言えるのは今の 図式全体の可換性ではない 事に注意しないとね」

「えーと、なるほど可換図式って言ったら、二通り以上の行き方は合成したら同じってのがどこでも成り立ってないといけないですもんね。それに対して、条件 **(K5)** から言えるのは、$h \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} f)$ と $(h \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} f$ が同じになるってことだけですね。でもすっかり可換図式体質になっちゃった私にはつらぽよですねこれは……」

「じゃあ上の図式から $h \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} f)$ と $(h \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} f$ の部分を拾ってきて展開して可換図式にしちゃおう」

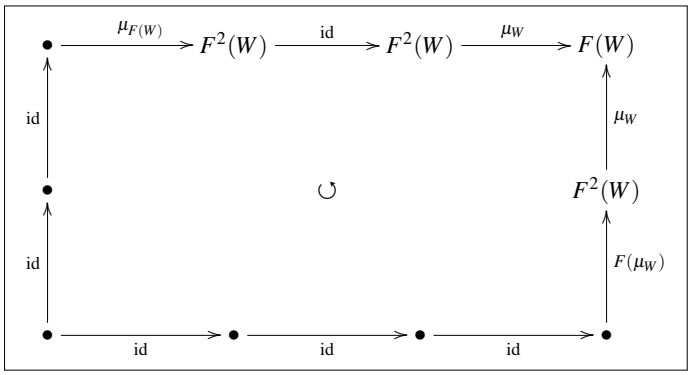
そう言って私は次のような可換図式を書いた。



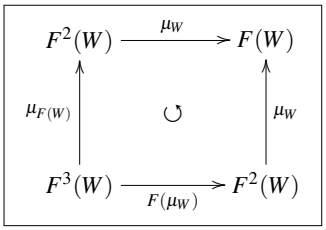
「翼さん、この続きやらせてください。

$$X := F^3(W),\ Y := F^2(W),\ Z := F(W),$$
$$f := \mathrm{id}_{F^3(W)},\ g := \mathrm{id}_{F^2(W)},\ h := \mathrm{id}_{F(W)}$$

としてやるんです！　大量に $F^3(W)$ が出てくるのでそこを全部 $\bullet$ で代用して、さっきみたいに自己関手の性質 (p.378) を使って整理するとこんなふうになります。あ、ついでというか、$\mathrm{id}_{F^3(W)}$ も全部 id って略すことにします」



「——id の部分をどんどん削るとこんなにかわいくなります」



「こうして出来た図式の可換性の条件を書いてみると任意の $W \in \mathrm{Ob}(\mathbf{C})$ について成立する次の関係式が得られるね：

$$\boxed{\forall W \in \mathrm{Ob}(\mathbf{C}) \quad \mu_W \circ F(\mu_W) = \mu_W \circ \mu_{F(W)}.}$$

よって、確かに条件 **(K5)** から条件 **(M5)** が導けたね」

「いやー終わってみると結構簡単でしたね。でも気になったんですが、こうやって特殊な場合に還元して条件 **(M1)**-**(M5)** を示してきたから、なんか条件 **(K1)**-**(K5)** を示すのが無理っぽい気がするんですが」

「逆向きの、条件 **(M1)**-**(M5)** から条件 **(K1)**-**(K5)** を示すほうは、自分でやってみるといいよ。私もさすがに疲れてきたな。ねえ翼ちゃん、良かったらお茶の時間にしない？」

26.6.7 Kleisli 圏まとめ

「さて、いままで調べたことを表にまとめてみましょう。関手の記号は『モナド』だから M にしたよ」

圏 $\mathbf{C}$ 上のモナド M が作る Kleisli 圏	圏 $\mathbf{C}$ における実体
対象 X	$X \in \mathrm{Ob}(\mathbf{C})$
射 $f\colon X \to Y$	$f\colon X \to M(Y)$
対象 X に対応する恒等射 id_X	$\eta_X\colon X \to M(X)$
$f\colon X \to Y$ と $g\colon Y \to Z$ の合成射 $g \circ f$	射 $\mu_Z \circ M_A(g) \circ f$
対象 X と Y が定める hom 集合 $\mathrm{Hom}(X, Y)$	$\mathrm{Hom}\big(X, M(Y)\big)$

表 26.4: 圏 $\mathbf{C}$ 上のモナド M が作る Kleisli 圏の構成要素とそれらの圏 $\mathbf{C}$ における実体

「なるほど、随分長い話になっちゃいましたけど、圏論のモナドと、『計算の抽象化』としての Haskell のモナドの関係もわかってきた感じがします。要するに、圏 **Hask** の関手 M から作った $X \to M(Y)$ の形の射が『合成』できることを要求するとモナドになるんですね」
こうして私たちは数学の話を終えて[*10]、しばし雑談に興じていた。そのとき——
地震が起きた。

26.7 カンザスでないどこか

26.7.1 竜巻に乗って

突き上げるような衝撃があり、グラリと家が揺れた。机の上のカップがわずかに跳ねたあと天板に落ちてカタッと音を立てた。残り少ない紅茶も波を打っている。

まだ揺れが残るなか、扇ちゃんがすっくと立ち上がった。目つきがすっかり変わっていた。
その顔に現れた強い緊張が徐々にこちらにも感染してゆく。
「扇——ちゃん？」
私の声は少し掠れていたと思う。そこにいたのは先程までの表情豊かな美少女ではなかった。
静かで力強いその瞳はあらぬ方を見つめており、小声で何事か呟いていたが、何を言っているのかは聞き取れなかった。
唐突に扇ちゃんは私に向き直り、
「わたしは、あれを止めに行きます——翼さんはここにいてください」
と言うと踵を返して部屋を出ていった。
「ねえ、ちょっと——」
止めるって何を——？ まさか地震ではないだろうし。地震にも驚いたが、それ以上に私は彼女の豹変ぶりにショックを受けていた。
バタリと玄関のドアが閉じる音がした。
私もあわてて——ここにいてください、と言われてはいたが——彼女を追って外に出た。

信じられない！ 墨を流したように暗い空から、雹混じりの暴風が叩きつけるように吹いている。
美しい朝焼けを見たのがわずか数時間前のはずだ——天気予報では何と言っていたっけ、と思い出そうとする。
それにしても、こんなに天気が悪くなっていたのにどうして今まで気づかなかったのだろう——？
降り注ぐ雹に視界を遮られているせいか、扇ちゃんの姿は見えなかった。
この突然の嵐も含めて何もかも不自然だ。

猛烈に寒かった。扇ちゃんを探すにしても一度家に戻って上に何かを羽織らないと——そう思って振り向

[*10] お疲れ様でした。本章での数学や Haskell の話はこれでおしまいです。

いたが——家はもうなかった。
それどころか見渡す限り何の人跡も見当たらない——。
広大無辺の平地に私は放り出されていた。
携帯電話はうっかり机の上に置いてきてしまった——しかしこの場所で電波が届くかどうかすら怪しいな、私はそう思った。
　それにしても寒い——強風が急激に体温を奪ってゆく。自然と歯がガチガチと鳴る。
パラパラとはじける雹で覆われた地面に足跡が見えないかと目を凝らしながら歩く。
人間が目印なしに歩く場合、真っ直ぐ歩いているつもりでもわずかに右か左にカーブしてしまうことが多い。このせいで、視界の効かない場所で遭難した場合同じ円の上を何度も歩いてしまい徒に体力を消耗することがあるのだという。——もちろんその行き着く先は死だ。
しかし、真っ直ぐ歩けたとしても、とにかくこの寒さから逃れないことには低体温症で死ぬことになるのも時間の問題だった。自分が一体どこにいるかも分からず、このまま死んでしまうのだろうか——
絶望感がこみ上げてくる。

　遠くから呼ばわる声——それとも悲鳴だろうか——が聞こえた気がした。
振り返ってみてぎょっとした。
竜巻——だろうか？　しかし、緑色の竜巻など聞いたことがない！
数十匹のうねり混じる大蛇のように不気味に動く雲の中から、薄い緑色を帯びた漏斗状の竜巻が象の鼻のように垂れていた。距離感がわからない——だが気のせいかその竜巻は次第にこちらに近づいているように思えた。逃げなくては——だが寒さで脚が思うように動かない。
錯覚ではなかった——振り返ると、本当に緑色の竜巻はこちらに近づいている！
確かにその竜巻は妙だった——色だけではない。近づいているにも関わらず土埃のようなものが伴っていないのだ。あの竜巻は物理的な実体を持たない幻影のようなものなのかもしれない——。

　ついに竜巻に追いつかれた。やはりこの竜巻には物理的実体はないらしい。これだけ接近してみると、この竜巻が緑色を帯びていた理由がよくわかる。この竜巻は群れをなして猛スピードで飛び回る、コインほどのサイズの緑の光点の群れで構成されていた。竜巻から逸れて飛来した光点は、私の身体に当たっても何の物理的衝撃も与えず、身体に吸い込まれてゆく。
周囲の風鳴りに混じって人の鳴き声や囁き声が聞こえてきた——それは竜巻からではなく、奇妙なことに私の耳元で聞こえてきていた。老婆の声、幼児の声、男の声、様々な声が私に向かって何かを言おうとしているようだった——しかし私にはどの言葉も聞き取れなかった。

　更に竜巻が接近し、大量の光点が私の身体に吸い込まれていった。周囲の景色が揺らいでゆき、私は意識を失った——。

26.7.2　黄色い (?) レンガの道

　気がつくと私は青い光に包まれた回廊のようなところを歩いていた——ここに来るまでに何があったのか全く思い出せない。
体力はすっかり回復していた。先程まで凍え死にしそうだった事が信じられない。
だが、本当にあれは先程のことだったのだろうか——。この場所に至るまでの記憶が途切れている事が気になった。以前にも記憶が途切れたことがあったが、そのときは私が怪異にまつわる事件に巻き込まれていた——というより私自身が怪異の力で騒動を起こしていた——のだった。
今度は一体何に巻き込まれているのだろう——そんなことを考えながら青い道を一人で歩き続けた。

　回廊、と私が思ったのは道の両側に等間隔で白い柱が並んでいたからだ。どの柱も上に行くにつれて樹木のように枝分かれを繰り返しており、はるか上方で細い枝が互いに絡みあって林冠を形成している。その更に上の方から、柔らかな青い光が降り注いでいた。
白い柱と柱の間の壁は、銀色の線で描かれた模様——雰囲気は唐草模様に似ているがもっと込み入ってい

る――で覆い尽くされていた。模様をなす銀線たちは緩慢な動作でからみ合ったりほどけたりを繰り返しており、壁のデザインは刻々と変わっていた。
歩いている道は透き通ったガラスのように透明でつややかな物質で出来ていた。上方から降り注ぐ光を受けて床も青みを帯びていた。
それにしても、ここは一体何処なのだろう？

気がつくと私はもう独りではなくなっていた。左にもう一人の「私」、そして右に堂々たる体躯の虎――。

左にいる「私」は肌も髪も白く、頭には猫のような耳が生えていた。下着姿だった。私はこの「私」を知っている――この姿は私の裏の人格が怪異の力を纏って現れたものだ。
初めて自分の眼で「私」の姿を間近に見た私は驚異の念に打たれていた。顔の造作や体型は私と大体同じはずなのに――元々は同一人物なのだ――これほどまでに印象が違うとは……。
歩くたびに揺れる前髪の間からのぞく金色の瞳は射抜くように力強かった。野性的で力強い目だ――目が合ったときに彼女が微笑み返してくる。この凄みのある笑顔も、普段鏡の中で見る私のものとはかけ離れている――なんだか落ち着かない。

それにしても――彼女と私は一つの身体をシェアしてる存在なので、私の横で彼女が歩いているという事態は本来起こり得ない。だが、その起こり得ないはずのことが起こっていた。
そんな疑問を察したかのように彼女はこう教えてくれた。
「ここは特殊な空間だから世界を縛るルールが少し緩いんにゃ」
「ふうん、なるほど――」
あれ、全然説明になってないような？
それにしても、彼女はなんと堂々と歩くのだろう――。
背筋を伸ばし全身をうねらせるようにして大股で歩く彼女の姿は肉感的な魅力を放っていた。
私はこんな姿で私は暴れ回っていたのか――そう考えると少し気恥ずかしかった。
「ブラック羽川さん――」
「さん付けするなんて水くさいにゃ」
歩きながら彼女が答えた。彼女は私が抱えきれなくなった心の暗部が切り離されて怪異と結びついて現出したものだ――それゆえに「ブラック羽川」と呼ばれていたのだった。
水くさい、か。
確かにその通りだった。彼女は私の背中から伸びる影、私が直面できないもう一人の私なのだから。
「そうだね――それに、そもそも『ブラック羽川』だなんてあまりにも他人行儀だよね――あなたは私なのに」
「ご主人はご主人、俺は俺にゃ」
「――ねえ、あなたを『クロちゃん』って呼んでいいかしら？」
彼女はちょっとびっくりしたようだったが「気に入ったにゃ」と照れくさそうに答えた。
いくらなんでも安易過ぎる名前だったかもしれないな――そう考えていると
「――では拙者の事は『シロちゃん』と呼んで頂けないだろうか」
右を歩いていた虎がうっそりと言った。あまりにも真面目な口調と不釣合いなので、私とブラック羽川さん――じゃなくてクロちゃん――は笑ってしまった。
「うん、じゃあよろしくね、『シロちゃん』――」
苛虎――私が以前そう名付けた存在に私は呼びかけた。
この虎もまた――絶大な炎の力を持つ――私の分身だった。
「ところで――いま私達はどこにいてどこに向かってるのかな？」
こうして口に出すと実に間抜けなのだが、切実な疑問だった。
「ご主人は聞いてなかったにょか――」

「恐らく、我ら怪異の部分が分離してしまったために主（あるじ）から記憶が欠落してしまっているのだろう」
「にゃるほど――。取り引きで、ご主人と私達はこれからハスケルスレイヤーと戦うことになったにゃ」

「取引って――？　一体誰と？」
「バーバヤガにゃ」
スラブ民話に出てくる超自然的な力を持つ妖婆――そう名乗る存在が夏休みに阿良々木くんがハスケルスレイヤーに襲撃されたときにも関与していた。
「我らはいま、Maybe モナドで創られた異空間に閉じ込められている」
「なるほどね――」
今更こんなことでは驚かなくなっていた。さっきクロちゃんが言っていた「特殊な空間」ってそういうことだったのか。
「元々はこの異空間に閉じ込められていたのはハスケルスレイヤーだけだったのだが、不可思議な力により主（あるじ）が、そして我等がここに呼び寄せられたのだ」
阿良々木くんの話によるとハスケルスレイヤーは Haskeller を殺してまわる忍者なのだという。
「勝手に話を進める形になってしまい恐縮だ――だが主よ、取引いかんに関わらずハスケルスレイヤーは危険な存在だ。看過はできない」
夏休みの事件でも、Haskeller と見ると問答無用で襲いかかってきたらしい。その理由というのも支離滅裂だったらしいし、確かに説得が通用するような相手では無さそうだ。だとするならば。
「――つまり、ハスケルスレイヤーを放置しておけば――また阿良々木くんが襲われるかもしれない。そして、バーバヤガが次も阿良々木くんを助けてくれる保障もない――」
「そういう事にゃ」
「ところで、さっき『取り引き』って言ってたと思うけど、私達への見返りは――？」
「ハスケルスレイヤーを倒せばご主人は元の空間へ帰れるにゃ」
「なるほどね――ええと、それじゃあなた達の取り分は――？」
「――俺はもう見返りを受け取ったにゃ」
「……拙者も既に受け取っている」
あれ、何を受け取ったか聞いては駄目なのかな？　そんな思考を読んだかのように――
「俺はもう一度ご主人に会って話がしたかったにゃん」
「拙者も、もう一度主と話がしたかった。前回は――いささか後味の悪い別れ方だったのでな」
そうだ――忘れていたわけではないが、クロちゃんもシロちゃんも私の心の不安定な暗部が怪異と結びついて現出した存在だ。したがって、私が自分の心を自分で面倒見られるようになった頃には自然に消滅してしまうはず――そういう存在なのだった。
クロちゃんにもシロちゃんにも私は自分の人生を大きく引っ掻き回された。でも、彼らは私の心が生み出した分身、私の一部なのだ。そんな彼ら、クロちゃんにもシロちゃんにも、会えるのはこれが最後なのかもしれない――。
「ごめんね――」
今までクロちゃんにもシロちゃんにも嫌なことばかり押し付けてしまって。そして――
「ありがとう、いままで一緒にいてくれて」

青い光の中、銀の猫の怪異たるもう一人の私と精悍な虎を従えて三者で歩いているうちに、私は再び『オズの魔法使い』を思い出していた。

かかしは脳みそが欲しい
ブリキの木樵はハートが欲しい
ライオンは勇気が欲しい
ドロシーは故郷に帰りたい

思わず呟く――
「本当にオズの国に紛れ込んでしまったみたいだね――」
「そうなると、この道が『黄色いレンガの道』ってことになるにゃ」
「――『黄色』というには少々青すぎるようだが」
シロちゃん――右を歩いている虎――が生真面目な口調で言った。

私とクロちゃんは顔を見合わせてクスクス笑った。
「にゃんにせよ、ここがもうカンザスじゃないことだけは確かにゃ」
私たちは――今度はシロちゃんも――声を上げて笑った。

回廊は途中から暗いトンネルにつながっており、私たちはその中を歩いていた。
目の前に扉があった。
旅の終わりだった。扉には

忍者を殺せ

と大きく書かれていた。

扉を見やり、クロちゃんに頷く――ぐらりと視界が歪み、つかの間私は彼女の眼で私の顔を見ていた。シロちゃんに頷く――夜行性肉食獣の視界で見ると色の感じ方や輪郭の見え方まで違うのだな――ほんの一瞬の経験の後、私は元の「私」に戻る。
扉の前に一人佇む私は、私のなかの存在たちに心で話しかける――どうか力を貸してください、と。
私の内側で何か答えようとする気配が感じられたが、融合は急速に始まっていた。
それに伴い、怪異の力が私の内側に急速に流れこむ。

クロちゃん――「ブラック羽川」の驚異的な運動能力。そしてシロちゃん――「苛虎」の操る圧倒的な炎の力。この二つの怪異の力が混ざりながら私の中で急速に膨らんでゆき、身体の隅々まで満ちてゆくのを感じた。身体の奥底からつきあがる巨大な力が全身を駆け巡っている。静電気を帯びたように皮膚がぴりぴりする。
「阿良々木くんのいる世界に帰るんだ」
高揚感を覚えながら私はそう呟き、戦いの舞台へと続く扉を押した。

26.8　もう一つの世界

26.8.1　◆1◆

重苦しい灰色の空の端を真紅に染める太陽は、まさに海に沈まんとしていた。
ハスケルスレイヤーには、その太陽がまるで惑星を飲み込むほどのサイズの巨獣の貪婪な眼のように見えた。空の凶眼を睨み返しながら、ハスケルスレイヤーはこの浜辺を馴染み深く感じている自分に気づき当惑していた。当惑を打ち払うかのように首を振り、クージ・チャントを唱える。平安時代の偉大なる剣聖詩人ミヤモト・マサシが考案したこのマントラには武人の精神を研ぎ澄ます効果があるとされているのだ。
＜扉＞はもう少しで開くはずだ。ハスケルスレイヤーには確信があった。彼は＜扉＞に一瞥をくれると、再びクージ・チャントを唱えた。夕日に照らされてメンポに刻まれた「蓮」「殺」の文字がギラリと光った。
周囲に人の暮らしていた痕跡はなかった。浜辺に面した岸壁を登ってみたことがあるが、そこから広がっているのは、ただひたすら続く荒野であった。黒っぽい溶岩質の土地は痩せており、拳ほどの高さの植物すらない有様だった。浜辺から少し沖合に向かうと、ゴツゴツした岩の並ぶ海底には巨大な貝類が生息していたため、食事には困らなかった。巨大な貝たちはそのサイズに見合った硬く頑丈な殻と蓋を持っていたが、ニンジャ筋肉が生み出す強力なカラテ・パンチの前には気休めにもならないのだ。
飲料水の確保には苦労していた。よく洗って日干しした貝殻に雨水を溜めていたが、雨頼みというのも心細かったため、貝殻を重ねた蒸留装置を試作していた。熱源は太陽光である。しかし、これとて効率は良くなかった。結局、生存に必要な水分の大半を貝の肉から得ている事になる。
浜辺には幾つか大岩が並んでおり、その中の一つの岩陰を彼は寝床にしていた。時折強くなる海辺の風を避けるために、石と貝殻を積み上げて申し訳程度の風よけにしていた。つまるところハスケルスレイヤーはこの浜辺での生活基盤をなんとか作り上げており、馴染み深さを覚えるのも当然なのだった。

26.8.2　◆2◆

この場所に不本意な到着をしてから数ヶ月が経っていた。ハスケルスレイヤーは、頭から烏賊のような青い触手を生やした怪物との死闘の末に、正体不明の妖術によって生み出された緑色の渦の直撃を受けたのだった。その不可思議な渦には物理的な実体がなかった。しかし、その渦が直撃したときに奇妙な感覚がこみ上げてきたのを覚えている。——まるで、自分を取り巻く世界との間の何かが切り離されてしまったかのように感じたのだ。

我を取り戻したときには、怪物は姿を消していた。怪物の姿を求めて街をさまよっているときに強い違和感を覚えたが、その違和感は容易に説明できた。街から人の姿が消えていたのだ。先程までいた場所は確かに人通りが少ない場所だったが、駅の近くまで来ても誰の姿も見かけなかった。

誰もいない。夏とはいえ、夕方が近づいていたため海水浴客の姿は少なくて当然だが、それにしてもまったく人の気配が感じられないのはおかしい。通りを歩き続け、観光客用の土産物店や雑貨屋、食堂、小さな薬局の並ぶ駅前通りに出た。どの店も営業中に見えたが、店員の姿はどこにも見当たらなかった。ハスケルスレイヤーは通りにある店を調べてみることにした。

「実際旨い！　シーフード・ラーメン」と書かれたのぼり旗を出している店の奥からはうっすらと湯気が立ち上っており、近くに行くと焦げた食用油とソイソースとタバコの煙が混じったような匂いが漂ってきた。ノーレン・カーテンをかき分けて店に入る。小さな店だ。一ダースほどのテーブルしかない。店内は無人だったが、幾つかのテーブルにはラーメン・ボウルが載っていた。ボウルの中のヌードルスープはまだ温かった。木のテーブルの上には灰皿が置かれていた。その少し外に火の着いたタバコが落ちており、テーブルの表面をゆっくりと焦がしている。ゆっくりと首を振る扇風機が生み出す気流がタバコの煙をかき乱していた。従業員用の通路を通って厨房へ向かった。大型の鍋が火にかけられており、ヌードルが茹でられていたが、ここにも人の姿はなかった。

その店を出たあとも色々な建物を調べたが、彼だけを例外として街にいたすべての人間が、謎めいた方法で忽然と姿を消したのだという確信は強まるばかりだった。

ハスケルスレイヤーは駅の近くの奇妙なビルの前で足を止めた。正確には、奇妙なのはビルそのものではなくその有り様だった。夕暮れの陽光に照らされて赤く輝く雲はビルのグレーのシルエットで切り取られていたが、ビルの一部を通して雲の輝きが微かに漏れているのに気づいたのだ。見間違いかと思い凝視したが、確かに光はビルを通して漏れていた。そうして見つめているうちにも、ビルの透明度が次第に増していた。

いまやビル全体がガラスで出来ているかのように透けている。赤く輝く雲の輝きは、もはや遮るものなど何もなかったかのようにはっきりと見えた。ビルの輪郭はいよいよ薄れてゆき、ついにはビルそのものが消えていた。我に返って辺りを見渡すと自分が立っている場所の周囲の建物が姿を消していた。そもそも街などなかったかのような開けた平地だけが残っていた。周囲の樹木も姿を消しており、風景は非常に単調なものに変貌していた。今や彼の周囲に広がるのは荒涼たる大地のみであった。

何が起きているのかは相変わらず理解できなかった。とにかく面白い事態ではなかった。怪物との戦闘の際の怪我からはひとまず回復していたが大量の血液を失ったため、疲労もたまっていた。休息を取らねばならない。周囲に敵の気配がないことを——敵はおろか実際にはいかなる生き物の気配もなかったのだが——もう一度確認してから大地の真ん中にゴロリと横たわると、次の瞬間には眠りに落ちていた。

夜中に目覚めたハスケルスレイヤーは満天の星空の下にいた。うっすらと白くけぶる天の川銀河に見入っていると「星が好きなのかい、アッヒヒヒ！」と話しかける声が近くで聞こえた。

その耳障りな笑い声の主はボロ布を重ね着した巨大な妖婆だった。その肩には大ガラスが止まっている。

「あたしはバーバヤガってんだ。迷子の面倒を見るのがあたしの仕事さ」

枯れ枝のような手からしなびた指を突き出し、妖婆はそう言った。迷子、というのはどうやらハスケルスレイヤーの事を指しているらしい。

「俺はハスケルスレイヤー。全ての Haskeller を殺すニンジャだ」

「アッヒヒヒ！　何のために殺すんだい？」

「それは——」

そう言いかけて突然ハスケルスレイヤーは不機嫌に黙り込んだ。Haskell の普及を下地としてもたらされる技術的特異点がやがては宇宙そのものを破滅させるに至るということを他人に理解させることが非常に難しいことを彼は経験から学んでいた。

「言いたくないかい、アッヒヒ！。まあいいさ、本題はそれじゃない。本題は、あンたがこの先どうするかってことさ」

「それよりこの場所は一体どこなのだ？」

「そうそう、それも大事さね、複雑な事情だから順番に説明しないといけないンだけどあたしは順序立てて話すのが苦手でね――。まずこの場所だけど、ハスケルスレイヤー=サン、あンたは Maybe モナドが作り出した空間に閉じ込められてる」

「生憎だが、こちらは冗談に付き合う余裕はない」

「冗談とね、アッヒヒ！ 確かに冗談みたいな話さね、でもこれは冗談じゃないンだよ！ あンたが今いるこの世界はイカ娘がコトダマ空間の操作をした結果生み出されたのさ」

荒唐無稽な話に思えたが、改めて自分が体験した奇妙な出来事と照らし合わせてみると不思議な説得力を持っていた。彼がいま置かれているこの巨大な空間全体が一瞬にして生成されたという考えは、技術的特異点の発生によって可能になった野放図な物理法則の書き換えが引き起こした大災害を想起させ、ハスケルスレイヤーを不安に駆り立てた。

「空間を生成する、などという事はたやすいことではないはずだ」

「正確には、この空間はゼロから作られたンじゃなく、オルドヴィス紀から分岐してつくられた『あったかもしれない現在』なのさ」

「つまりあの怪物―イカ娘と言ったか―が過去の地球から分岐させてこの時空を作り出したのか」

「それも正確じゃないね。あいや、あたしの説明がまずかったンだね。そうさね、確かにこれは込み入ってる。すごく込み入ってるんだよ。そう、ここは確かにオルドヴィス紀から分岐した時空だし、そのトリガーを引いたのはイカ娘だけど、この場所を選んだのはあンたなのさ、ハスケルスレイヤー=サン。イカ娘が作り出したあの渦は心を読み取って動作する。あれは、精神の増幅器の一種なんだ。自分を取り巻く世界を拒んでるあンたみたいな人間の精神を核にして、別の世界を作り出してしまう」

「俺の精神――？」

「わからないのかい？ それともわかりたくないのかい？ つまりね、ここはある意味であンたの願った世界の姿なのさ」

「ばかばかしい」

「あンたがやろうとしてることは、突き詰めればプログラミングをする知性体が存在しない世界でしか実現しないんだろうね」

それには取り合わず、ハスケルスレイヤーはバーバヤガに尋ねた。

「そんな話はどうでもいい。いま知りたいのはここを出る方法だ」

「一番簡単なのは、あたしについてくることさ。あたしはあンたの在り方をもっと高い次元に引き上げて、あたしみたいにここを自由に出入り出来るようにできる――まあこの空間は不安定だから、あたしたちが引き払ったあとは存在の根拠を失って勝手に消えるけどね」

「なるほど、うまい話のようだがバーバヤガ=サン、お前は見返りに何を要求するのだ？」

警戒心を隠そうともせずハスケルスレイヤーは聞いた。

「見返りってほどのものは何もないさ。ただし、さっきも言ったようにこの方法はあンたの存在の次元を引き上げることでなされるンだから、あたしについてくる場合はその肉体もここで捨てることになる。当然、元の空間には帰れないけどね」

「論外だ。俺の使命は Haskeller を皆殺しにすることだ。使命を全うするためには元の世界に帰らねばならない」

「あンたは肉体に固執してるから、あたしの申し出の価値がわからないンだね」

そんなバーバヤガの嘆息を無視してハスケルスレイヤーが言う。

「他の選択肢について教えてくれ」

「本当に強情な子だね、あンたは。まったく、ニンジャってのはどうしてこう強情なのが多いンだろうね――」

やれやれ、と大げさな溜息をつき、渋々といった調子で妖婆は話し続けた。

「――いいかい、イカ娘が作り出したあの奇妙な渦はあンたにこの並行世界を作らせた。そこまではあンたもわかったね？　でもあの渦はそれじゃ止まらなかったンだよ。あの渦はエネルギーが大きすぎたせいで分裂してこの空間から漏れだしたのさ。そして、世界の隙間をずうっと漂い続け、やがてあの渦は別の凶暴な精神を核にしてもう一つの平行世界を作らせる。元々は一つの渦から出来たンだから、もう一つの平行世界はこの平行世界と融合してしまう。だけどね、あンたを中心にして作られたこの並行世界と別の誰かを中心にして作られた平行世界は共存できない。だから二人が戦って決着をつけないといけないのさ。一方がもう一方を殺せば殺した相手の全てのエネルギーを受け取れる。宇宙一個分の膨大なエネルギーだよ！　これを使えばこの空間そのものをねじ曲げて、元の世界に帰れる」

バーバヤガの饒舌はハスケルスレイヤーにとってほとんど意味をなさなかったが、要点だけははっきりと汲み取れた。

「つまり、あの渦にまた別の誰かが巻き込まれる。その誰かと俺が戦う。俺が相手を殺せば元の世界に戻れる――ということか」

「そういうことになるね」

「バーバヤガ=サン、この空間に自由に出入り出来るほどのジツを持つお前なら俺をいまここで殺すことなど造作も無いことではないのか？」

ハスケルスレイヤーが挑発的な問いを投げかけた。

「なんであたしがあンたを殺さにゃならないんだい？　――あンたは、あたしを誤解してる。うん、純粋に能力という意味で言えば、あたしが何とかできないニンジャなんてのはほとんどいないね。そういう意味から言えば、あンたは正しい。やろうと思えば、あたしはあンたを殺すことができる。あたしの管理者としての責任から言えば、そうするのが正しいのかもしれないね。だけどね、あたしはニンジャと近しい存在なんだ。それはもう、あンたなんかには窺い知れないほど深い縁があるのさ」

「それは――ひょっとして『カラテノミコン』に関係する話か」

考え深げにハスケルスレイヤーが言った。

「察しがいいね。なるほど――あンたが少し変わってる理由がすこし飲み込めてきたよ。あンたはあの忌まわしい書物と関わりを持ってしまったんだね。あンたがどの程度あの本を理解しているかは知らないし、あの本に書かれてる事がすべて真実というわけじゃないけれど、そうだね。あたしの事は『カラテノミコン』にも書かれている」

「全てのニンジャを守護する七人の巨人――そのうちの一人がお前か」

「アッヒヒヒ！　そういうことになるかね。まあそこまでわかってるなら説明も省けるね。あたしはニンジャの守護者なんだ。どんな理由があってもニンジャを自分の手にはかけられない」

「俺を間接的に殺すのは構わない、というわけか」

「あンたがどう思うのも勝手さ。さっきも言ったように、あンたがあたしについてきてくれればあたしはこの空間を安全に解消できるンだし、あたしとしてはそれが一番望ましいと思ってる。それが嫌だってんなら、あンたがもうじき現れる誰かを殺せば元の世界に帰るチャンスだってある。でも、そっちはあたしの好みの結末じゃない。だからあの渦に少しばかり干渉して、あたしの望む結末が得られるように細工をするなんてことはあるかもしれないね」

「バーバヤガ=サン、結局のところお前は俺の敵だ。だが、俺に宿るニンジャソウルに免じて情けをかけてくれようとしたというのも、また事実なのだろう。先ほどの無礼な物言いについては謝る」

「あたしとしちゃ、謝られるよりあたしについてきてくれた方が嬉しいよ。しつこいようだけど、本当にあたしについてくるつもりはないのかい？　あたしについてくればあンたはもっと偉大な存在になれるんだよ！」

「申し出は有難いが、わたしもまた元の世界に対して責任があるのだ。次こそは技術的特異点の発生を阻害しなければならない」

バーバヤガの肩にとまっていた大ガラスが嗄れた声でコー、とひと鳴きした。バーバヤガは大ガラスをそっと撫でると悲しげに首を振った。すると、バーバヤガと大ガラスの姿がぐんぐんと大きく希薄になってゆき、ついには消滅した。

こうして、ハスケルスレイヤーはこの不毛な並行世界にただ一人残されることになった。

26.8.3 ◆3◆

ハスケルスレイヤーがその＜扉＞を見つけたのは今朝だった。いつものように起きた彼は、空気がいつもと違っているのに気づいた。いや、空気ではなく音だ——岩に打ち寄せた波の残響が普段と微かに違うのだ。鋭敏なニンジャ聴覚は、前夜まで存在しなかった何かが浜辺に出現している事を告げていた。

それは鈍い金属光沢を持つ青い板だった。
人工物としか思えない鋭い稜線を持つその物体は、久しく人界から隔離されているハスケルスレイヤーの胸に文明への懐旧の念を掻き立てた。厚さ三インチほどのその板は何の支えもなしに砂浜から一フィートほど浮かんでいた。触ると微かにひんやりとする。板には線刻が施され、乙女と虎と魔物が描かれている。
乙女は巨大な虎を従えて歩いていた。虎の巨躯の輪郭線の闊達なうねりは、どこか炎を思わせなくもない。顔まで毛に覆われた魔物が乙女の横に付き添っていた。乙女の高貴な表情と魔物の残忍そうな表情が対比をなしている。よく見ると魔物の脚は地面に近づくにつれて解剖学的な秩序を壊しながら湾曲し、乙女の足元から伸びる影と融合していた。線刻の図像はタロットカードを想起させた。

そうだ——この絵はタロットの「力」のカードに似ている。どんな困難な運命をも従わせる絶対的な力と精神を象徴するカードだ。蝶番も取っ手もついていなかったが、ではこれこそが我が運命を切り開く＜扉＞に違いない。もちろん、カードの暗示には両義性がある。したがって、この＜扉＞が意味するのは、あるいはハスケルスレイヤーに襲いかかる恐るべき強敵を意味するものなのかもしれなかった。
だが、相手がどんなに強大だろうと——ハスケルスレイヤーは心に誓った——必ずや相手を殺してみせる。

26.9 死闘

太陽は沈み、空は急速に暗く染まってゆく。
風が吹いた。＜扉＞は音もなく消滅し、銀髪の少女が立っていた。
頭から猫のような耳が突き出していた。リボンだろうか。少女の均整のとれた優美なシルエットもネコ科の動物のそれを思わせた。
なるほど——あの少女が殺すべき相手であり、運命の＜扉＞でもあるということか。いいだろう、扉をノックし、運命を切り開いてやろう——少々手荒なノックになりそうだがな。

砂浜を歩き出した姿勢がまったく左右に揺れていないことから、少女が発達した運動神経と卓越した筋肉の持ち主であることが伺える。彼女の身体に漲る野生の気配は、彼女が容易ならざる敵手であることを告げていた。風に揺れる少女の銀髪の間から金色の瞳がハスケルスレイヤーを鋭く睨み据えた。彼女のバストは豊満であった。
少女が決断的にアイサツした。
「ドーモ、ハスケルスレイヤー=サン、ハネカワ・ツバサです」
ハスケルスレイヤーもオジギを返す。
「ドーモ、ハネカワ・ツバサ=サン、ハスケルスレイヤーです」
ハスケルスレイヤーはオジギを終えた 0.02 秒後に身体を丸めて前転し始めた。そして前転を繰り返しながらクナイ・ダートを連続投擲！
言うまでもなく、不安定な姿勢で運動しながらの精密な投擲はタツジンならではの高度なワザマエである。連続して投擲された合計 16 本のクナイ・ダートは猛スピードでハネカワ・ツバサに迫るかと思えたが——しかし！
クナイ・ダートはいずれも虚しく空を切り浜辺に突き刺さった！
一体何が起きたのかと訝しむハスケルスレイヤーの後方から声がした。
「コヨミから聞いた話ではもう少し手強そうな印象だったんだけどにゃ」
ハスケルスレイヤーが慌てて後ろに向き直ろうとした瞬間、右側から猛烈な勢いで蹴りが叩きこまれた。反射的に右の前膊でガードしたが、蹴りの衝撃でハスケルスレイヤーの身体が宙を舞った。無様に砂浜に落下したハスケルスレイヤーが立ち上がろうとした瞬間、腹を下から強烈に蹴り上げられた。ハスケルスレイ

ヤーは身体を丸めたままボールのように転がり、そのまま寝床にしていた大岩にぶつかった。ガシャーン！あまりの衝撃に大岩が割れた。崩れた大岩の欠片がガラガラと落ちてくる。その下敷きになってもがいているといきなり首根っこをむんずと掴まれた。岩の隙間から強引に引き出されたハスケルスレイヤーはそのまま宙に投げられた。猛烈なスピードで顔から岸壁に叩きつけられたハスケルスレイヤーは力なく崩れ落ち、貝殻で作った蒸留装置の上に落下した。ガシャーン！

「ふーん、人間と違ってすごく丈夫だにゃ。でもさすがに首をもぎ取ったら死ぬんじゃないかにゃ？」

ウケミ・ジツを使って蹴りの衝撃を全身に分散させていたにもかかわらず、ハスケルスレイヤーの右腕は無残に折れてあらぬ方向に曲がっており、腹を蹴られた衝撃で折れた肋骨が肝臓に突き刺さっていた。鋼鉄製のメンポは岩に激突したせいで歪んでいた。

「――それでも死なないなら肉片になるまで岩で叩き潰してみればいいにゃ」

ハネカワ・ツバサは瀕死のハスケルスレイヤーにゆっくりと歩み寄りながら、こともなげに言った。

イモータルニンジャ計画の流れを汲む冒涜的な研究によって作り出されたハスケルスレイヤーの身体は、肉体が消し炭になるレベルの損傷を受けたとしても時間が経てば自動的に再生する。しかしながら、この平行宇宙もろとも消滅させられてしまえば、さすがに再生するのは不可能であろう。こんなところで絶対的な死を迎えるというのなら、俺の今までの戦いはなんだったというのか。

こんな結末を迎えるために、俺は何人もの Haskeller を殺めてきたというのか。

全ては無意味だというのか。

――断じて――否！　力だ――！　力が欲しい――ハネカワ・ツバサを殺す力が！

クージ・チャントを繰り返し唱えていたハスケルスレイヤーの脳に分泌された天然のザゼン成分は一種の変性意識状態を作り出していた。力への渇望はニューロンの奥底まで鋭いスパイクとなって響き、ついには精神の奥底に宿るニンジャソウルを揺さぶり起こした。

ニューロンの叢の深層で微かな波動が沸き起こる――ハスケルスレイヤーには、それは彼方から呼ばわる静かで深い暗赤色の＜声＞として感じられた。その＜声＞に意識を向けた瞬間、脳中の一点から大量のカラテ粒子が湧きだした。脳から溢れだしたカラテ粒子の奔流は肉体の隅々まで流れこみ、身体中の細胞と結合して賦活化してゆく。カラテ粒子によって賦活化された血球は全身の骨折箇所、ねじれて破損した関節包、断裂した筋肉を瞬く間に修復し強化してゆく。岩に打ち付けられて壊れていたメンポもまた、カラテ粒子の奔流がもたらす超自然的な力によって再構成されてゆく。

轟音が聞こえた。耳を塞ごうとしたハスケルスレイヤーは、その轟音が自分の喉からほとばしる絶叫であることに気づいた。制御不能な力に対する本能的な恐怖がハスケルスレイヤーを無意識に絶叫させていたのである。

だめだ――。カラテだ……カラテに意識を集中するのだ――。

膨大な力に対して、ハスケルスレイヤーはウケミ・ジツを使い始めた。エネルギーの種類を変えながら分散させて逃がすのだ。身体から火花が飛び散る。ウケミにより余分なカラテ粒子が電荷に変換されているのだ。歯を食いしばりながら、ハスケルスレイヤーは身体から飛び出す電荷の奔流が渦となって身体を巡る様子を思い浮かべた。すると、身体を軸として大量の電荷がフラフープめいて渦を巻きながら流れだした。

ハスケルスレイヤーがなおも電流を増大させながら立ち上がり、クナイ・ダートを連投したとき、すでにハネカワ・ツバサの身体は強力な磁場に捉えられていた。連投されるクナイ・ダートを高速移動で回避したハネカワ・ツバサの身体がぐらりと傾き、倒れた。超強力な磁場の下で高速に移動したため、全身のいたるところで巨大な渦電流が発生し、ジュール熱によって深層筋に達する程の火傷を負っていた。身体を冷やそうとしてか、彼女は這いずりながら波打ち際に移動し、うつ伏せに倒れて動かなくなった。だが――彼女が「死んで……たまるか」と呟いたのと同時に彼女の身体に異変が起こり始めたのをハスケルスレイヤーは見逃していた。ウカツ！

ハスケルスレイヤーの身体を貫く巨大な電流は身体の内側を焼き焦がそうとしていたが、尽きることなく湧き出るカラテ粒子により、ただちに修復されていた。身体を流れる電流が作り出した超強力な磁場により、彼の身体は鉄分を含んだ岩にガッチリと縛り付けられていた。ハネカワ・ツバサは倒れ伏したまま動いていない。そろそろ止めを刺す頃合いだ。彼の身体を自由にするためには電磁カラテを中断する必要があった。超強力な電磁コイルとなった彼の身体を流れる電流は瞬時には停止できない。ハスケルスレイヤーはゆっくりと電磁カラテの威力を下げ始めた。

波打ち際で瀕死の状態で横たわっていると思われたハネカワ・ツバサの身体は怪異の力で完全に修復されていた。風が強まった。原因を作っていたのはハネカワ・ツバサの全身から放たれる膨大な熱であった。如何なる作用に守られてか、彼女自身は熱の影響を全く受けていなかった。激しい輻射熱により周囲の海が沸騰し、岩が赤熱し始めていた。猛烈な上昇気流が発生していた。湿潤な空気塊が上空に昇りながら膨張し急激に冷却され、熱を奪われた水分は氷のダストを形成してゆく。彼女の頭上はるか上空では驚くべき勢いで積乱雲が形成されつつあった。急激に形成された巨大な低気圧は、地上で突風を巻き起こしていた。

猛火を衣として身にまといながら、ハネカワ・ツバサがゆっくりと立ち上がった。

自らの身体を取り巻く電流のグロー放電で視界の透明度が下がっていたハスケルスレイヤーは、ハネカワ・ツバサの周囲に起こりつつある変化に気づいていなかった。「非常に明るいボンボリの真ん前はかえって見にくい」というコトワザ通りの事態であった。

コツ、と何かが降ってきた。小石ほどのサイズの氷塊である。雹が降り出していた。先程から吹き付ける強風に混じってバラバラと降り始めた雹は激しさを増し、今や辺り一面に雹が降り注いでいる。激しくなる雹には拳ほどの氷塊が入り混じっている。微かに帯電している雹が磁力線に巻きつきながら飛来しているため、ドカドカとハスケルスレイヤーの頭を打ち据えた。ハスケルスレイヤーが飛来する雹に気を取られている間に、ハネカワ・ツバサが至近距離まで接近していた。アブナイ！
炎はプラズマの一種であり、導電性を持つ。精密な炎のコントロールにより、ハネカワ・ツバサは周囲に幾重にも層をなす導電性シールドを展開していた。このシールドのせいで、彼女は強磁場ダメージをほとんど受けなくなっていた。ようやく彼女に気づいたハスケルスレイヤーがクナイ・ダート投擲の動作に入るより、彼女の白熱する双掌がハスケルスレイヤーに触れるほうが早かった。

ハネカワ・ツバサの両手から注がれる熱はたちまちハスケルスレイヤーの身体の全身の血液を沸騰させた。攻撃を払いのけようと彼女の手首を掴んだハスケルスレイヤーの前腕は瞬時に爆ぜて消滅し、肘から先に切り株だけが残った。ハスケルスレイヤーはヤバレカバレになり、最後の闘志を振り絞って身体を流れるリング状の電流を再加速した。

ハネカワ・ツバサが口を開いた。
「ハスケルスレイヤー=サン、あなたの身体はこれだけの強磁場を発生させても、これだけの熱を注いでもまだ爆発しないのね。面白いわ。何万度まで耐えられるか試してあげる」
ハネカワ・ツバサの全身を包む炎の温度がさらに上がった。彼女がさらに熱を注ぎ込むとハスケルスレイヤーの内蔵はすべて焼きつくされ、プラズマに変わった。ハスケルスレイヤーの身体を爆裂から守っていたのは全身を駆け巡るカラテ粒子だったが、いまや膨大な熱が生み出す巨大な圧力に押されつつあった。ハスケルスレイヤーの身体に亀裂が入った。耐え切れなくなった身体はついに崩壊し、彼の身体は爆発四散した。これと同時に、ハスケルスレイヤーとハネカワ・ツバサを取り囲んでいた平行宇宙は巻き取られた。虚空の中に取り残されたハネカワ・ツバサは折りたたまれた平行宇宙の膨大なエネルギーを利用し、その願いを叶えた。

26.10　エピローグ

こうして私は元の世界に帰ってきた。怪異の力はどうやら完全に沈黙してしまっているらしく、身体も含めて私は元の羽川翼、ただの女の子に戻っていた。もっと普通の方法が良かったのだが、爆発する大火球の中から轟音と共に放り出されるという派手な方法で帰還——阿良々木くんがいる世界に——を果たした私は、早速いくつかの問題を抱えることになった。

第一に、私が放り出されたのは海面から 100M ぐらいの上空だった。まあでも運が良ければ骨一つ折らずに着水できるかもしれない。第二に、直前にいた世界で少々派手に暴れていたお陰で服が全て焼失していた。公然わいせつ罪で補導歴なんてちょっと困る。

水面が近づいてきた。先程までの身体能力を失った私はそのまま無様に水面に叩きつけられそうだったが、いきなり——ぐにゃりとしたものが何本も現れて私の身体に巻き付いてゆく。その青いヒモのようなものはしっかりとと私の身体を受け止めており、自由落下状態にあった私の身体は次第に減速され、ふんわりと海面に近づいた。

「もう大丈夫でゲソ！」

と元気よく声をかけてきた青いヒモの持ち主は、白い水着を着たほっそりとした少女だった。何本もの青いヒモは帽子の下から出ている。そうか、これはヒモじゃなくて触手なのか。初対面だったが、その姿と口調には心当たりがあった。

「助けてくれてありがとう——イカ娘——さん？」

触手にぐるぐる巻きにされて抱きかかえられながらお辞儀しようとしたがうまくいかなかった。

きょとんとしているイカ娘さんに自己紹介と事情説明を簡単に済ませた。

ハスケルスレイヤーの最期についてのくだりを聞きながら、イカ娘さんは「バーバヤガは余計なことをしすぎでゲソ」と漏らした。

モーターボートが近づいてきた。イカ娘さんはそれにちらりと目をやると「面倒だからとりあえず逃げなイカ？」と言った。そして

「さあ——行くでゲソ！」

という元気な掛け声とともに私を抱きかかえたまま海面を走りはじめた。モーターボートは猛スピードでこちらを追ってた。拡声器を持った金髪の女性がボートの中で仁王立ちになり「イカ娘さん！　その宇宙人を渡しなさい！」と叫んでいる。その後ろで「シンディーさん、ちゃんと操縦してください！」と絶叫してる白衣の男たちの声が聞こえた。

「宇宙人って……わたし……？」

「いきなり空が爆発して翼が出現したから、爆発した宇宙船から脱出した宇宙人だと思われてるんじゃなイカ？」

イカ娘さんがそう答えた。怪異になって忍者と戦ったあとは宇宙人と思われて逃げている。本当に忙しい日だった。

変な人たちの追跡を逃れた後、イカ娘さんが居候している相沢家の親切な皆さんのおかげで無事に帰宅できた。帰宅して自室に戻ると、携帯にメールが入っていた。扇ちゃんからだった。「翼さん、忍者との戦いお疲れ様です。私は巻き込ませたくなくて邪魔しに行ったんですが、バーバヤガに一歩先を越されちゃいました。あと、今日は圏論とかおしえてくれてありがとうございました」と書いてあった。私がどこで何をしていたのか、だいたい把握してる様子だった。本当に謎の多い子だ……

こうして、あらためて自分の部屋のベッドにごろりと横たわった私は『The Wizard of Oz』のドロシーのセリフ

「お家が一番だわ」

を口に出してみた。なぜか涙がぽろぽろとこぼれだした。
ひとしきり泣いてみたら不思議と心が楽になった。それから私はたっぷり眠った。

参考文献

本記事の執筆にあたり、以下の諸作品からキャラクターや設定を拝借いたしました。素晴らしい作品を発表されている原作者様各位に敬意を表明いたします。勝手なキャラクターや設定の拝借および改変については何卒ご寛恕下さいますよう願い申し上げます。

26.10.1 ネタ元

- 安部真弘「侵略！ イカ娘」
- 西尾維新〈物語〉シリーズ
- ブラッドレー・ボンド、フィリップ・ニンジャ・モーゼズ「ニンジャスレイヤー」
  http://d.hatena.ne.jp/NinjaHeads/

また、執筆にあたって以下の文献を参考にいたしました。

26.10.2 参考にした教科書および専門書

[1] 結城 浩「数学ガール」シリーズ ソフトバンク・クリエイティブ
本記事の直接のネタ元というわけではありませんが、主人公「僕」が周囲の人間を巻き込みあるいは巻き込まれながら、対話を交えた試行錯誤のなかで数学を学んでいくというスタイルには多大な影響を受けています。

[2] 松村 英之「集合論入門」 朝倉書店 (1966)
本記事が前提としている数学独特の言い回しや記号遣いに不慣れの場合は、「日常語としての集合論」についての教科書に目を通しておくことをおすすめします。そのような教科書は数限りなくありますが、個人的な好みからこの松村先生の本を推薦いたします。数学の諸分野で日常語として使われる集合論が、圏論への誘導を意識して説明されています。

[3] 前原昭二「記号論理入門」 日本評論社 (1967)
込み入った事柄を整理し、論理的な構造を際立たせる表記法が数学ではよく使われます。このような表記によって色々な命題を表現し、解析する学問は「記号論理学」と呼ばれています。この本は記号論理学の初歩的な話題を初心者向けにゆったりと説明した本です。大学生向けの教科書で「入門」とタイトルにあるものには初心者にはまったく向かないものも多いのですが、この本は本当に初心者向けに書かれていると思います。

[4] Miran Lipovača 「すごい Haskell 楽しく学ぼう！」, 田中 英行・村主 崇行 共訳, オーム社 (2012/5)
プログラミング経験者向けの Haskell 入門書として評判の高い
"Learn You a Haskell for Great Good!: A Beginner's Guide", No Starch Press (2011/4)
の日本語訳です。とても読みやすく訳されていると思います。なお、英語原文は Web で全文を読むこともできます：http://learnyouahaskell.com/

[5] S. Mac Lane, "Categories for the Working Mathematician", Springer-Verlag (1971)(2nd ed : 1998)
よく読まれている圏論の教科書です。何の説明もなく「CWM」と言う略称で呼ばれることもあります。すでに何らかの分野における数学の専門知識を持っている読者を想定読者としているため難しいところも多いのですが、圏論について真剣な興味があるなら持っていて損はありません。本記事における Kleisli 圏の扱いは、この本における「モナドの Kleisli 圏」の項目を参考にしています。ただし、この本での Kleisli 圏の記述は非常にあっさりとしており、証明も概略が与えられているにとどまります。

[6] S. Mac Lane 「圏論の基礎」 三好 博之・高木 理 共訳, シュプリンガー・ジャパン (2005), 丸善
CWM 第二版 (1998) の日本語訳です。

[7] —— 「Haskell/圏論」
http://ja.wikibooks.org/wiki/Haskell/圏論

Haskell と圏論の関係について基本的な事が書かれています。モナドについても解説されています。本記事における条件 **(M1)**-**(M5)** は、このページの unit と join によるモナド則をお借りしたものです。

[8] Eugenio Moggi “Computational lambda-calculus and monads” (1988)
計算の圏論的意味論がモナドの理論に基いて述べられています。副作用モナドの話はこの論文の Example 2.6 を参考に書きました。

第27章

λ力娘探索 ？

— @ark_golgo

27.1 プロローグ

図 27.1: λカ娘三連敗 ！

「あううう、また負けたでゲソ」
「これで儂の 3 連勝。やはり置石が足らんかな」
「そ、そんなことないでゲソ。4 子も置いて負けたのはたまたまでゲソ! 次は勝つでゲソ!!」
「まあその意気込みだけは買ってやるがな」
λカ娘は最近ある老人と囲碁を打つようになった (図 27.1)。囲碁に自信があったλカ娘であるが、この老人には 4 子のハンデ (段級位にして 4 段級差に相当) を付けてもらってもまるで勝てなかった。
「あううう、ああは言ったものの、そう簡単に勝てる相手ではないことは良くわかるでゲソ。囲碁はそう簡単に強くなれるゲームでは無いでゲソ。私も囲碁歴は 10 年近くて、アマ三段はあると思うでゲソが……。あの老人は何者でゲソか……。特に読みの力が全然違うでゲソ。うーん、仕方ないでゲソ。こうなったら……、自分で囲碁の探索プログラムを作ってそれを使ってカンニングするでゲソ! 」
そう考えたλカ娘は、早速図書館で囲碁の探索に関する論文を探し始めた。しばらくして、『証明数と反証数を用いたλ探索』という論文と、『分岐因子が一様な探索空間のための AND-OR 木探索アルゴリズム』という博士論文、そして『ゲーム計算メカニズム』という書籍を見つけた。
「よし、これを基に実装するでゲソ。でも博士論文はちょっと読みこむの大変そうでゲソ。あくまで参考にとどめて、一番目の論文を基に実装するでゲソ」
さて、λカ娘が取り組んでいる囲碁の探索について、ここでもλカ娘に紹介してもらうことにしよう。なお、すでに分かっている読者も多いと思うが、今回の話は関数型とは関係なく、あくまでλつながりなだけ

である。

27.2　囲碁のルールおさらい

「囲碁のルールを説明するでゲソ! 囲碁のルールは細かいものまで説明していると結構大変なので、今回の記事に関係ある部分だけ説明するでゲソ! 以下の 4 つを把握しておけば十分でゲソ! 」

1. 黒白交互に盤の交点に打つ。1 手目は黒から (図 27.2)。ただし置き碁 (ハンデ戦) の場合は黒があらかじめ石を何個か置き、その後白から打ち始める。λカ娘が負けた対局は、黒のλカ娘が先に　4 個石を置いていた。

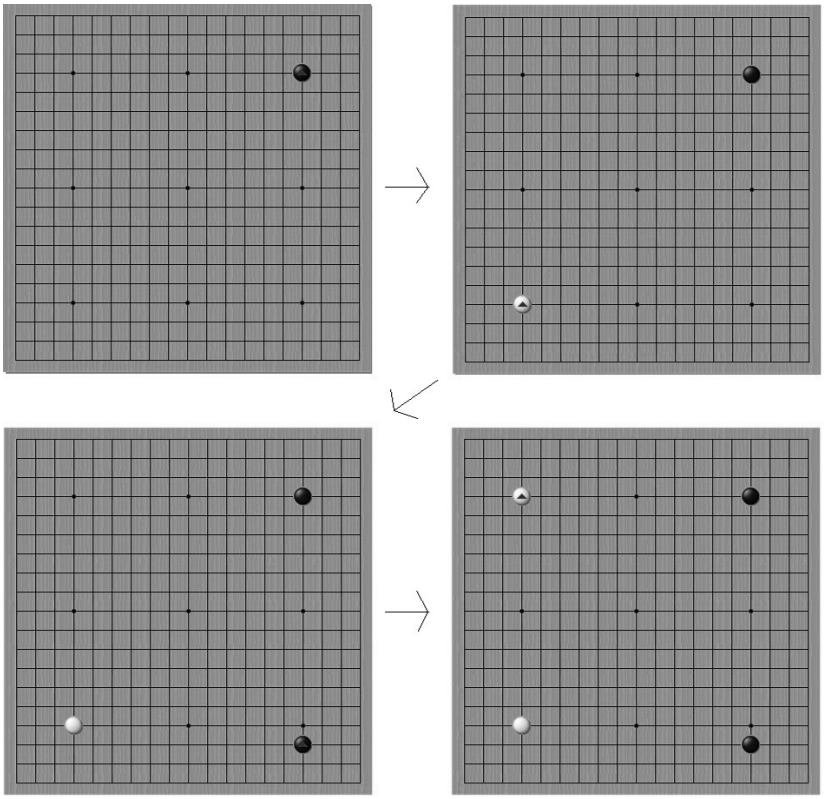


図 27.2: 交互に打つ

2. 相手の石を囲むと取れる (図 27.3。石を取るために打たなければならない場所の残りを『空(あき)ダメ』と呼ぶ。

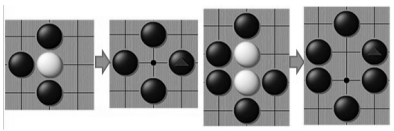


図 27.3: 石を取る

3. 自分から石を取られるような手は打てない (自殺手の禁止)。それ以外は空いている場所のほぼどこでも打てる。ただし一見自殺手に見えても、先に周りの相手の石を取れる場合は打てる (図 27.4)(『コウ』と

いう概念があって、厳密には打てない場所もあるが、今回は説明しない)。

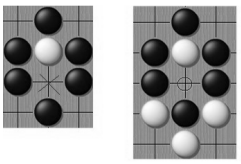


図 27.4: 白は×には打てないが○には打てる

4. 相手より大きな陣地を囲えば勝ち (図 27.5)。

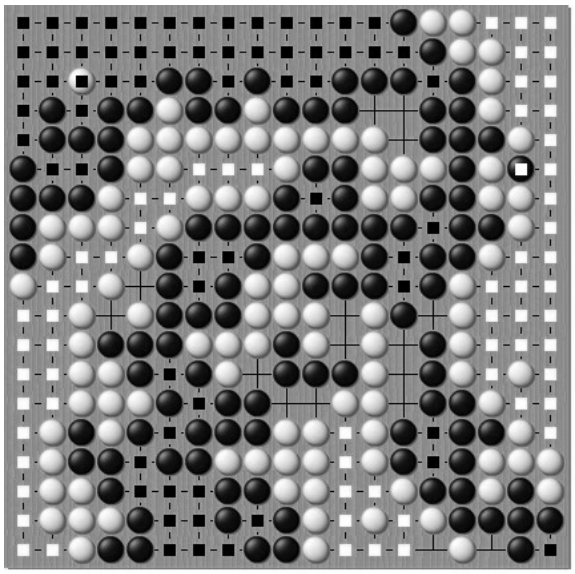


図 27.5: ■が黒の陣地、□が白の陣地。黒 3 目勝ち

27.3 そもそもゲームにおける探索とは何か

「通常、ゲームにおける探索とは、自分の手を考え、相手の手を考え、その次の自分の手を考え……ということをある一定の深さまで行うということを全通り試すことでゲソ! それにより、相手が最善に打ってきたときのこちらの最善手を調べるでゲソ! 人間がゲームにおいて先を読むことに相当するでゲソ! 例えば Tic-Tac-Toe(三目並べ) の場合、探索は図 27.6 のようになるでゲソ! 」

「もし完全な探索をゲームが終わる深さまですることが出来れば、ゲームの必勝手順 (ただし引き分けの場合もある) が見つかるでゲソ! ただ、Tic-Tac-Toe ならまだしも、オセロ、チェス、将棋、囲碁などのゲームは、当然ながら完全な探索を行うことは時間的にも記憶容量的にも不可能でゲソ!

だから、通常はある程度の深さで、最後まで探索した時の結果を予測する『**評価関数**』というものを呼ぶでゲソ! これは、ゲーム途中の局面で、どちらがどの程度優勢かを判断する関数に相当するでゲソ! 評価関数が正確なほど、探索が深いほど、一般には強い AI が作れるでゲソ! 」

「全通り試さなくても、探索の途中で出てきた評価関数の値を使うことで、結果の正確性を保ちつつ、その後の探索を大幅に削減する方法があるでゲソ! 詳しい説明は省くでゲソが、理想的な場合は、元の探索空間のサイズの平方根くらいまで探索空間が小さくなるでゲソ! これを α-β **探索**と呼ぶでゲソ! 」

27.4 囲碁で探索が難しい理由

「でも実は、囲碁でさっき説明したような探索をするのは難しいでゲソ。これには二つ理由があるでゲソ」

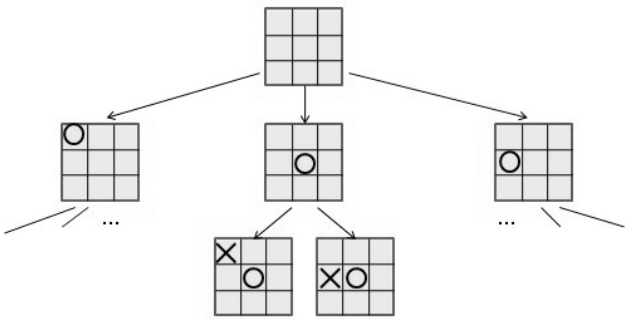


図 27.6: Tic-Tac-Toe(三目並べ) の探索の一部

「まず一つ目は、そもそも囲碁においては**正確で高速な評価関数を作るのが非常に難しい**、というか今のところ不可能なことでゲソ! チェスや将棋だと駒や駒組みにあらかじめ決められた点数をつけるとよい評価関数が作れるでゲソが、囲碁は無限とも言える石の組み合わせによって価値が生まれてくるためでゲソ! だからそもそも今まで述べてきたことを囲碁 AI に適用するのは無理があるでゲソ! 実際、昔の囲碁 AI はそのようにして作られていたでゲソが非常に弱くて、アマチュアの弱い方の人でも楽に勝てたでゲソ!

最近になってようやく、モンテカルロ木探索という、全く別の探索手法が出てきて、ようやくアマチュア高段者レベルになってきたでゲソ! ただ、今回はその話は書かないでゲソ!」

「二つ目の理由は、**探索空間が広すぎる**、つまり自分と相手の可能な手の数が大きすぎて、探索時間がかかりすぎることでゲソ。

詰碁などの部分的な読みに限った話にすれば、ある意味では『最後まで打って詰んだ』『最後まで打ったけど詰まなかった』という結果が出るので途中局面の評価関数がなくても何とかなる気もするでゲソ。実際、打つ手の範囲を限定した詰碁等は、α-β 探索でもある程度解けるでゲソ!

でも、詰碁の場合は α-β 探索ではあまり探索空間のサイズが削減できないでゲソ! これを詳しく説明するには α-β 探索をきちんと説明してないといけないので今回はしないでゲソが、大雑把な説明としては、囲碁における評価関数の値が『詰み』『不詰み』の二通りしかないことが、α-β 探索の不効率につながってると言えるでゲソ!

しかも、探索空間が削減できたとしても、囲碁の探索空間はすさまじく大きいので、打つ手の範囲を限定していない詰碁は未だに解けないでゲソ! それを解決するために、いろいろな探索手法が研究されているでゲソ! 大雑把に言うと、『詰みがありそうな手』だけを優先的に探索して『詰みがなさそうな手』は後回しにする、あるいは探索しないことで、正確性を犠牲にして速度を得るでゲソ。

今日はそういう話をしようじゃなイカ! 具体的には、そういう探索手法のうち、Df-pn、λ探索、 Df-pn λ 探索を紹介するでゲソ! 」

27.5 Df-pn

「Df-pn の解説をするでゲソ! Df-pn とは

- Allis の証明数探索を、長井が反復深化を利用して効率化したもの。

でゲソ! Df-pn は、**選択肢が少ない手順を優先的に探索する**方法でゲソ!

Df-pn は、詰将棋に非常に有効でゲソ! 1000 手以上の詰将棋も Df-pn によって解かれているでゲソ! これはなぜかと言うと、将棋では相手の選択肢を狭めるような手が有効なことが多いでゲソが、Df-pn を使うと自然と相手の選択肢を狭めるような手から探索するからでゲソ!

具体的には、**証明数、反証数**という『どれくらい選択肢が狭いか (証明数、反証数が小さい方が狭い)』を表す指標を計算して、それが小さい手から探索していくでゲソ!

ただ、囲碁ではどこに打っても証明数、反証数に差がつかないことが多いでゲソ。これは、囲碁だと盤面が大きすぎて、石を一つ置いたくらいでは選択肢はほとんど増減しないからでゲソ。証明数、反証数に差が出ないので、Df-pn も囲碁では少ししか有効でないでゲソ! 別の方法を考える必要があるんじゃなイカ!?

また、将棋でも、詰まないことを証明するのは難しいでゲソ!
なお、厳密な定義は以下のような感じでゲソ!」

- 証明された……ゴールが達成された場合。
- 反証された……ゴールが阻止された場合。
- 節点 n の証明数 $\mathrm{pn}(n)$……n を証明するために展開しなければならない葉節点数の最小値。
- 節点 n の反証数 $\mathrm{dn}(n)$……n を反証するために展開しなければならない葉節点数の最小値。
- $n_1,\ldots,n_k$ を n の子節点とすると、
  - n が証明された終端節点の時、$\mathrm{pn}(n)=0,\mathrm{dn}(n)=\infty$
  - n が反証された終端節点の時、$\mathrm{pn}(n)=\infty,\mathrm{dn}(n)=0$
  - n が未展開節点の時、$\mathrm{pn}(n)=\mathrm{dn}(n)=1$
- n が内部 OR 節点の時、

$$\mathrm{pn}(n)=\min_{i=1,\ldots,k}\mathrm{pn}(n_i),\quad \mathrm{dn}(n)=\sum_{i=1}^{k}\mathrm{dn}(n_i).\quad \text{(一つ証明するか、全部反証するか)}$$

- n が内部 AND 節点の時、

$$\mathrm{pn}(n)=\sum_{i=1}^{k}\mathrm{pn}(n_i),\quad \mathrm{dn}(n)=\min_{i=1,\ldots,k}\mathrm{dn}(n_i).\quad \text{(全部証明するか、一つ反証するか)}$$

- 探索ルール
  - OR 節点……最小の証明数を持つ子節点を選択。
  - AND 節点……最小の反証数を持つ子節点を選択。
  - 末端に到達したら、そこの葉節点を展開する。
  - 根節点が解けるまで繰り返す。

27.6 λ探索

「次はλ探索について説明するでゲソ ！λ探索は、

- Thomsen が考案した。
- 『脅威度』に基づいて節点の展開を抑制する探索手法。

でゲソ! それぞれの石について、『その石がどれくらい取られそうか』みたいな**脅威度**を定義しようじゃなイカ。取りたい石の脅威度に注目して、探索途中で『この石、脅威度が低くなって、なんだかもう取れなさそう』みたいになったら探索を打ち切れば、その石が取れそうな手を優先的に探索できてよさそうでゲソ。

この脅威度の定義には色々考えられるでゲソが、λ探索では、『 **λ次数** 』というものを使って脅威度を測るでゲソ。ざっくり言うと『λ次数』とは攻め方が何回連続で打てばその石を取れるか、という回数から 1 引いたものでゲソ。

λ次数が小さい方が石が取れそうで脅威度が高そう、λ次数が大きい方が石が取れなさそうで脅威度が低そうということになるでゲソ。そして、ある手順の探索の途中でλ次数がある閾値を超えたら、そこで探索を打ち切って別の手順を探索するでゲソ。

ただ、ある一つの石のλ次数に注目して探索するでゲソが、探索木のどこを展開するかとか、どの石に注目するのかとか、λ次数の閾値をいくつにするのかはいいやり方が決まっていないでゲソ。肝心なところが決まっていないでゲソね。λ次数の閾値は 3 以下でないと実用に耐えないという推測もされているでゲソ」

- $\lambda^n=1$……λ次数が n の時にゴールが達成された場合 (べき乗を意味しているわけではないので注意)。
- $\lambda^n=0$……λ次数が n の時にゴールが達成されない場合。
- 囲碁においては、ざっくり言うと以下のようになるでゲソ! また、図 27.7、27.8、27.9 も見てほしいでゲソ! これは、相手がパスをし続けるとすると、基本的には空ダメを埋めるだけで石が取れるからで

ゲソ!

- $\lambda^0 = 1$……攻め方が石を 1 つ連続で打てば取れる。
- $\lambda^1 = 1$……攻め方が石を 2 つ連続で打てば取れる状態が続いて、最終的に取れること。

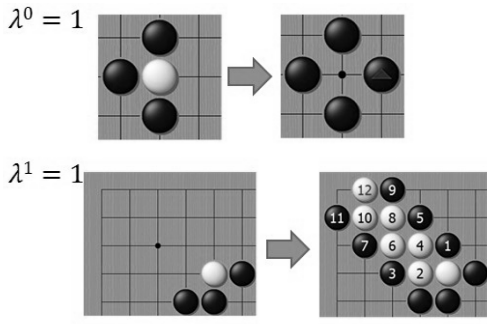


図 27.7: 取れる場合

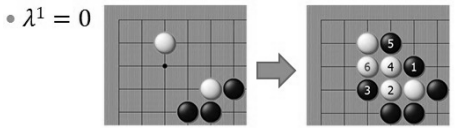


図 27.8: 取れない場合

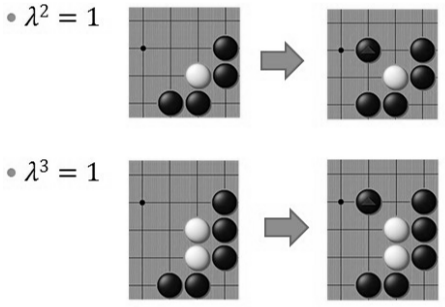


図 27.9: 相手が 2 回,3 回パスをすると取れるような手の方が、
相手が 1 回,2 回パスをしただけで取れる手より有効な場合もある。

- λ_a^0-手……直接ゴールが達成される攻め方の手。囲碁での、石を取る手など。
- λ_a^0-木……λ_a^0-手のみで構成される木。
  - λ_a^0-手を持つ λ_a^0-木……$\lambda^0 = 1$
  - それ以外の λ_a^0-木……$\lambda^0 = 0$
- λ^n-木 $(n \geq 1)$ の終端節点の値……攻め方手番なら $\lambda^n = 0$、受け方手番なら $\lambda^n = 1$(これ以上打つ手が無いということだから)。
- λ_a^n-手 $(n \geq 1)$……λ_a^n-手の直後に受け方がパスをすれば $\lambda_a^i = 1 (0 \leq i \leq n-1)$ となる λ_a^i-木が存在する手。
- λ_d^n-手 $(n \geq 1)$……λ_d^n-手の直後に $\lambda_a^i = 1 (0 \leq i \leq n-1)$ となっている λ_a^i-木が存在しない手。
- λ^n-木 $(n \geq 1)$……λ^n-手で構成される AND/OR 木。

「図で示すと図 27.10 のような感じになるでゲソ! 以下のような性質を持つでゲソ! 」

- $\lambda^n = 1 \Rightarrow \lambda^i = 1 (n \leq i)$

- $\lambda^n = 0 \Rightarrow \lambda^j = 0 (1 \leq j \leq n)$

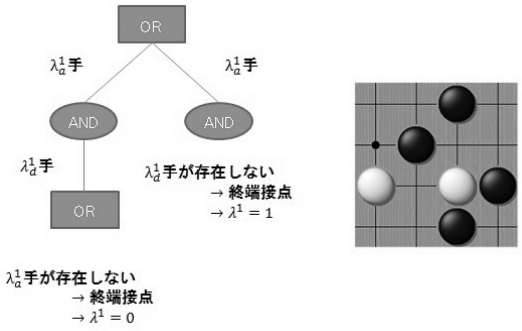


図 27.10: 探索の例

27.7 Df-pn λ探索

「Df-pn λ探索について説明するでゲソ ！Df-pn λ探索とは、」

- 証明数・反証数とλ次数の両方を考慮する探索 (根節点で、どのλ次数で探索を行うべきかはあらかじめ与える)。

「でゲソ! 2 つ合わせれば良くなるんじゃなイカ?! 以下のように定義されるでゲソ! 」

- $\mathrm{pn}^i(n)$ と $\mathrm{dn}^i(n)$ を用いる。
  - $\mathrm{pn}^i(n)$……各節点 n の各λ次数 i に対する証明数
  - $\mathrm{dn}^i(n)$……各節点 n の各λ次数 i に対する反証数
- 疑似節点……節点 n の疑似節点 c_l は、λ次数 l の探索を行うために n 内に仮想的に作られた節点。
- OR 節点 n において、探索のλ次数が l の時、λ次数 0 から l までの疑似節点を用意する。
- 証明数拡幅戦略……$n_1, \ldots, n_k$ をλ次数 l の OR 節点 n の子節点とした時、

$$\mathrm{pn}^i(c_i) = \min_{j=1,\ldots,k} \mathrm{pn}^i(n_j)$$
$$\mathrm{dn}^i(c_i) = \sum_{j=1}^{k} \mathrm{dn}^i(n_j)$$
$$\mathrm{pn}^l(n) = \min_{i=1,\ldots,l} \mathrm{pn}^i(c_i)$$
$$\mathrm{dn}^l(n) = \mathrm{dn}^l(c_l)$$

- (証明数拡幅戦略の説明)
  - $\mathrm{pn}^i(c_i)$ と $\mathrm{dn}^i(c_i)$ の計算は従来の証明数と反証数と同様。
  - $\mathrm{pn}^l(n)$ には最小の証明数を持つ疑似節点の証明数を入れる。
  - $\mathrm{dn}^l(n)$ には最大のλ次数を持つ疑似節点の反証数を入れる。
- (証明数拡幅戦略の考え方)
  - λ^i-木 $(i < l)$ が証明されれば n の証明は終了する。
  - λ^i-木は λ^l-木よりも探索空間が小さいことが多い。
  - できるだけ低いλ次数で探索を行いたい。
  - λ^i-木が反証された時は、λ^l-木も探索する必要がある (この場合 λ^i-木の探索は不要だったことになる)。
  - λ^i-木の反証の方が、λ^l-木の証明よりも難しいことが多い。

- 疑似節点のうち、最も小さな証明数を持つ疑似節点を先に展開する。
- λ次数 l の AND 節点 n では、通常の指し手に加え、パスも合法手であるとする。パス後の探索は、λ^{l-1}-木に移行する。
- $n_1,\ldots,n_k$ をλ次数 l の AND 節点 n の子節点とした時、

$$\begin{aligned}
&\mathrm{pn}^l(n) = \mathrm{pn}^{l-1}(c_{l-1}) + \mathrm{pn}^l(c_l) \quad \text{(合計戦略) または}\\
&\mathrm{pn}^l(n) = \mathrm{pn}^l(c_l) \quad \text{(最高次戦略)}\\
&\mathrm{dn}^l(n) = \min\left(\mathrm{dn}^{l-1}(c_{l-1}), \mathrm{dn}^l(c_l)\right)\\
&\mathrm{pn}^i(c_i) = \sum_{j=1}^{k} \mathrm{pn}^i(n_j)\\
&\mathrm{dn}^i(c_i) = \min_{j=1,\ldots,k} \mathrm{dn}^i(n_j)
\end{aligned}$$

- (説明)
  - n の子節点の選択は、反証数に基づく。λ^{l-1}-木と λ^l-木のうち、反証の容易そうな節点を先に展開する。
  - 合計戦略では n の証明数が大きくなるため、パス以下の探索が証明に有効でない場合には n の探索を後回しに出来る。
  - 合計戦略では証明数を大きく見積もることで、必要な探索が後回しになる場合もある。
  - 最高次戦略では、逆のことが生じる。
- (考え方)
  - 受け方が反証する手段は、2 通り。λ次数 l の指し手の一つを反証するか、パス後の節点の脅威度が $l-1$ より高いことを示すか。
  - 一般には λ^{l-1}-木は λ^l-木よりも小さいが、λ^{l-1}-木が証明されたときは λ^l-木の結果を調べる必要がある。
  - n を証明する際には、λ^l-木を証明する必要がある。

27.8 実験結果

「論文に載っている実験結果によると、Df-pn λ探索で囲碁の攻め合いの問題 110 問中 77 問が解けたということでゲソ! 単純な Df-pn の 75 問よりも少し多く、また単純な Df-pn より速く解けた問題もあるようでゲソ!」

27.9 エピローグ

「やっぱりカンニングは良くないでゲソ……囲碁は自分で考えてこそのゲームでゲソ! よし、囲碁の勉強やり直して、いつかあの老人に互先 (ハンデ無し) で勝てるように頑張るでゲソ!」

27.10 参考文献

- 証明数と反証数を用いたλ探索 (副田俊介、美添一樹、岸本章宏、金子知適、田中哲朗、ミュラーマーティン 著 情報処理学会論文誌 48(11),3455-3462,2007-11-15)
- 分岐因子が一様な探索空間のための AND-OR 木探索アルゴリズム (美添一樹 著、博士論文 人工知能学会誌 25(1),148,2010-01-01)
- ゲーム計算メカニズム (小谷善行 編著、岸本章宏、柴原一友、鈴木豪 共著 コロナ社)

なお、図 27.1 は@dif_engine さんに書いていただきました。

第28章

ロマンティック・パージング

— @xhl_kogitsune

本章は、某 LLVM 狐本[*1](の表紙) に端を発した非公式妄想です。キャラ設定・名前等は公式設定[*2]とは異なりますのでご注意ください。他所の実在の人物 (勿論きつねさん含む)・言語処理系等とは関係ありません。

キャラクター及びストーリーは『簡約! λカ娘 (4)』所収の「インターフェース」の続きにあたりますが、技術的内容は前作とは独立となっています。

登場人物紹介:

キヒメ : 金髪バックエンドきつねさん
コヨネ : 黒髪フロントエンドきつねさん

二人は仲良し。

28.1 ことのはじまり

「……ねえコヨネ、えるあーる、って、なに?」

はじまりは、キヒメのそんな一言だった。その一言で頭が一瞬真っ白になって、それまで何を話していたか忘れてしまった。

「……え?」

完全に固まりつつも、なんとか声を絞り出すわたし。

「……え?」

自分が何か変なこと言ったのか、と、首をかしげるキヒメ。かわいい。

「……おしおき確定」

「え? わたし、何も悪いことしてないよ!?」

落ち着こう。そう思って私は湯呑みに手を伸ばす。晴れたおだやかな昼下がり。ともすると殺風景になりそうな合理性と、かわいらしさが同居している空間 —— キヒメの部屋で、私たちは優雅にお茶をしていた……はずだったのだ。

「ありえないわ! バックエンドとはいえ、コンパイラやってて LR 知らないとか!」

落ち着けなかった。

「えへへ……よく分からないけど、何の話?」

「LR(k) 構文解析(パージング)よ!」

「あっ……わたし、構文解析は授業も受けたけど、あまり得意じゃなくて……」

「得意じゃないってレベルじゃ……!」

「……赤点で……」

「でもっ、さすがに言葉くらいは……」

「……れいてん、でした……」

「……」

[*1] 柏木餅子, 風薬 (著), 矢上栄一 (表紙): **3 日で出来る LLVM**, MotiPizza (2012).
http://motipizza.com/catalog/c82
http://d.hatena.ne.jp/motipizza/20120724
http://d.hatena.ne.jp/sabottenda/20120728

[*2] http://motipizza.com/member

唖然。でも、そこでキヒメがうつむいた顔を真っ赤にして泣きそうになっているのに気づいて、冷静さを取り戻した。よほど苦手で嫌な思い出だったのか……こういう形でいじめるのは本意ではない。……でも。私はキヒメの髪をなでながら、やさしく話しかける。

「ねぇキヒメ、構文解析は嫌い?」

「嫌いっていうか、よく分からないっていうか……」

キヒメの耳をもふもふしながら言葉を続ける。

「でも、私は構文解析好きよ。フロントエンドですもの。私としては、キヒメに私の好きなものを、もう少し嫌わないでほしいのだけれど……」

「……!!」

私の手の中でキヒメの耳が動いた。

「だから……ねぇ、キヒメ、少し私と構文解析の勉強をしてみない?」

「……じゃあ、コヨネ、よろしくおねがい、します」

そういうことになった。

28.2 構文解析と文脈自由言語

「とりあえず、分かっているとは思うけど基本的なことを確認していくわね」

「はい、分かっていてもコヨネの声で解説を聞きたい、です」

「……たとえば式

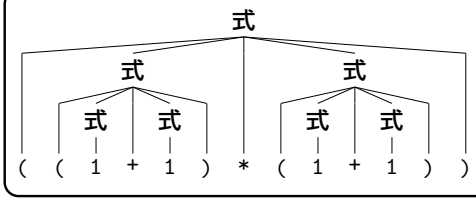
```
((1+1)*(1+1))
```

が与えられた時、この値を計算するにはどうすればいいかしら? * とか + の意味は普通のものだとしても、どの式とどの式を掛けたり足したりしているかを考える必要があるわね?

構文解析っていうのは、そういう文字列と、と文法規則 —— 例えば

式全体 は、**式** の形をしている
式 は、**(式+式)** か **(式*式)** か 1 のどれかの形をしている

が与えられた時、文法規則に従って、この文字列がどのような構造を持っているかを示す**構文木**



を作ることね。この構文木を見れば、* がどの式とどの式を掛け合わせているか、みたいなことも分かるわ。私みたいなコンパイラのフロントエンドは、プログラミング言語の文字列から、プログラミング言語の文法規則に沿って、関数や式などの構造を示した構文木を作って、解析や変形をして、最終的にキヒメみたいなコンパイラのバックエンドに渡すわけ」

このあたりのことは、フロントエンドの基礎の基礎あたりなので、さすがにキヒメも分かっていると思うけれど、キヒメは面白そうに私の話を聞いている。私の声を聞いている。

「文法規則は色々な種類があるけれど、プログラミング言語でベースになるのは**文脈自由文法**よ。厳密には文脈自由文法から少し逸脱している言語も多いのだけれど、今日はそのあたりの話はせずに文脈自由文法(と、文脈自由文法で書かれた文法規則に則った文字列の構文解析) について説明するわね。

文脈自由文法では、終端記号と非終端記号を使って生成ルールを記述することで文法規則を表現するわ。

さっきの例で `1+*()` の 5 種類の文字は実際の入力文字列の中に出てくる文字で、こういうものを **終端記号** と呼ぶわ。逆に言うと、入力文字列は終端記号からなる列、ってなるわね。ちなみに、行の終端を示す記号、とかじゃないわよ。あと、『文字』と言ったけれど、文字以外の何かでもいいわ。フロントエンドの実装でも、構文解析の前に**字句解析**を行って、`while` とか `!=` とかの文字のかたまりに分割してから、そのかたまり一つを終端記号一つとして構文解析することが多いわ。

一方で、**式全体** や **式** は実際の文字列の中には現れない、構文木の内部ノードを構成する記号で、そのような記号を **非終端記号** と呼ぶわ。

文法規則は、非終端記号から、非終端記号または終端記号からなる列への変換規則、みたいな**生成ルール**で表現できるわ。さっきの例だと、

$S \to E$
$E \to (E+E)$
$E \to (E*E)$
$E \to 1$

こういう感じね (**式全体** を S に、**式** を E に置き換えてそれっぽくしてみたわ)。

そして、特別な非終端記号である**開始記号** S から始めて、**生成ルールの左辺を右辺で置き換える**、つまり
『$S \to E$　つまり　S を E に置き換える』、
『$E \to (E+E)$　つまり　E を $(E+E)$ に置き換える』、
『$E \to (E*E)$　つまり　E を $(E*E)$ に置き換える』、
『$E \to 1$　つまり　E を 1 に置き換える』、
という操作を繰り返して、終端記号の列 `((1+1)*(1+1))` に到達できればいい、ということ。実際にやってみると下のようになって (下線を引いた部分が次の行で置き換えられているわ)、この過程を追うことで構文木が作れるわ」

$\underline{S}$
$\to \underline{E}$
$\to (\underline{E}*E)$
$\to ((\underline{E}+E)*E)$
$\to ((1+\underline{E})*E)$
$\to ((1+1)*\underline{E})$
$\to ((1+1)*(\underline{E}+E))$
$\to ((1+1)*(1+\underline{E}))$
$\to ((1+1)*(1+1))$

「文脈自由文法をフォーマルに定義するとこうなるわ」

文脈自由文法 (*Context Free Grammar, CFG*) $G = (\Sigma, N, P, S)$ とは、
終端記号 (*terminal*) の有限集合 Σ、
非終端記号 (*non-terminal*) の有限集合 N、
開始記号 (*the starting symbol*) $S \in N$、
生成ルール (*production*) の有限集合 P の組。
ここで、生成ルールとは、
「非終端記号 A から、終端記号または非終端記号の列 $\gamma \in (\Sigma \cup N)^*$ を生成する」$A \to \gamma$
という形をしているもの。
開始記号 S からはじめて、生成ルールの左辺 A を右辺 γ で置き換える、というのを繰り返して得られる終端記号列の集合を、文脈自由文法 G の生成する **文脈自由言語** (*context free language*) という。
記号の使い方:
終端記号はタイプライタ体の文字 (例: $\mathtt{1}, \mathtt{+}, \mathtt{*}, \mathtt{(}, \mathtt{)} \in \Sigma$) で、
非終端記号は大文字英字 (例: $S, E \in N$) で表します。
終端記号を表す変数は小文字英字 (例: $a \in \Sigma$) で、
終端記号と非終端記号の列を表す変数はギリシャ文字 (例: $\beta, \gamma, \delta \in (\Sigma \cup N)^*$) で表します。
長さ 0 以上の終端記号列全体の集合を Σ^*,
長さ 0 以上の終端記号と非終端記号からなる列全体の集合を $(\Sigma \cup N)^*$ のように表記します。

「コヨネせんせー、CFG っていったら *Control Flow Graph*[*3] じゃないんですかー?」
「はいキヒメさん、テンプレな反応ありがとう。両方 CFG で紛らわしいので気をつけましょう。相手と違う CFG のことについて話しているのにしばらく誤解に気づかなかったりもするわ[*4]。今日は Context Free Grammar の方しか出てこないので大丈夫ね。

さて、キヒメが分からないって言ってた『えるあーる』は、この文脈自由文法の一部を構文解析する bottom-up 構文解析の手法の一つね。実際には LR を元にした LALR が `yacc` などの構文解析器ジェネレータでよく使われていて、プログラミング言語の構文解析に広く使われているわ」

28.3 Bottom-Up 構文解析

「さっきの話では、構文木を上から下へ、つまり開始記号 S から始めて、$A \to \gamma$ の生成ルールの**左辺を右辺に**置き換えていって、最終的に終端記号列 $w \in \Sigma^*$ に到達したけれど、bottom-up 構文解析では逆に何か終端記号列 $w \in \Sigma^*$ が与えられて、そこから $A \to \gamma$ の生成ルールの**右辺を左辺に**置き換えていって (これを *reduce* する、と言うわ)、開始記号 S まで下から上に到達するものよ。

例として、以下の文脈自由文法 G_{expr}[*5] を考えましょう。今日はこの文法を使って構文解析の説明をするわね」

$$
\begin{aligned}
\text{文脈自由文法 } G_{\text{expr}} &= (\Sigma_{\text{expr}}, N_{\text{expr}}, P_{\text{expr}}, S_{\text{expr}}) \\
\Sigma_{\text{expr}} &= \{\texttt{1}, \texttt{+}, \texttt{(}, \texttt{)}\} \\
N_{\text{expr}} &= \{S, E\} \\
P_{\text{expr}} &= \{\ S \to E, \\
&\qquad E \to E\texttt{+(}E\texttt{)}, \\
&\qquad E \to \texttt{1}\} \\
S_{\text{expr}} &= S
\end{aligned}
$$

は

$$
\begin{aligned}
\text{文脈自由言語 } L_{\text{expr}} = \{&\texttt{1}, \\
&\texttt{1+(1)}, \\
&\texttt{1+(1+(1))}, \\
&\texttt{1+(1)+(1)}, \ldots\}
\end{aligned}
$$

を生成する。

「$w =$ `1+(1)+(1)` を構文解析する時には、

$\underline{\texttt{1}}\texttt{+(1)+(1)}$
$\to E\texttt{+(}\underline{\texttt{1}}\texttt{)+(1)}$
$\to \underline{E\texttt{+(}E\texttt{)}}\texttt{+(1)}$
$\to E\texttt{+(}\underline{\texttt{1}}\texttt{)}$
$\to \underline{E\texttt{+(}E\texttt{)}}$
$\to \underline{E}$
$\to S$

[*3] プログラムの制御構造を表現したグラフ。コンパイラの中間表現で多用される
[*4] 他に類似の語として「言語処理」(自然言語処理/プログラミング言語処理) とか「L2」(L2 ノルム/L2 キャッシュ) とかあって、いずれも主に自然言語処理屋とコンパイラ屋の間で誤解の元となります
[*5] この文脈自由文法 G_{expr} の例、及び以降で出てくるパーズ対象の終端記号列は、
University of California, Berkeley の CS 164: Programming Languages and Compilers
`http://www-inst.eecs.berkeley.edu/~cs164/sp06/lectures.shtml`
の講義資料
`http://www-inst.eecs.berkeley.edu/~cs164/sp06/lectures/lecture07lr-2.pdf`
で使われている例を、`int` を `1` に置き換えて使いました (但し説明方法は異なります)。

という感じで S に到達できるわ (各ステップで下線部分が reduce されているわ)」

「……なるほど。でも、各ステップでどこを置き換えてもいいの?」

　このあたりから理解が怪しくなってくるのか、キヒメの耳の毛が少しだけ逆立っている。

「そうでもないわ。変な置き換え方すると成功する (S に到達できる) はずのものも失敗する (S に到達できずに詰む) ことがあるし、アルゴリズムを工夫して終端記号列を左から右に順に読みながら置き換えていけばいい、ということにできれば実行効率とか色々便利だったりするわ。

　そういう bottom-up 構文解析アルゴリズムの一つのクラスが、shift-reduce 構文解析ね。これは、終端記号と非終端記号を積んでいくスタックを用意して、構文解析対象の終端記号列 $w \in \Sigma^*$ を左から右に順番に読み込みながら、*shift* と *reduce* の 2 つの操作で構文解析を進めていくの」

shift：w から次の一文字を読んでスタックに積む (push する)
reduce：生成規則 $A \to \gamma$ の右辺 γ がスタック末尾と一致したら、γ と一致した部分を A に置き換える

「終端記号列を左から shift していって、適当なタイミングでおもむろに reduce する、って感じね」

「……んんん、すたっく?」

「たぶん、実際に見てみるのが分かりやすいわ。次のページからの図を見てちょうだい。

　構文解析途中の状態は、『スタックの中身』『w 内での現在の読み込みカーソル位置』(カーソル $\triangleright$ で表します) の二つよ。以下の図では、スタックは右側がスタックトップ (push/pop される末端) よ。ついでに、reduce する時に構築する構文木も一緒に示しておくわね」

　最初は、スタックは空で、カーソルは先頭。

状態 0

スタック: (空)

カーソル位置: ▷ 1+(1)+(1)

構文木: ▷ 1+(1)+(1)

↓ shift します (1 文字読み込みます)。

状態 1

スタック: 1

カーソル位置: 1 ▷ +(1)+(1)

構文木: 1 ▷ +(1)+(1)

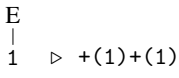
↓ reduce: 下線部分が $E \to 1$ の右辺と一致するので reduce します。

状態 2

スタック: E

カーソル位置: 1 ▷ +(1)+(1)

構文木:



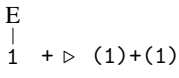
↓ shift します。

状態 3

スタック: E+

カーソル位置: 1+ ▷ (1)+(1)

構文木:



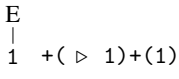
↓ shift します。

状態 4

スタック: E+(

カーソル位置: 1+(▷ 1)+(1)

構文木:



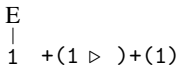
↓ shift します。

状態 5

スタック: E+(1

カーソル位置: 1+(1 ▷)+(1)

構文木:



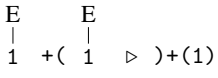
↓ reduce: 下線部分が $E \to 1$ の右辺と一致するので reduce します。

状態 6

スタック: E+(E

カーソル位置: 1+(1 ▷)+(1)

構文木:



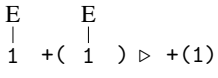
↓ shift します。

状態 7

スタック: E+(E)

カーソル位置: 1+(1) ▷ +(1)

構文木:



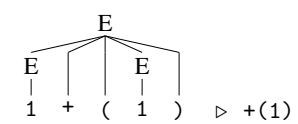
↓ reduce: 下線部分が $E \to E$+(E) の右辺と一致するので reduce します。

状態 8

スタック: E

カーソル位置: 1+(1) ▷ +(1)

構文木:



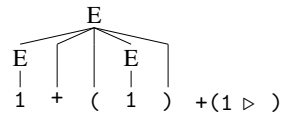
↓ 3 回 shift します。

状態 11

スタック: E+(1

カーソル位置: 1+(1)+(1 ▷)

構文木:



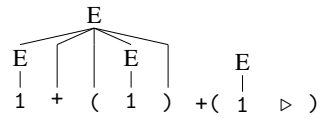
↓ reduce: 下線部分が $E \to 1$ の右辺と一致するので reduce します。

状態 12

スタック: E+(E

カーソル位置: 1+(1)+(1 ▷)

構文木:



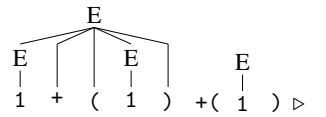
↓ shift します。

状態 13

スタック: E+(E)

カーソル位置: 1+(1)+(1) ▷

構文木:



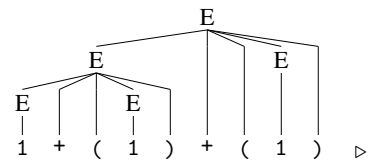
↓ reduce: 下線部分が $E \to E$+(E) の右辺と一致するので reduce します。

状態 14

スタック: E

カーソル位置: 1+(1)+(1) ▷

構文木:



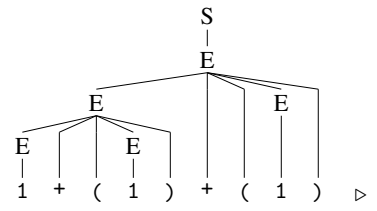
↓ reduce: 下線部分が $S \to E$ の右辺と一致するので reduce します。

状態 15

スタック: S

カーソル位置: 1+(1)+(1) ▷

構文木:



28.4 LR 構文解析

28.4.1 LR item

「キヒメ、ここまではいいかしら?」

「うん、d.y.d. でも見たし[*6]」

その割には反応が怪しかったけれど、やはり凹んでいるキヒメよりも強がっているキヒメの方がかわいい。

「じゃあ、これからどうやって『shift するか、reduce するか、どの生成ルールで reduce するか』を決める話に入るわ」

「? shift できそうな時は shift して、reduce できそうな時には reduce すればいいんじゃないの?」

「そんな『食べたい時に食べて、眠りたい時に眠ればいいんじゃないの』みたいなこと言って……」

「食べたい時にコヨネのごはんが食べたい、です」

[*6] k.inaba. 『Derive Your Dreams』 http://www.kmonos.net/wlog/
「ボトムアップ・テガキパーサ」 http://www.kmonos.net/wlog/109.html#_2034100523
「構文解析の話をしよう」 http://www.kmonos.net/wlog/83.html#_1021080304

どうやら私はキヒメの餌付けに成功していたらしい。
「いきなりそんな真面目な声で言われても……とかにく、目先だけ見ると shift も reduce もできそうな場合があるから、何らかの情報を使ってどれをするか決めなきゃいけない時があるの。その決め方の方法の一つが **LR(k) 構文解析**と呼ばれるもので、これがさっきキヒメが分からないって言ってた『えるあーる』ね」
「えるあーるこわい……」
「大丈夫、怖くないわ」
「こわい……」
「……それは『まんじゅうこわい』みたいな話かしら?」
「えるあーるこわくない」
「よろしい」
「コヨネこわい。だからコヨネいっぱいちょうだい」
……キヒメはたまに急に変なことを言い出す。
「何よ、一人じゃご不満?」
「いや、そういうわけじゃないんだけど……」
なにやら考え込むキヒメ。たまに、この子が何考えているのか分からないことがある。
「……何考えてるのか知らないけど、話を戻すと、たとえば状態 6、

状態 6	スタック: E+(E カーソル位置: 1+(1 ▷)+(1)

では、次に) が来た時に shift もできるし、スタックだけ見ると $S \to E$ で reduce もできるように見えるわ。でも、実際にはここでは shift を選ぶの。何でだと思う?」
「え? あーうん、がんばればコヨネ三人くらいまでならいけるんじゃないかな」
何が。繰り返すけど、この子が何考えているのか分からないことがある。
「あー、ごめん。なんか shift はできそうだけど reduce はできなさそうな雰囲気がする、から?」
脱線から回復するキヒメ。話自体は聞いていた、ようだ。
「具体的に、なんでそういう雰囲気がするのかって説明できるかしら?」
「えー……っと、) を shift したらスタックが E+(E) になってそれっぽい感じになるし、ここで $S \to E$ で reduce しても

状態 16	スタック: E+(S カーソル位置: 1+(1 ▷)+(1)

みたいになって、なんかそれっぽくない、とか? この後は 1 を E に reduce する以外は shift する以外に選択肢がなくなって

状態 17	スタック: E+(S)+E カーソル位置: 1+(1)+(1) ▷

こんな感じになって詰むのかなぁ」
「そんな感じね。状態 6 では

1. 『$E \to E$+(E) **の途中まで来ている感じ**』

があるから shift していい気がするし、ここで $S \to E$ で reduce すると詰むのは、まぁ色々捉え方はあるけど、たとえば状態 17 でそもそも

2. 『S **の後に**) **が来るのっておかしい**』

からダメって感じがするわよね。この感じ、つまり

1. **どの生成ルールの、どのあたりの途中まで来ているか**
2. **その生成ルールで reduce した後にどんな終端記号列が来るはずか**

を表現するのが、**LR**(k) **項** (*LR*(k) *item*) というものなの」

LR(k) 項は

$$[A \to \beta \bullet \gamma, \delta]$$

という形をしているわ。ここで、

$A \to \beta\gamma \in P$ は生成ルール、
$A \in N$ は非終端記号、
$\beta, \gamma \in (\Sigma \cup N)^*$ は長さ 0 以上の終端記号と非終端記号の列、
δ は**先読み記号列** (*lookahead symbols*) と呼ばれる長さ k 以下の終端記号の列

ね。これは

1. 現在、生成ルール $A \to \beta\gamma$ の途中、β までの部分を処理済で、
   これから γ に対応する部分を処理すれば A に reduce できる
2. そうして A に reduce した後は次に終端記号列 δ が来るはずだ

っていう雰囲気。

1. • が途中にある場合は、そのすぐ右にある記号を shift すれば • が一つ右に移動して、
2. • が一番右にある場合 ($[A \to \beta\gamma\bullet, \delta]$) は、カーソルのすぐ右に終端記号列 δ が来ていればこの生成ルールで A に reduce できる、

ということで、shift や reduce ができるかどうか、を表現しているわけね

「k は先読み記号列 δ の最大長で、増やせば増やすほど情報が増えるので LR(k) 構文解析が構文解析できるものの幅は広がるけれど、実用上は 1 つ先読みすれば、つまり LR(1) を使えばだいたい十分なので LR(1) が使われることが多いわ。

$k = 1$ の場合の LR(k) item、つまり **LR**(1) **項** (*LR*(1) *item*) は、先読み記号列が長さ 1、つまり終端記号 $a \in \Sigma$ 一つのみの

$$[A \to \beta \bullet \gamma, a]$$

または長さ 0 の記号列 $ (空記号列; 入力の終端を示す)

$$[A \to \beta \bullet \gamma, \$]$$

になるわね。

これから先は LR(1) の場合で説明するけど、基本的に別の LR(k) でも同じよ」

28.4.2　一本道

「ああ、• はもう二度と見たくなかったわ……」
「LR item[*7] が分からないの?」
「うん。構文解析の授業も、LR item の話をしているうちになんか途中で一気に分からなくなった気がするし」

また少し元気がなくなるキヒメ。

「『shift したいような気持ち』『reduce したいような気持ち』を LR item で表現する、っていうのはなんとなく分かったけど、具体的にどういう風に動くのか、よく分からないというか」
「現在の状況を表現する LR item を持って、現在の LR item とカーソルの次にある終端記号を見て shift するか reduce するかを決めて、LR item を更新する、みたいな感じかしら」
「『今 • の所にいるの』みたいな説明は昔聞いた気がするけどいまいちピンとこなくて……」
「• が現在位置だから、それが右に進んでいくだけよ?」
「えー、ほんと? それだけで終わるものではなかった気がするんだけど……」

[*7] 正しくは LR(k) item とか LR(1) item なのでしょうが、以下では (k) の部分を省いてこう書きます

……私に教師役は似合わない。キヒメに対して裏表なくやさしくするなんて柄でもない。私のキヒメに対する表面的なやさしさには、大抵裏の個人的嗜好があるのだ。でも、珍しく元気のないキヒメを見て、私は、やさしく、本当にやさしく、キヒメに教えるのであった。裏を欠いたわたしと、どこか大人しいキヒメ。今までのキヒメとの付き合いの中で、かなり稀少といえた。たまにはこういうのもいいのかもしれないと思いつつも、やはりいつも通り元気にツンツンしているキヒメに戻ってほしい。

「確かにそれが全てではないけれど……大丈夫よ、少しずつ進んでいけば。いきなりフルバージョンの LR(1) 構文解析の話をすると混乱しそうだから、少し段階を追って雰囲気だけの『なんちゃって LR(1)』から説明を始めて、少しずつフルバージョンに近づけていきましょうか」

「コヨネやさしー」

「私はいつだって優しいわよ?

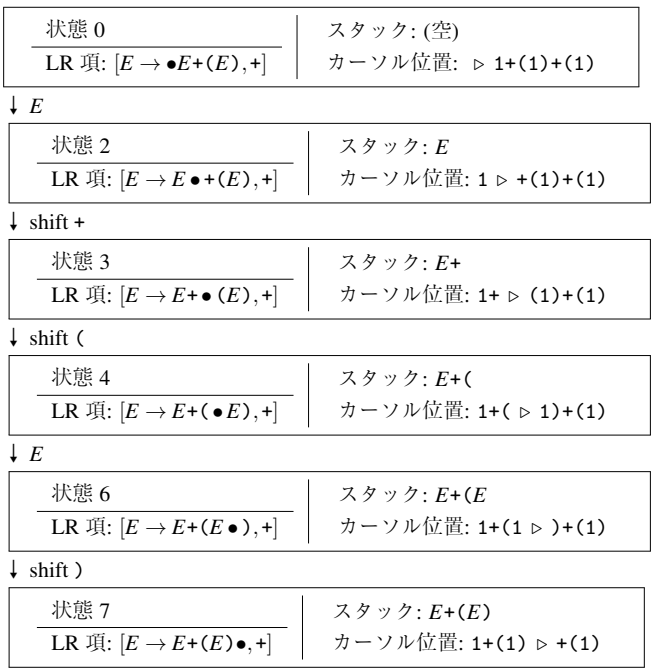
まず、一番基本的なところでは『記号を 1 つずつ読んでいって、LR item の中の • を 1 つずつ右にずらしていく』って感じかしら。

たとえば、状態 0 から状態 8 までは、「生成ルール $E \to E\texttt{+(}E\texttt{)}$ に対応する部分をパーズして、次に + が来る」という部分だから、

$[E \to \bullet E\texttt{+(}E\texttt{)}, +]$ から始めて (状態 0)、
E を入力するとスタックに E を積んで $[E \to E \bullet \texttt{+(}E\texttt{)}, +]$ になって (状態 2)、
+ を入力するとスタックに + を積んで $[E \to E\texttt{+} \bullet \texttt{(}E\texttt{)}, +]$ になって (状態 3)、
(を入力するとスタックに (を積んで $[E \to E\texttt{+(} \bullet E\texttt{)}, +]$ になって (状態 4)、
E を入力するとスタックに E を積んで $[E \to E\texttt{+(}E \bullet \texttt{)}, +]$ になって (状態 6)、
) を入力するとスタックに) を積んで $[E \to E\texttt{+(}E\texttt{)} \bullet, +]$ になって (状態 7)、
+ が次なら $E \to E\texttt{+(}E\texttt{)}$ で reduce (状態 8 へ)

みたいな感じね」

LR(1) 項の状態遷移 (1): 一本道



28.4.3 離れたり戻ったり

「なるほど!!! • を右に動かしていけばいいんだね!!」

「だからそう言ったじゃない」

「……ん、でもコヨネせんせー、やっぱりただ右に動かして終わるものではなくない? この『E を入力す

る』って、どうやるの? shift の部分は状態の番号が 1 ずつ進んでるのに、E の部分だけ番号が飛んでるし」
「いい質問です。• のすぐ右にあるのが終端記号なら、単純に shift すればいいんだけど、E は非終端記号なので、終端記号のように直接終端記号列から読んできて shift することができません。なので、非終端記号については生成ルールを使って reduce することでゲットします。

たとえば、状態 0 の場合だと E を入力したいのだけれど、生成ルール $E \to 1$ が使えそうなので、一度元の LR item から離れて、『$E \to 1$ を reduce したい気持ち』になります。具体的には、LR item

$$[E \to \bullet 1, +]$$

つまり『これから 1 を読んで、$E \to 1$ で reduce して、次に + を読みたい気持ち』になります。そして、1 を shift して、$E \to 1$ で reduce したら、また元の LR item の系列に戻ってきます」

LR(1) 項の状態遷移 (2): 離れたり戻ったり

状態 0 | スタック: (空)
LR 項: $[E \to \bullet E+(E), +]$ | カーソル位置: ▷ 1+(1)+(1)

↓ E を入力したいので、一旦生成ルール $E \to 1$ のシーケンスに入る

状態 0′ | スタック: (空)
LR 項: $[E \to \bullet 1, +]$ | カーソル位置: ▷ 1+(1)+(1)

↓ shift 1

状態 1 | スタック: 1
LR 項: $[E \to 1\bullet, +]$ | カーソル位置: 1 ▷ +(1)+(1)

↓ reduce $E \to 1$

↓ reduce したら元の生成ルールのシーケンスに戻る

状態 2 | スタック: E
LR 項: $[E \to E \bullet +(E), +]$ | カーソル位置: 1 ▷ +(1)+(1)

↓ shift +

状態 3 | スタック: E+
LR 項: $[E \to E+\bullet (E), +]$ | カーソル位置: 1+ ▷ (1)+(1)

↓ shift (

状態 4 | スタック: E+(
LR 項: $[E \to E+(\bullet E), +]$ | カーソル位置: 1+(▷ 1)+(1)

↓ E を入力したいので、一旦生成ルール $E \to 1$ のシーケンスに入る

状態 4′ | スタック: E+(
LR 項: $[E \to \bullet 1,)]$ | カーソル位置: 1+(▷ 1)+(1)

↓ shift 1

状態 5 | スタック: E+(1
LR 項: $[E \to 1\bullet,)]$ | カーソル位置: 1+(1 ▷)+(1)

↓ reduce $E \to 1$

↓ reduce したら元の生成ルールのシーケンスに戻る

状態 6 | スタック: E+(E
LR 項: $[E \to E+(E \bullet), +]$ | カーソル位置: 1+(1 ▷)+(1)

↓ shift)

状態 7 | スタック: E+(E)
LR 項: $[E \to E+(E)\bullet, +]$ | カーソル位置: 1+(1) ▷ +(1)

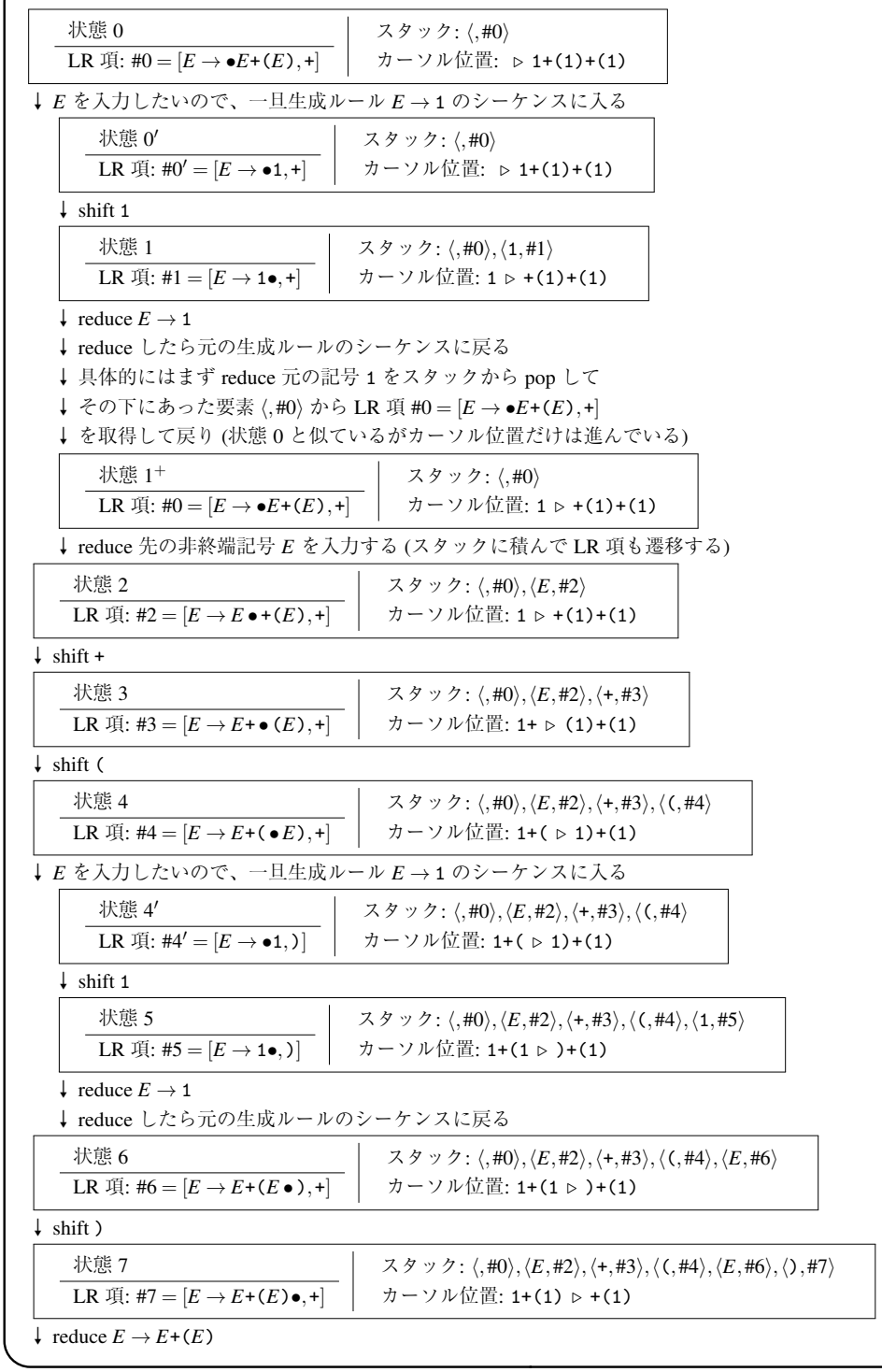
↓ reduce $E \to E+(E)$

「んー、でもどうやって戻るの? LR item には戻る情報書いてないよね?」
「LR item の中じゃなくて、スタックに積んでおくの。具体的には、スタックに『記号だけ』ではなく

『⟨ 記号,LR item⟩ の組』を積んでおけば、reduce する時に元の LR item が見れるわ。あとは、最初の LR item #0 にも戻れるようにするために、最初にスタックにはダミー ⟨,#0⟩ を突っ込んでおくといいわね。

たとえば次の図 (単純化のために、LR 項に番号を振っておくわ) では、状態 1 から reduce する時に、スタックを pop してその下の要素を見ることで (状態 1^+)、元の LR item の情報を得ているわ」

LR(1) 項の状態遷移 (3): スタックによる実装



「スタック! そういうのもあるのね!!」

「意外?」

「うん、なんか LR item だけで状態と状態遷移が表現される気がしてて。でもなるほどー、call/return みたい

な感じだね」
「そうね。LR 分かりそう?」
「えへへ、なんかちょっと分かる気がしてきた」
　それは何より。

28.4.4　LR item から LR item set へ

「さて、今までは各状態で『こういうのをパーズしたい』っていう LR(1) 項を 1 つだけ取り出していたけれど、実際には複数の『これ以降の終端記号の出方によってはこういうのがパーズできるかも』っていう候補の LR 項が存在するわ。
　たとえば、状態 $4'$ で、上の図では
『E を入力したいので、一旦生成ルール $E \to \mathtt{1}$ のシーケンスに入る』
に対応して

$$[E \to \bullet \mathtt{1}, \mathtt{)}]$$

　という LR 項しか書かなかったけれど、この他にも
『E を入力したいので、一旦生成ルール $E \to E\mathtt{+(}E\mathtt{)}$ のシーケンスに入る』
に対応する

$$[E \to \bullet E\mathtt{+(}E\mathtt{)}, \mathtt{)}]$$

　という LR 項も可能なはずよね。
　結果的には $[E \to \bullet \mathtt{1}, \mathtt{)}]$ しか使えないんだけど、状態 $4'$ でのカーソル位置まででは両方ありえるから、両方考えておく必要があるわ。どっちが結果的に正しかったかは、実際にパーズを進めていかないと分からないもの。なので、LR 構文解析の状態は、LR 項 1 つだけではなく、可能な LR 項を全て集めた **LR 項の集合** (*LR item set*) にするの」
「うっ……なんだか難しそう」
「大丈夫よ。今まで話した、LR item が何で、どんな感じでパーズが進んでいくか分かって、複数の LR item の可能性を保持しつつ進めていくことを把握しておけばまぁ困ることはないと思うわ。まぁ、せっかくなので、LR item set の作り方とかも説明しておきましょう。これからの話が、ほぼ厳密な版の LR パーザになるわ」
「おー」
「LR item set の場合、つまり複数 LR item がある場合でも、基本的な原理は LR item が 1 つの時と変わらないわ。後できちんとまとめるけど、複数の LR item それぞれについて考えた結果を総合する感じ」
「つまり、shift すると LR item set のそれぞれの LR item の • が右に移動していく感じ?」
「そうね。shift の条件に合わない LR item は消えていくわ。
　一つ新しいのは、『E を入力したいので、一旦生成ルール $E \to \mathtt{1}$ のシーケンスに入る』みたいな可能性を全て考える、って所かしら。非終端記号 $Y \in N$ に対して、現在の LR item set s に LR item

$$[X \to \alpha \bullet Y \beta, a]$$

が入っていれば、『Y を入力したいので、一旦生成ルール $Y \to \gamma$ のシーケンスに入る』みたいな可能性に対応する LR item

$$[Y \to \bullet \gamma, b]$$

も s に入れる必要があるの。この操作を closure を取る、みたいに言うことがあるわ」

`closure(`s`)`: LR item set s の closure を取る
　s に含まれる $[X \to \alpha \bullet Y\beta, a]$ の形をした LR item と、$Y \to \gamma$ という各生成ルールについて、
　『Y を入力したいので、一旦生成ルール $Y \to \gamma$ のシーケンスに入る』
という可能性に対応する LR 項

$$[Y \to \bullet\gamma, b]$$

　を s に追加する。
　ここで、b は「Y を入力して reduce した次に来る可能性のある終端記号」、つまり βa に対応する終端記号列の先頭に来うる終端記号 ($b \in \mathrm{First}(\beta a)$ などと書かれる)。複数ある場合にはそれぞれに対応する LR 項を追加する。
　これを、追加する LR 項がなくなるまで繰り返す。

「たとえば、closure($\{[E \to E\texttt{+(}\bullet E\texttt{)}, \$], [E \to E\texttt{+(}\bullet E\texttt{)}, \texttt{+}]\}$) の場合 (状態 4)[*8]、$\bullet$ のすぐ右に E が来ているので、E が左辺に来る生成ルール、つまり

$E \to \texttt{1}$
$E \to E\texttt{+(}E\texttt{)}$

に対応する LR item

$[E \to \bullet\texttt{1}, \texttt{)}]$
$[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{)}]$

を追加するわ。$[E \to E\texttt{+(}\bullet E\texttt{)}, \$]$ と $[E \to E\texttt{+(}\bullet E\texttt{)}, \texttt{+}]$ の中で、$\bullet$ の次の E の次に来るのは `)` だったので ($\mathrm{First}(\texttt{)}\$) = \mathrm{First}(\texttt{)+}) = \{\texttt{)}\}$)、先読みトークンは `)` になっているわ。
　更に、$[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{)}]$ について、$\bullet$ のすぐ右にあるのは E でその次に来るのは `+` なので、E が左辺に来る生成ルールそれぞれに対応する、先読みトークンが `+` の LR 項

$[E \to \bullet\texttt{1}, \texttt{+}]$
$[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{+}]$

を追加します。
　ここで追加する LR 項が尽きるので、

$$\begin{aligned}\mathrm{closure}(\{[E \to E\texttt{+(}\bullet E\texttt{)}, \$], [E \to E\texttt{+(}\bullet E\texttt{)}, \texttt{+}]\}) = \{&[E \to E\texttt{+(}\bullet E\texttt{)}, \$],\\ &[E \to E\texttt{+(}\bullet E\texttt{)}, \texttt{+}],\\ &[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{)}],\\ &[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{+}],\\ &[E \to \bullet\texttt{1}, \texttt{)}],\\ &[E \to \bullet\texttt{1}, \texttt{+}]\}\end{aligned}$$

となるわ。
　この closure に、実は状態 $4'$ も包含されていることに注意ね。つまり、『E を入力したいので、一旦生成ルール $E \to \texttt{1}$ のシーケンスに入る』結果の状態 $4'$ は、実は状態 4 から遷移した別の状態ではなくて、状態 4 に内在する可能性の一つ、みたいな感じかしら。だって、状態 4 と状態 $4'$ とではスタックの内容とカーソルの位置は同じだし、『E を入力したいので、一旦生成ルール $E \to \texttt{1}$ のシーケンスに入る』っていうのにもこの時点では根拠がなくて、この後のパーズの進展によって結果的に『E を入力したいので、一旦生成ルール $E \to \texttt{1}$ のシーケンスに入る』という可能性が選択された、というだけのことなの。
　さて、まとめましょう。パーズを開始する時の LR item set と、shift や reduce をした時にどのような LR item set に遷移するのか (+ スタックをどう操作するか) を決めましょう」

[*8] 実際には $[E \to E\texttt{+(}\bullet E\texttt{)}, \$]$ は前の図の状態 4 には出てきていないけれど、最後に実際に LR item set を構築する時にはこの closure を取る必要があるのでこの例で説明するわ。まぁあまり気にする必要はないわ

開始状態: LR item $[S \to \bullet\beta, \$]$ が開始記号 S のパーズ開始に対応するので、その closure を取った LR item set $s_0 = \text{closure}(\{[S \to \bullet\beta, \$]\})$ から開始します。スタックには s_0 に戻るためのダミー要素のみを積んで、カーソル位置は先頭です。

状態 S	スタック: $\langle, s_0\rangle$ (ダミー要素のみ)
LR item set: $s_0 = \text{closure}(\{[S \to \bullet\beta, \$]\})$	カーソル位置: ▷ ... (先頭)

shift 状態遷移: 現在 (状態 P) の LR item set s に $[X \to \alpha \bullet y\beta, b] \in s \quad (y \in \Sigma)$ という LR item が含まれる場合、カーソルの次に終端記号 $y \in \Sigma$ が来れば y で shift できる。

shift y した場合、カーソルを 1 つ右に (y の右に) 動かし、状態は LR item set s から LR item set $t = \left\{[X \to \alpha y \bullet \beta, b] \;\middle|\; [X \to \alpha \bullet y\beta, b] \in s\right\}$ に遷移し、スタックに $\langle y, t\rangle$ を積む (状態 Q)。

つまり、LR item $[X \to \alpha \bullet y\beta, b]$ は y を shift すると $\bullet$ が y の分だけ一つ右に動いて $[X \to \alpha y \bullet \beta, b]$ になる、っていうのを全ての LR item について考えるってことね。$\bullet$ の右にあるのが y 以外だったら、その LR item の可能性は消えることになるわ。

状態 P	スタック: $..., \langle ?, s\rangle$
LR item set: $s = \{..., [X \to \alpha \bullet y\beta, b], ...\}$	カーソル位置: ... ▷ y...

↓ shift y

状態 Q	スタック: $..., \langle ?, s\rangle, \langle y, t\rangle$
LR item set: $t = \{..., [X \to \alpha y \bullet \beta, b], ...\}$	カーソル位置: ...y ▷ ...

reduce 状態遷移: 現在 (状態 P) の LR item set s に $[A \to \gamma\bullet, a] \in s$ という LR item がある場合、カーソルの次に終端記号 $a \in \Sigma$ が来れば生成ルール $A \to \gamma \in P$ で reduce できる。

生成ルール $A \to \gamma \in P$ で reduce した場合、まずスタックのトップから γ に相当する分だけ pop する (状態 P^+)。γ の長さ (記号数) を m とすると、実はスタックのトップから m 個分が γ と必ず一致するので、スタックの中身は見ずに m 個 pop すればいい。

次に、pop した後の新しいスタックのトップ要素 $\langle ?, r\rangle$ を見て、LR item set r に A を入力して LR item set $t = \left\{[X \to \alpha A \bullet \beta, b] \;\middle|\; [X \to \alpha \bullet A\beta, b] \in r\right\}$ に遷移し、スタックに $\langle A, t\rangle$ を積む (状態 Q)。

下線部が γ に相当する、つまり、$c_1 ... c_m = \gamma$。

説明のために状態 P^+ をはさんだけど、実際には A から B までが reduce 1 回の遷移よ。

状態 P	スタック: $..., \langle ?, r\rangle, \underline{\langle c_1, ?\rangle, ..., \langle c_m, s\rangle}$
LR item set: $s = \{..., [A \to \gamma\bullet, a], ...\}$	カーソル位置: ... ▷ a...

↓ reduce $A \to \gamma$ (前半: pop する)

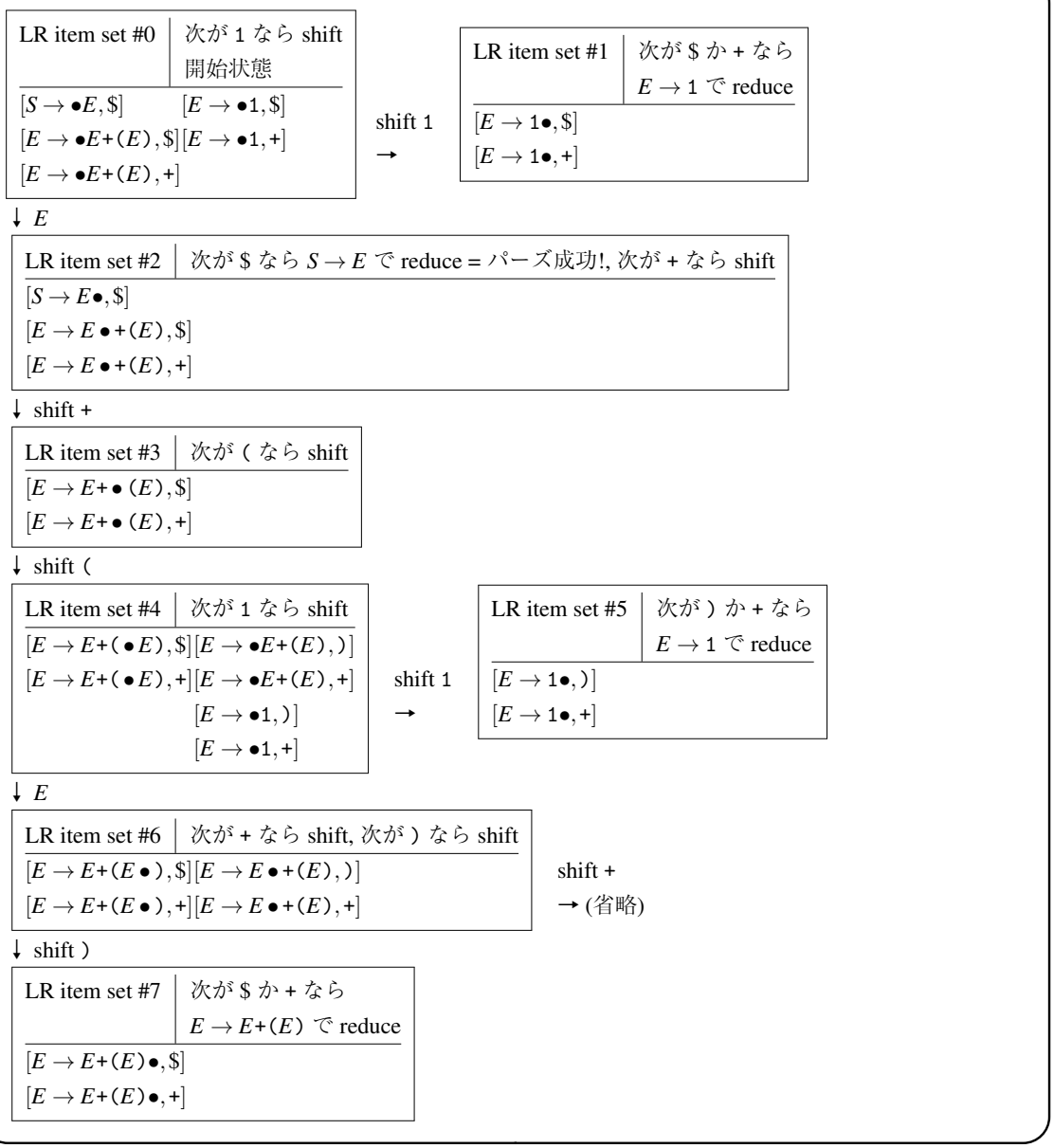
状態 P^+	スタック: $..., \langle ?, r\rangle$
LR item set: $r = \{..., [X \to \alpha \bullet A\beta, b], ...\}$	カーソル位置: ... ▷ a...

↓ reduce $A \to \gamma$ (後半: A を入力する)

状態 Q	スタック: $..., \langle ?, r\rangle, \langle A, t\rangle$
LR item set: $t = \{..., [X \to \alpha A \bullet \beta, b], ...\}$	カーソル位置: ... ▷ a...

「shift も reduce も、現在の LR item set s とカーソルの次の終端記号 a が分かれば、shift や reduce できるか、できる場合遷移先の LR item set t が何かが一意に決まる (スタックを見る必要はない) ので、それを表現する、LR item set を頂点とする状態遷移図が描けるわ。LR item set 全体は有限個とはいえかなり多いけれど、開始状態の LR item set s_0 から始めて、shift や reduce で到達可能な LR item set だけ考えればいいわ」

LR item set 状態遷移図



「あ、この LR item set の遷移図見たことある気がする! 昔見た時はよくわからない袋小路が多いとか思ったけど、実際の状態遷移はこの LR item set 遷移図の中で閉じていないから、reduce → スタックを pop して状態を巻き戻す → 遷移、みたいな経路が見えないけど存在するから袋小路じゃない、っていうことなんだね」

「そうね。shift の場合は shift と書かれた枝を、reduce $A \to \gamma$ の場合はスタックを pop して、元の LR item set r に戻ってから (ここでどの LR item set に戻るかというのはスタックに積まれる情報なので、この LR item set 遷移図には書かれていないわ)、その r から生成ルール左辺の A が書かれた枝を、それぞれ辿ればいいわ。

この LR item set を使って、`1+(1)+(1)` のパーズ過程を書くと次のページの図になるわ」

LR(1) 項の状態遷移 (4): LR item set

状態 $0 = 0'$ / LR item set: #0 | スタック: $\langle$,#0$\rangle$ / カーソル位置: ▷ `1+(1)+(1)`

↓ shift `1`

状態 1 / LR item set: #1 | スタック: $\langle$,#0$\rangle$,$\langle$`1`,#1$\rangle$ / カーソル位置: `1` ▷ `+(1)+(1)`

↓ reduce $E \to$ `1` (スタックを pop し、#0 に戻ってから E を積んで遷移する)

状態 2 / LR item set: #2 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$ / カーソル位置: `1` ▷ `+(1)+(1)`

↓ shift `+`

状態 3 / LR item set: #3 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$ / カーソル位置: `1+` ▷ `(1)+(1)`

↓ shift `(`

状態 $4 = 4'$ / LR item set: #4 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$ / カーソル位置: `1+(` ▷ `1)+(1)`

↓ shift `1`

状態 5 / LR item set: #5 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle$`1`,#5$\rangle$ / カーソル位置: `1+(1` ▷ `)+(1)`

↓ reduce $E \to$ `1`

状態 6 / LR item set: #6 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle E$,#6$\rangle$ / カーソル位置: `1+(1` ▷ `)+(1)`

↓ shift `)`

状態 7 / LR item set: #7 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle E$,#6$\rangle$,$\langle$`)`,#7$\rangle$ / カーソル位置: `1+(1)` ▷ `+(1)`

↓ reduce $E \to E$`+(`E`)` (スタックから 5 つ pop し、#0 に戻ってから E を積んで遷移する)

状態 8 / LR item set: #2 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$ / カーソル位置: `1+(1)` ▷ `+(1)`

↓ shift `+`, shift `(`, shift `1`

状態 11 / LR item set: #5 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle$`1`,#5$\rangle$ / カーソル位置: `1+(1)+(1` ▷ `)`

↓ reduce $E \to$ `1`

状態 12 / LR item set: #6 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle E$,#6$\rangle$ / カーソル位置: `1+(1)+(1` ▷ `)`

↓ shift `)`

状態 13 / LR item set: #7 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$,$\langle$`+`,#3$\rangle$,$\langle$`(`,#4$\rangle$,$\langle E$,#6$\rangle$,$\langle$`)`,#7$\rangle$ / カーソル位置: `1+(1)+(1)` ▷

↓ reduce $E \to E$`+(`E`)`

状態 14 / LR item set: #2 | スタック: $\langle$,#0$\rangle$,$\langle E$,#2$\rangle$ / カーソル位置: `1+(1)+(1)` ▷

↓ 受理! reduce $S \to E$ して終了

「やり方さえ分かっちゃえば簡単なもんだね!」
「その台詞は変なフラグを立てていないかしら?」
　でもまぁ、いつものキヒメに戻っている感じがするので、甲斐があったというものだろう。
「ん、でもこれって決定的なの? LR item が複数あるのはいいけど、同じステート、同じ先読みトークンで shift も reduce もできたりしたらどうするの?」
「それは *shift/reduce conflict* ね。reduce の方法が複数ある *reduce/reduce conflict* とかもあるわ。shift/reduce conflict の場合は shift を優先したり、文法を修正して conflict が起きないようにとか色々な話はあるけど ―― 今日はここまでにしましょう」
　そう言って私はノートを閉じた。
　いつのまにかかなり時間が経っていたらしく、キヒメの部屋も射し込む夕日で赤く染まっていた。
「夕陽、きれいだね……」
　キヒメもそれに気づいたのか、顔を上げてそうつぶやく。赤い光。キヒメと二人きり。それは、あの告白の時と同じで――
「……」
「……」
　ふとキヒメと眼が合った。夕日以外のもので赤く染まる頬。キヒメも、そしておそらく私も。

　急に無言になってしまった私たちは、そのまましばらく、夜の帳が下りていくのを黙って一緒に眺めていたのだった。

第29章

HaskEll Shaddai

— @fumieval

```
data Person = Person
    { name :: String, self :: Object, endurance :: Int, equipment :: Equipment }

data Object = Object
    { position :: Vector3
    , velocity :: Vector3
    }

data Equipment = Equipment
    { sword :: Object
    , body :: ArmorType
    }

data Vector3 = Vector3
    { x :: Float
    , y :: Float
    , z :: Float
    }
addV3 :: Vector3 -> Vector3 -> Vector3
```

「そんなレコードで大丈夫か？」
「大丈夫だ、問題ない」

```
enoch = Person
    { name = "Enoch"
    , endurance = 2
    , equipment = Equipment
        { sword = Object
            { position = Vector3 0 0 0
            , velocity = Vector3 0 0 0 }
        , body = Bronze}}
```

```
modify $ \e -> put e { endurance = endurance e - 1   }
```

「グゥッ！」

```
modify $ \e -> put e
    { equipment = equipment e
        { sword = sword (equipment e)
            { position = addV3 (position (sword (equipment e)))
                (velocity (sword (equipment e))) }}}
```

```
modify $ \e -> put e
    { equipment = equipment e
        { sword = sword (equipment e)
            { velocity = velocity (sword (equipment e))
                { x = x (velocity (sword (equipment e))) + 5 }}}}
```

```
modify $ \e -> put e { endurance = endurance e - 1   }
```

「ヌゥッ！」

神は言っている、ここで死ぬ定めではないと…

第29章

HaskEll Shaddai

— @fumieval

「そんなレコードで大丈夫か？」
「一番いいのを頼む」

```
import Control.Lens

data Person = Person
    { _name :: String, _self :: Object, _endurance :: Int
    , _equipment :: Equipment }

data Object = Object
    { _position :: Vector3
    , _velocity :: Vector3
    }

data Equipment = Equipment
    { _sword :: Object
    , _body :: ArmorType
    }

data Vector3 = Vector3
    { _x :: Float
    , _y :: Float
    , _z :: Float
    }

makeLenses ''Person
makeLenses ''Object
makeLenses ''Equipment
makeLenses ''Vector3
```

```
do
    zoom (equipment . sword) $ do
        velocity . x += 5
        v <- use velocity
        position %= addV3 v
    self . velocity . z += 3
```

29.1 Lens

未来に思いを馳せることはいいことだが、地に足のついた考え方をしないと夢物語でしかない。つまり、私が君たちの前に現れるのは、まだ先ということだ。

```
import Control.Applicative

type LensLike' f s a = (a -> f a) -> s -> f s

lens :: Functor f => (s -> a) -> (s -> a -> s) -> LensLike' f s a
lens getter setter f s = fmap (setter s) (f (getter s))

_1 :: Functor f => LensLike' f (a, b)   a
_1 f (a, b) = fmap (\a' -> (a', b)) (f a)

contains :: (Ord a, Functor f) => a -> LensLike' f (Set.Set a)   Bool
contains a f s = fmap (\b -> if b
    then Set.insert a s
    else Set.delete a s)
    $ f (Set.elem a s)
```

ん？ この関数か？ んー……これは Lens だ。私はあまりタフガイなプログラミングは好きではない。だが、これが構造に対するアクセスに優れた武器であることは認めるよ。`LensLike' f s a` という型は、「s から a を取り出すことができ、また a に対する変更は s に反映できる」という意味を持っている。

え？ 発動の仕方がわからないのか？……あーすまない。Lens にはいくつかの形態がある。

```
type Getting r s a = LensLike (Const r) s a
type Getter s a = Getting a s a -- (a -> Const a a) -> s -> Const a s

view :: Getter s a -> s -> a
view g = getConst . g Const
```

これは Getter 形態。Const a s という型の値は、s に関わらず常に a の値を持っている。Lens に Const を渡すと、Const に包まれた a が返ってくるから、それを剥がせばいい。

```
view _1 (360000, 14000)
= { view の定義 }
getConst $ _1 Const (360000, 14000)
= { _1 の定義 }
getConst $ fmap (\a' -> (a', 14000)) (Const 360000)
```

```
= { fmap }
getConst $ Const 360000
= { getConst }
360000
```

```
> let ws = Set.fromList $ words "The quick brown fox jumps over the lazy dog"
> view (contains "fox") ws
True
```

そして…

```
type Setter' s a = LensLike' Identity s a

over :: Setter' s a -> (a -> a) -> s -> s
over l f = runIdentity . l (Identity . f)

set :: Setter' s a -> a -> s -> s
set l a = over l (const a)
```

これが Setter 形態だ。これは一部分を変更する関数を受け取り、全体を変更する。Lens は、関数に値を渡した後、全体を復元する関数を fmap する。だから、fmap がそのまま中身への関数適用になる Identity を使えばいいんだ。

```
over _1 (*2) (9, 1)
= { over の定義 }
runIdentity $ _1 (Identity . (*2)) (9, 1)
= { _1 の定義 }
runIdentity $ fmap (\a' -> (a', 1)) ((Identity . (*2)) 9)
= { 計算 }
runIdentity $ fmap (\a' -> (a', 1)) (Identity 18)
= { fmap }
runIdentity $ Identity (18, 1)
= { runIdentity }
(18, 1)
```

```
> let ws = Set.fromList  $ words "The quick brown fox jumps over the lazy dog"
> set (contains "fox") False ws
fromList ["The","brown","dog","jumps","lazy","over","quick","the"]
```

お前にあと 2〜3 本 Functor があれば、他の形態にもできたんだが、残念だったな…

Functor ならいくらでも持ってるって？……すまん、では見せてあげよう、Lens のもう一つの形態を。

```
data Store a s = Store (a -> s) a deriving Functor

type ALens s a = LensLike (IStore a) s a
```

```
cloneLens :: Functor f => ALens s a -> LensLike f s a
cloneLens l f s = let IStore bt a = l (IStore id) s in fmap bt (f a)
```

これは ALens。Lens と同等の機能を持っているが、こちらはリストなどのコンテナの要素にすることができる。それにしても、不思議な変形をするだろう。だがこれは newtype による変形ではない。きれいに特殊化しているだけなんだ。嘘だと思うなら自分で作ってみるといい。

```
cloneLens _1 f (e1, e2)
= { cloneLens の定義 }
let IStore bt a = _1 (IStore id) (e1, e2) in fmap bt (f a)
= { _1 の定義 }
let IStore bt a = fmap (\a' -> (a', e2)) (IStore id e1) in fmap bt (f a)
= { fmap }
let IStore bt a = IStore (\a' -> (a', e2)) e1 in fmap bt (f a)
= { let の束縛 }
fmap (\a' -> (a', e2)) (f e1)
= { _1 の定義 }
_1 f (e1, e2)
```

```
> view (_1 . _1) ((1,2), (3,4))
1
```

おぉ。Lens は独自定義のデータ型と違い、Prelude の (.) で左から右に合成できる。性質を関数に集中させているため、非常に強い記述力を誇っている。また、id は Lens になるし、任意の Lens は id と合成しても変わらない。

すべての Lens には、以下の「Lens 則」という制約が課されている。これを満たさないものは Lens としての価値がないから気をつけろ。

すべての `Lens`、`l` **に対し**:
`view l (set l b a) == b`
`set l (view l a) a == a`
`set l c (set l b a) == set l c a`
が成り立つ。

実は Getter には、神が作り出した知恵の一つ、モナドを埋め込むことができる。Const の代わりに、このような Functor を使うだけで済む。純粋な Getter も、副作用のある処理も、すべて (.) で合成できるのは嬉しいと思わないか？

```
newtype Effect m r a = Effect { getEffect :: m r }  deriving Functor
instance (Monad m, Monoid r) => Applicative (Effect m r) where
    pure = return mempty
    Effect ma <*> Effect mb = Effect (liftM2 mappend ma mb)

perform :: LensLike' (Effect m a) s a -> s -> m a
perform g = getEffect . g (Effect . return)
```

29.2 Traversal

リストの n 番目の要素にアクセスしたい場合を考えてみよう。実は、これは Lens では実現できない。実行時エラーを出したいなら話は別だがな。というのも、リストの範囲外を指定した場合が表現できないからだ。そこで、Functor の代わりに Applicative を使ってみよう。

```
type Traversal' s a = forall f. Applicative f => LensLike' f s a
nth :: Traversal' [a] a
```

こちらはうまく表現できる。まあ見てくれ。

```
nth 0 f (x:xs) = fmap (:xs) (f x)
nth n f (x:xs) = fmap (x:) (nth (n - 1) f xs)
nth _ _ [] = pure []
```

Getter、Setter は与えられた `a -> f a` に値を渡すことでアクセスを表現する。ここで `pure` を代わりに使えば、アクセスの対象から外すことができる。この仕組みは Traversal と呼ばれている。Applicative は Functor のサブクラスだから、Traversal は Lens の特殊な場合だ。

Identity は Applicative になるから、Setter 形態として使える。

```
> over (nth 3) (*10) [0..9]
[0,1,2,30,4,5,6,7,8,9]
> over (nth 30) (*10) [0..9]
[0,1,2,3,4,5,6,7,8,9]
```

ところが、Getter として使うには注意が必要だ。`Const r` が Applicative になるには、r が Monoid でなければいけないため、`view` は一般には使えない。そこで、First を使って最初に見つかった値を返すようにしよう。

```
preview :: Getting (First a) s a -> s -> Maybe a
preview g = getFirst . getConst . g (Const . First . Just)
```

ほら、値が取り出せる場合は Just、取り出せなかった場合は Nothing を返すことだってできる。

```
> preview (nth 3) [0..9]
Just 3
> preview (nth 30) [0..9]
Nothing
```

モノイドの制約がついたこのような Getter には、Fold という名前がついている。Fold ということは foldr などができるのかだって？……自分で試してみるといい。

29.3 Iso と Prism

29.3.1 Profunctor について

Functor は「出口」に関数を適用するのに対し、Profunctor は入口と出口のそれぞれに関数適用ができる構造です。
具体的には、

```
dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
```

が定義できます。
例えば関数 (->) は Profunctor になります。

```
dimap f g h = f . h . g
```

Profunctor のサブクラスとして、Strong と Choice の二つの型クラスがあります。Strong は first' :: Strong p => p a b -> p (a, c) (b, c)、Choice は left' :: Choice p => p a b -> p (Either a c) (Either b c) が使えるという能力があります。

よしわかった、説明しよう。これは Iso だ。神が創り出した知恵の一つ。いや、アクセサか。さ、まずは定義を調べてみるか。

```
import Data.Profunctor

type Iso' s a = forall p f. (Profunctor p, Functor f) => p a (f a) -> p s (f s)

iso :: (s -> a) -> (a -> s) -> Iso s a
iso f g = dimap f (fmap g)

enum :: Enum a => Iso' Int a
enum = iso toEnum fromEnum
```

見ての通り、独自のデータ型すらない美しい定義だろ？

```
> view enum 97 :: Char
'a'
> over enum toUpper 97
65 -- toEnum 65 == 'A'
```

逆関数の関係にある関数の対から Iso は構築される。Lens と同等の記述力を持つが、Setter や Getter などとして使うだけでなく、関係を逆転させることができる。

```
data Exchange a s t = Exchange (s -> a) (a -> t)

from :: (Exchange a a (Identity a) -> Exchange a s (Identity s)) -> Iso' a s
from i = let Exchange sa bit = i (Exchange id Identity)
    in dimap (runIdentity . bit) (fmap sa)
```

```
> view (from enum) 'a'
97
> over (from enum) (+1) 'a'
'b'
```

Exchange に一旦特殊化するという非常にトリッキーな方法を使う。すると、from を適用することで、元々と正反対の Lens のようになる。Exchange がどう Profunctor のインスタンスになるか分からないって？……またアーク Haskeller に訊いてみるんだな。

p を Profunctor のサブクラスである Choice、f を Functor のサブクラスである Applicative とすると、Prism と呼ばれるものになる。表の姿は Traversal だが、逆向きの Getter としても使うというところが大きく異なる。

```
type Prism' s a = forall p f.(Choice p,Applicative f) => p a (f a) -> p s (f s)

newtype Reviewed a b = Reviewed { runReviewed :: b }

instance Profunctor Reviewed where
    dimap _ f (Reviewed b) = Reviewed (f b)

instance Choice Reviewed where
    left' (Reviewed b) = Reviewed (Left b)
    right' (Reviewed b) = Reviewed (Right b)

review :: (Reviewed a (Identity a) -> Reviewed s (Identity s)) -> a -> s
review p = runIdentity . runReviewed . p . Reviewed . Identity

prism :: (a -> s) -> (s -> Either s a) -> Prism' s a
prism f g = dimap g (either pure (fmap f)) . right'

_Left :: Prism' (Either a b) a
_Left = prism Left (\s -> either Right (\_ -> Left s) s)

_Right :: Prism' (Either a b) b
_Right = prism Right (\s -> either (\_ -> Left s) Right s)
```

コンストラクタのように、作ることは常に可能でも取り出せるかは不明な場合に Prism は役立つ。

```
> review _Left 72
Left 72
> preview _Left (Left 72)
Just 72
> preview _Left (Right ())
```

```
Nothing
```

今日紹介した数々の知恵は、すべて lens というパッケージで実装されている。Lens をより扱いやすくするための演算子が沢山定義されているから、少し見てみよう。

(^.)、(^?)、(^!) はそれぞれ view、preview、perform の中置演算子版だ。左側には値、右側に Getter が来る。(%~) は over の中置演算子版で、引数の順番は over と変わらない。派生として、値を置き換える (.~)、四則演算を行う (+~)、(-~)、(*~)、(//~) などもある。さらに、~を=にすれば、State モナドの状態に対して Setter を発動させることができる。こいつはなかなか便利だ。

```
> execState (do {_1 .= 7000; _2 += 35; both *= 2 }) (360000, 1)
(14000,72)
```

29.4 Real World Lens

ところで、Lens を応用すれば JSON や XML などの人間の知恵の結晶を扱えるというのは、君たちならもうわかるはずだ。

lens-aeson という知恵を紹介しよう。lens-aeson は、JSON データを Lens で操ることができる……とても簡単にね。

```
> :m Control.Lens.Aeson Control.Lens
> "{\"hoge\": \"homuhomu\", \"fuga\": 32, \"piyo\": 64}" ^? key "hoge" . _String
Just "homuhomu"

> key "fuga" . _Number *~ 100 $ "{\"hoge\": 16, \"fuga\": 32, \"piyo\": 64}"
"{\"hoge\":16,\"piyo\":64,\"fuga\":3200}"
```

鍵になるのは、key、_Number、_String だ。key は、指定されたキーにアクセスする Traversal。

```
key :: AsValue t => Text -> Traversal' Text t Value
```

この AsValue というクラスは、JSON データの型である Value のほかに、String や ByteString もインスタンスになっていて、それらは JSON としてパースしてくれる。

_Number、_String は、JSON として解釈できるデータと数値や文字列を変換する Prism だ。

```
_Number :: AsNumber t => Prism' t Number
_String :: AsPrimitive t => Prism' t Text
```

AsNumber、AsPrimitive も同様に、Value、String、ByteString がインスタンスになっている。そいつらを好きなように合成すれば、もう自由自在だ。私はこれなしで JSON を扱うなんて考えられないがね。

そして、xml-lens は XML を lens の力で操る。HTML ファイルのタイトルを取得するのだって、ほら。

```
> :m Text.HTML.DOM Text.XML.Lens Network.HTTP.Conduit
> html <- simpleHttp "http://matome.naver.jp/odai/2134010966376465501"
> let doc = Text.HTML.DOM.parseLBS html
> Text.HTML.DOM.parseLBS doc ^? root . entire . ell "title" . text
Just "最強のビニール傘はどこのコンビニ製か - NAVER まとめ"
```

この root、entire、ell、text はすべて Lens 及び Traversal になっている。

```
root :: Lens' Document Element
entire :: Traversal' Element Element
ell :: Text -> Traversal' Element Element
text :: Traversal' Element Text
```

それぞれ、一番上の要素を取り出す、すべての要素を検索する、指定した名前を持つ要素を取り出す、要素の内容を取り出すという意味を持っている。これらを組み合わせることによって、どんなに複雑な階層構造でも狙い撃ちできるんだ。勿論、ただ読みとるだけでなく、特定の要素を書き換えることもできるから、色々試してみるといい。そうそう、下の階層にアクセスしたいときは、(./) 演算子で合成することを忘れないように。(.) とどう違うかだって？ うーん……実際に合成すればわかるんじゃないか？

```
> doc ^? root . localName
Just "html"
> doc ^? root ./ localName
Just "head"
> doc ^? root ./ id ./ localName
Just "meta"
> doc ^? root ./ ell "body"
Just "body"
```

もうこの武器の説明はいいか？ 分からないことがあれば、あとはアーク Haskeller に訊いてくれ。

29.5 リンク

- http://hackage.haskell.org/package/lens lens パッケージ
- http://hackage.haskell.org/package/lens-aeson lens-aeson のドキュメント
- http://hackage.haskell.org/package/xml-lens xml-lens のドキュメント

第VI部

簡約!? 入力娘　Rock! (2013年冬)

第30章

λカ娘探索 2?

— @ark_golgo

30.1 プロローグ

「……私の 1 目勝ちじゃなイカ？」
「そのようじゃな。ヨセで細かくなったとは思っておったが、逆転には至らなかったようじゃ。」
「3 子で初勝利でゲソ！　はぁ。ふらふらでゲソ！」
「この半年で置石 2 つ分くらい強くなったようじゃの。いやはやたまげたわい。お主と打ち始めた当初は 4 子でも楽勝だったがの。」
　λカ娘は　C84 のころからある老人と囲碁を打つようになった。当初は 4 子のハンデ (段級位差にして 4 段級差に相当) をつけてもらってもまるで勝てなかったが、最近は 3 子でもいい勝負が出来るようになっていた。そして、今回 3 子で初勝利を収めた。
「やっぱり囲碁ソフトと打ちまくったのが良かったんじゃなイカ？」
「囲碁ソフト？　はて、儂が昔試してみた囲碁ソフトはあまりに弱すぎて話にならなかったが……。最近はお主の相手になるくらい強いソフトが出ているのかね? 」
「そうでゲソ。ここ数年で囲碁ソフトは急激に進歩したでゲソ。 最近は私より強くて、トッププロに 4 子で勝ったりしているでゲソ！」
「ほう、そうなのか。面白そうじゃの。儂も一つ買ってみるか。」
　さて、最近強くなったという囲碁ソフトについて、ここでλカ娘に解説してもらうことにしよう。旧来の囲碁ソフトとの比較を交えながら、強くなった理由を語ってもらおう。

30.2 囲碁のルールおさらい

「囲碁のルールを説明するでゲソ。囲碁のルールは細かいものまで説明していると結構大変なので、今回の記事に関係ある部分だけ説明するでゲソ！　以下の 4 つを把握しておけば十分でゲソ。」

1. 黒白交互に盤の交点に打つ。1 手目は黒から (図 30.1)。[*1]
2. 相手の石の四方を囲むと取れる (図 30.2)。
3. 自分から石を取られるような手は打てない (自殺手の禁止) が、それ以外は空いている場所のほぼどこでも打てる。そのため、自由度が非常に大きい。ただし一見自殺手に見えても、先に周りの相手の石を取れる場合は打てる (図 30.3)(『コウ』という概念があって、厳密には打てない場所もあるが、今回は説明しない)。
4. 相手より大きな陣地を囲えば勝ち (図 30.4)。陣地とはそれぞれの石または盤端で四方を囲まれた石の無い領域のこと。図は数えやすいように 9 路盤 (練習用の小さな盤) で示している。黒と白の丸は実際の石で、四角はそれぞれの陣地を示す。四角の所には実際には何も置かれていない。この場合、黒の陣地は 36 目 (目(もく)とは地の単位)、白の陣地は 23 目で黒の 13 目勝ち。λカ娘が勝った対局は、 1 目だけλカ娘の陣地が大きかったということ。

[*1] ただし置き碁 (ハンデ戦) の場合は黒があらかじめ石を何個か置き、その後白から打ち始める。λカ娘が勝った対局は、黒のλカ娘が先に 3 個石を置いていた。

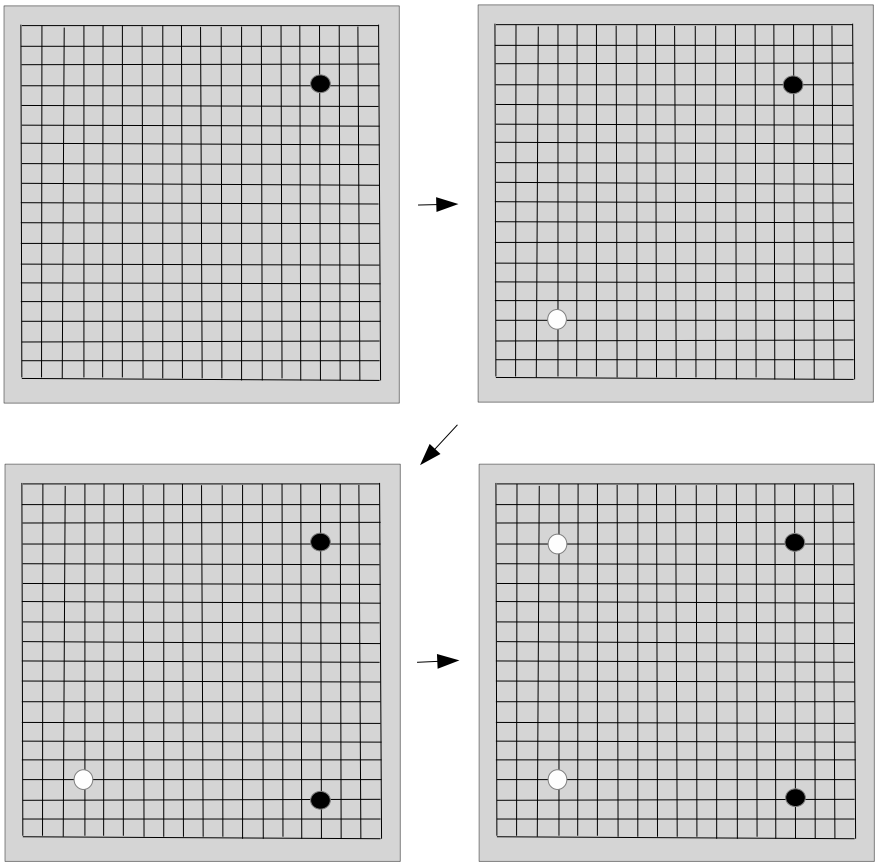


図 30.1: 交互に打つ

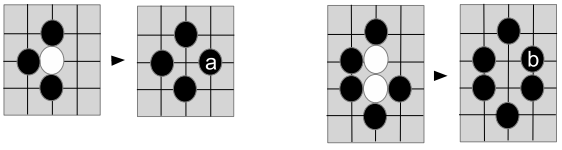


図 30.2: 左図の状態から、a で示した石を黒が打つと、真ん中の白石は上下左右囲まれるので取られる。b のように、複数石あっても同様である。

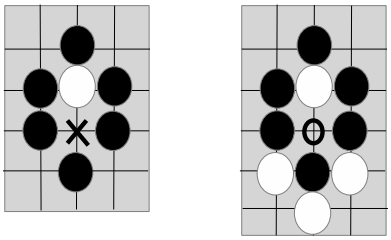


図 30.3: 白は×には打てないが○には打てる

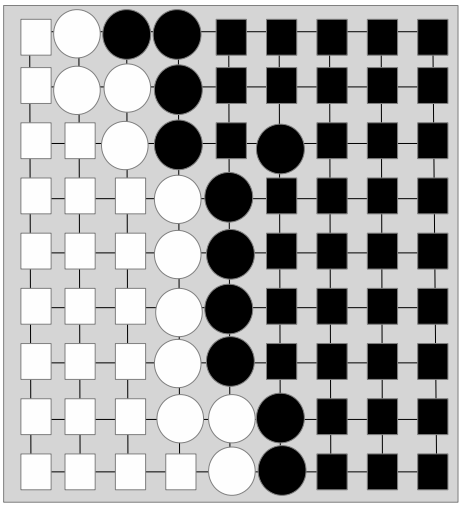


図 30.4: ■が黒の陣地で 36 目。□が白の陣地で 23 目。黒 13 目勝ち

30.3 AND-OR 探索

「ゲームの AI において人間の『読み』に相当する機能は基本的に『AND-OR 木』というものがベースになっているでゲソ。対戦型のゲーム (厳密に言うと『二人零和有限確定完全情報ゲーム』と呼ばれる種類のもの) ならばどんなものを扱ってもいいけど、ここでは盤面が小さな Tic-Tac-Toe で説明するでゲソ。」
「聞いたことないゲームじゃな。」
「日本では三目並べとか○×とか呼ばれているでゲソ。とりあえず図 30.5 を見るでゲソ。爺さんもやったことあるんじゃなイカ？」
「ああ、これか。これなら子供の頃やったことあるわい。」
「図 30.5 は最大あと 4 手で決着する、Tic-Tac-Toe のある局面 (a) からの変化を網羅したものでゲソ。(a) での手番は×でゲソ。×は次にどこに打てるでゲソか ?」

「上、左、左下、下の 4 マスじゃな。」
「その通りでゲソ！ その 4 つの可能性があるでゲソ。人間がこのゲームをする時には、それぞれの箇所に打ったらその後何が起こるか考えるんじゃなイカ？ 例えば、上のマスに打ったとするでゲソ。そうすると、局面は (b) のようになるでゲソ (上の局面から下の局面への線は、この一手打ったことによる変化を示しているでゲソ)。

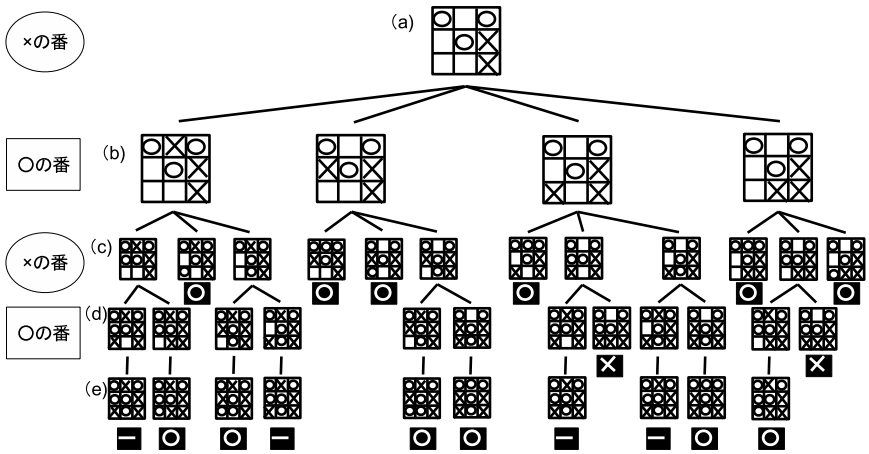


図 30.5: Tic-Tac-Toe の変化を網羅したゲーム木。ノードの下にあるマークは、どちらが勝ったかを表す。「-」は引き分けを表す。

局面 (b) では相手、○の手番でゲソ。この局面 (b) ではまだ決着がついていないのでゲームを続行するでゲソ。○は左、左下、下の三か所に打てるでゲソ。
○がもし左のマスに打ったとしたら……次は局面 (c) になるでゲソ。×がその次に左下に打つと局面 (d) に、そうすると次に○は下にしか打てなくて、下に打つと局面 (e) になって、引き分けになるでゲソ。これが一つの可能性じゃなイカ！
もちろん、可能性を一つ考えるだけではダメでゲソ。例えば、局面 (b) で○は左ではなく左下に打つこともできるでゲソ。そうすると……○の勝ちでゲソ!つまり私の負けじゃなイカ！ 局面 (b) は○の手番だから○の好きな場所に打つことが出来るので、当然一番○にとって良い、つまり×にとって一番悪い手、『左下』を打つでゲソ！ この可能性を見逃して最初の局面 (a) で×を上に打ってしまうと、次に○に左下を打たれて負けてしまうでゲソ。
だから、局面 (a) では上、左、左下、下のすべての可能性を、その次の局面 (b) でも左、左下、下のすべての可能性を……という風に、すべての可能性を網羅したものが図 30.5 でゲソ！ 」
「なるほどのう。でも、これでは全て列挙してみただけで、勝ちか負けか、どこに打てばいいのか分からんのう。」
「そこで『AND-OR 木』でゲソ！ 」
「この網羅図と論理式の AND-OR が関係するのか。難しいのう。」
「実際はそんなに難しくないでゲソ！ 図 30.6 を見るでゲソ！ これは、さっきの網羅図に対応する『AND-OR 木』でゲソ！
AND-OR 木を作る目的は、『自分が勝てる手』を探すことでゲソ (見つかったら当然それを打ちたいからでゲソ)。単純化すると、ゲームには『双方最善を尽くした時自分が勝てる局面』と『双方最善を尽くした時自分が負ける局面』しかないので、これを仮に『True(真)』と『False(偽)』で表すことにするでゲソ。引き分けがあるゲームについては、引き分けをどうするかは難しい問題でゲソが、今回は引き分けは自分の勝ち True とするでゲソ！
図 30.6 の場合は、自分が×だとして、×が勝てる局面 =True にたどり着けるような手を探すでゲソ。

自分の手番の時 X は、次の一手で True になるような局面が一つでもあれば、他の手の結果がどうなろうと関係なく、True になる手を選べるので、X も True になるでゲソ。どの手を選んでも False になる場合は X も False になるでゲソ。これは要するに次の局面の集合に対する『OR』演算と同じでゲソ！
相手の手番の時 Y は、逆に、次の一手で False になるような局面が一つでもあれば、他の手の結果がどうなろうと関係なく、False になる手を選んでくるので、Y も False になるでゲソ。どの手を選んでも True になる場合は Y も True になるでゲソ。これは要するに次の局面の集合に対する『AND』演算と同じでゲソ！
そうやって、図 30.7 のように、下から順に、つまり確定しているところから順に、各局面の勝敗を判定していくことが出来るでゲソ！ こうすると、今の局面で勝ちか負けか分かる上、勝ち (True) の場合は子ノードから True になっている奴選ぶのが最善手であることが分かるでゲソ！ 自分が負け (False) の場合は子ノードはすべて False なのでどこ打っても負けになるでゲソ…」
「なるほどのう。ところで、全く同じ局面が何度も登場しているようじゃが、それを全て調べるのは無駄じゃないのかね？ 例えば、引き分けになっている局面は全て同じじゃな。まあこれは AND-OR 木の末端じゃから、調べなおしても大した負荷にはならんかもしれんが、探索の途中にも同じ局面があるのう。これはかなりの負荷になる場合もあるのではないかね？」
「おっしゃる通りでゲソ！ 実は、いくつかのゲームではゲーム木中で全く同じ局面に到達する場合があるでゲソ。そのような場合、何度も探索し直すのは無駄なコストがかかるので、『置換表』というものに一度探索した局面とその結果を記録しておくのが普通でゲソ。」

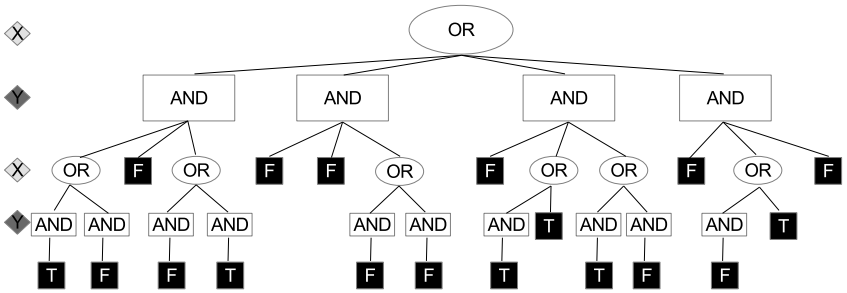


図 30.6: 図 30.5 のゲーム木を「AND-OR 木」で表したもの

30.4 Min-Max 探索

「上記の AND-OR 探索では、ゲームの決着がつくまで探索できることを想定していたでゲソ。でも、現実に楽しまれているゲームは、選択肢の幅が広すぎて、とてもゲームの決着がつくまで探索は出来ないことが多いでゲソ。あるいは、例えばリバーシならスコアは-64〜64 の整数になるでゲソが、勝つか負けるかだけではなく決着時に出来るだけ大差で勝ちたい/負けるにしても小差にしたいという場合もあるんじゃなイカ？ そのような場合は、『評価関数』と『Min-Max 探索』を組み合わせて使うでゲソ。順に説明するでゲソ！
評価関数はゲームの途中でどちらがどれだけ有利かを計算する関数で、自分の手番が有利なほど大きな値 (通常は正の値) を、相手の手番が有利なほど小さな値 (通常は負の値) を返すでゲソよ。例えば、絶対負け = −10000、絶対勝ち=10000、完全に互角=0、やや有利=1000 のようになるでゲソ。何せゲームの途中で判断するので、評価関数を作るのはなかなかに難しく、それだけで本何冊も書けそうでゲソ。でもここでは、人間のプレイヤーがゲームの途中で『勝ちそう！ 』とか『負けそう…』とかを判断しているのをプログラムにやらせるというイメージだけあればいいでゲソ！ これを、AND-OR 木で最後まで読み切った時の True/False

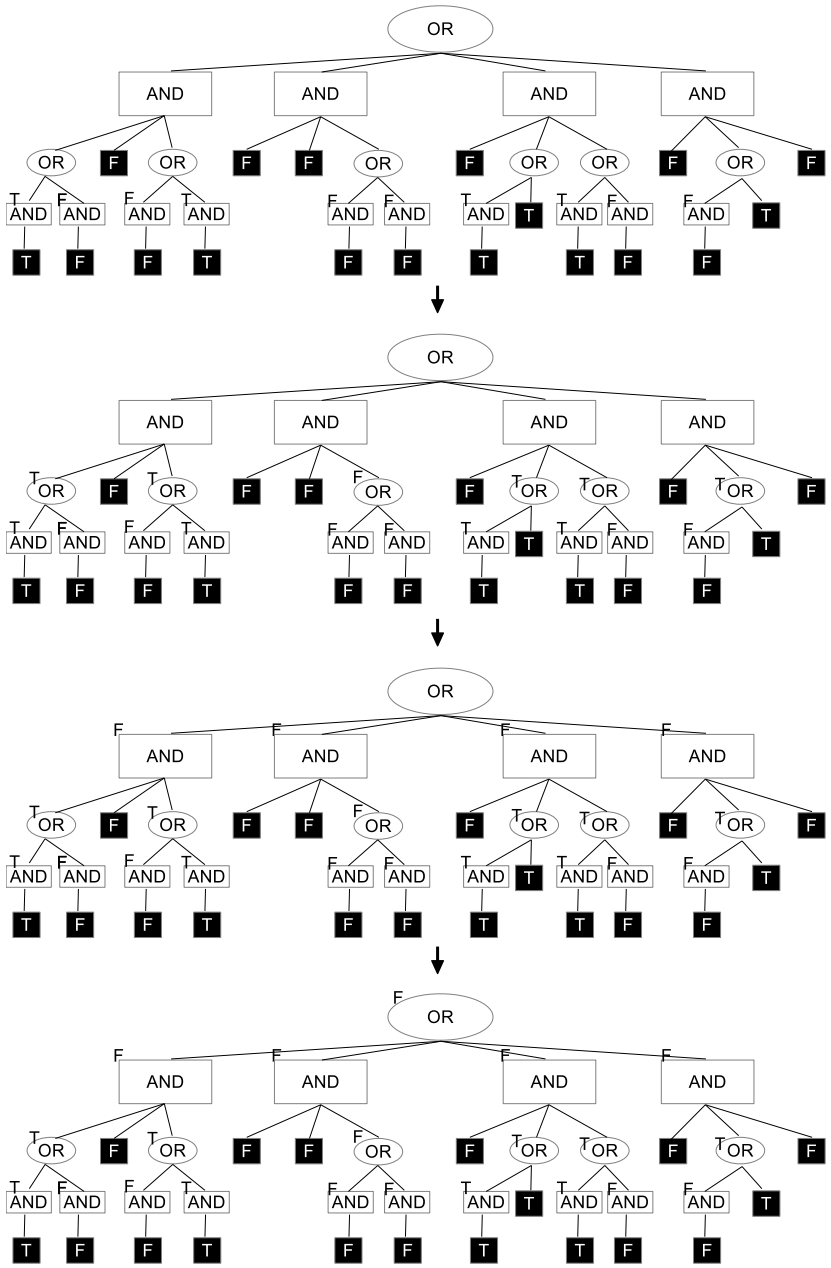


図 30.7: AND-OR 木のルート局面の勝敗が決まる様子

だった部分の代わりに使うでゲソ。
そして、さっきは True と False だったので、AND-OR で良かったでゲソが、評価関数を使うともう AND-OR では計算できないでゲソ。そこで Min-Max でゲソ！ Min-Max 探索は、先ほどの『OR ノード』が『Max ノード』に、『AND ノード』が『Min ノード』になったものでゲソ。
自分の手番の時 X は、次の一手で好きな手を選べるので、次の局面の中から出来るだけ大きな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Max』演算と同じでゲソ！
相手の手番の時 Y は、逆に、次の一手で相手が好きな手を選べるので、次の局面の中から出来るだけ小さな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Min』演算と同じでゲソ！
図 30.8 が、そうやって下から順に評価値が決まっていく様子を示したものでゲソ！ これで、現在の局面の評価値、つまりどれくらい有利/不利そうか、が分かる上、Max ノードなので子ノードのうち一番評価値が大きかった子ノードに対応する手を打つのが最善手であるというのも分かるでゲソ！ 」

30.5 α-β 探索

「実は、Min-Max 探索と同じ結果を保証しつつ、Min-Max 探索よりも大幅に少ない探索量で済む探索手法があるでゲソ。それが α-β 探索でゲソ！ どれくらい探索量が減るかというと、元の探索量の平方根くらいになるでゲソ。図 30.9 を使って説明するでゲソ。
(現在は並列化などのために厳密にそうとは言えないが) 通常は図の左にあるノードから評価関数が呼ばれるでゲソ。

- (a)start でゲソ。
- (b) 一番左の Min ノード (i) までの探索が終了した所でゲソ。ここで『よし、この一番左の手を選べば最低でも +7 で勝てる。でも、もっといい手はないのかな? 他の二つの可能性も調べてみて、+7 よりも私に有利な手ならそっちにしよう』というわけで次の探索に行くでゲソ。
- (c) というわけで二番目の手の探索を開始し、Max ノード (j) までの探索が終了した所でゲソ。
- (d)『ふむ、二番目の手を打つと、相手には評価値 +5 という手が発生する。相手はなるべく評価値の低い手を打ってくる (Min) から、二番目の手を打ったときのノード (k) の評価値が +5 より大きいということはありえない。さらに悪いことにもっと評価値が低い手、マイナスの手がある可能性もある。しかし、冷静に考えると、(b) で +7 より有利な手を探していたんだから、(k) が +5 以下であることが分かっただけで十分だな。これ以上 (k) について調べる必要はない。一部のノードは見てすらいないのだけど、三番目の手の探索に移れるな。』

という感じでゲソ！ 」
「ふむ、なるほどのう。少し分かりづらいが、考えなくて良い局面があるわけじゃな。では、囲碁プログラムもこの『あるふぁべーた探索』を使って作られているのかね？」
「昔の囲碁プログラムはそうだったでゲソ。でも、すごく弱いのしか作れなかったでゲソ！ その理由を説明するでゲソ。」

30.6 α-β 探索が囲碁ではうまく機能しなかった理由

「α-β 探索が囲碁でうまく機能しなかった理由は大きく二つあるでゲソ！」「一つは囲碁では良い評価関数が作れないからでゲソ。すごくざっくり言うと、リバーシでは石のパターンを学習することで良い評価関数を作ることが出来、チェスや将棋では駒や駒組みの評価値を学習することで良い評価関数を作ることが出来たでゲソ。しかし、囲碁は、一つ一つは個性の無い石が無限とも言える組み合わさり方で様々な価値を生み出すので、良い評価関数を作ることが非常に難しいでゲソ！ 人間の強い人がどのようにして形勢を判断しているかを言葉で説明しても、最低でもアマチュア初段くらいの力のある人でないと理解できないくらい、囲碁の形勢評価は難しいでゲソ!」
「確かにそうじゃのう。儂も良くアマチュアのあまり強くない人を指導するのじゃが、形勢判断の仕方を説明するのはすごく難しく感じるわい。」
「もう一つは、囲碁では打てる手の数が多すぎて、深く読めないからでゲソ。囲碁はルール上打てる手が、9 路盤 (練習用の小さい盤) の最初の状態で 81 通り、19 路盤 (公式の盤) の最初の状態で 361 通りあり、中盤付

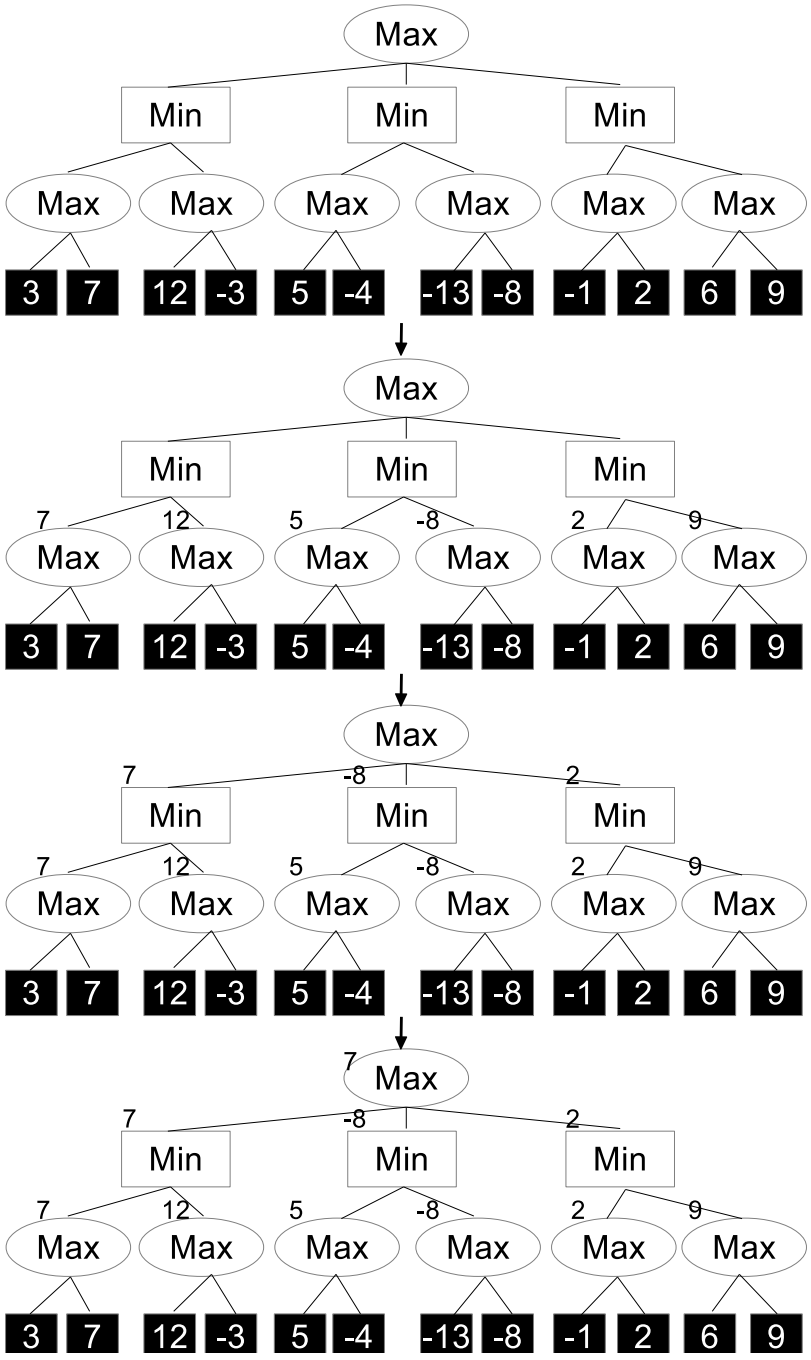


図 30.8: Min-Max でルート局面の評価値が決まる様子

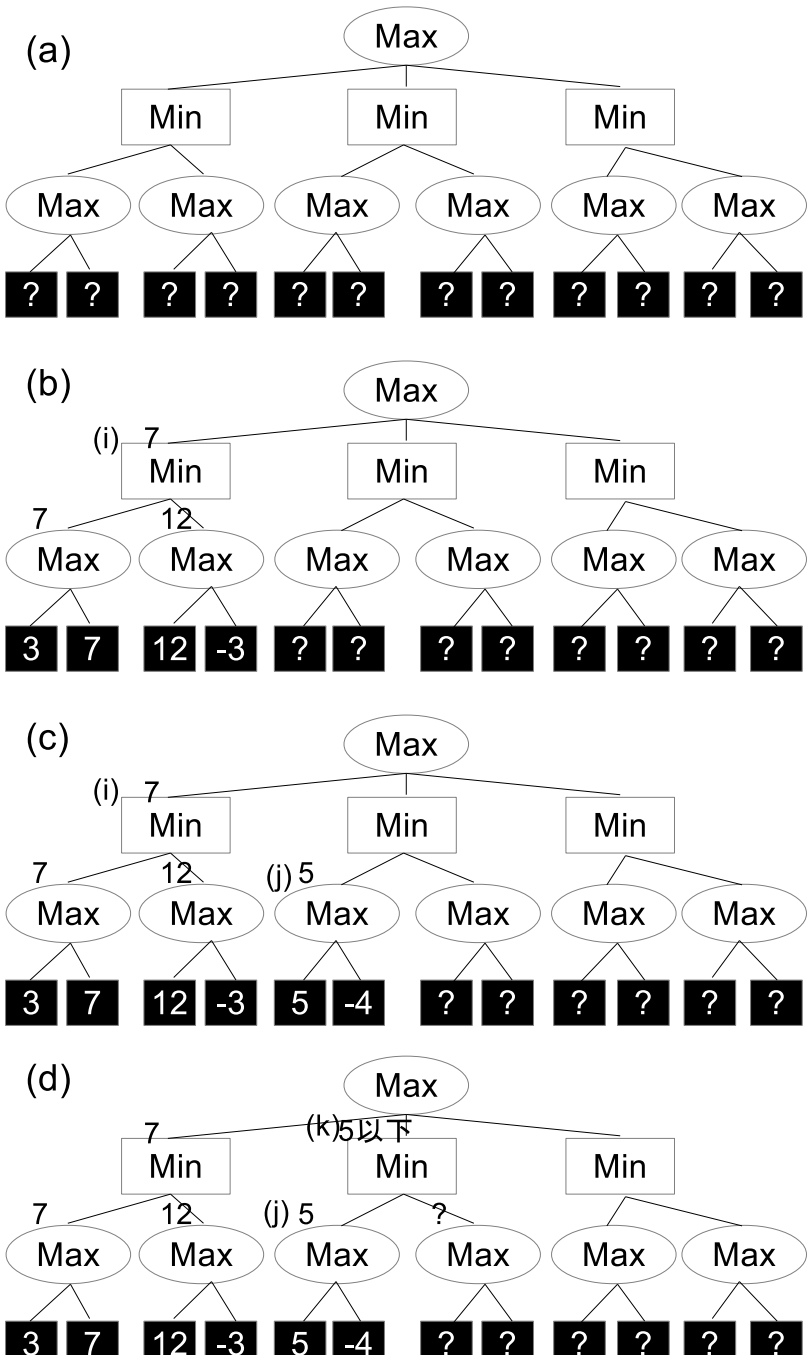


図 30.9: α-β 探索の様子

近でも 9 路盤で 60 通りくらい、19 路盤で 250 通りくらいもあるでゲソ！　ちなみに、リバーシだと石のひっくり返せる所しか打てないので、打てる場所は 10 通りくらいのことが多いでゲソ。もちろん手を絞り込む研究もされているでゲソが、少々絞り込んだだけでは焼け石に水でゲソ！　また、絞り込むと重要な手を見落とすでゲソ！」

「囲碁は、有段者になると、手を瞬間的に数か所に絞り込めると思うのじゃが、コンピュータにとって、それは難しいということじゃな。では、囲碁プログラムは一体どうやって強くなったのかね？　」

「それを今から説明するでゲソ！　」

30.7　原始モンテカルロ法

「『モンテカルロ法』という名前くらいは聞いたことがあるでゲソか？　乱数を使って近似値を求める方法でゲソ！　囲碁におけるモンテカルロ法、厳密に言うと原始モンテカルロ法とは、『次の一手として有力な手がどれかを探すために、まず候補手を打ってみて、その後無作為に最後まで、自殺手以外打つ場所が無くなるまで、打つということを何回も繰り返し、結果の平均、通常は何目勝ったかではなく何回勝ったか、すなわち勝率をその候補手の点数として、点数が最大の候補手を次の一手として選ぶ』手法でゲソ。なお、『無作為に最後まで打つ』シミュレーションのことを『プレイアウト』と呼ぶでゲソ。図 30.11 を見ると分かると思うでゲソ。この図の場合は、一個の候補手あたりのプレイアウトが 100 回ということでゲソね。単純に考えると候補手が 74 箇所あるでゲソから合計で 7400 回になるでゲソ。実際には、プログラムにもよるでゲソが、合計 1 万〜100 万回くらいプレイアウトするでゲソ！　一見すごく多そうでゲソが、αβで全部の手を深いところまで調べるのに比べたらはるかに少ないでゲソ！　」

「なんと！　そんな無茶苦茶な方法で強いソフトが作れるのかね！？　」

「モンテカルロ法は 1990 年代には提案されていたでゲソが、当初は弱いソフトしか作れなくて、皆もこんな方法で強いソフトが作れるわけがないと思っていたでゲソ！　でも、2006 年頃に再び注目され、多くの工夫が加えられたことで、強くなったでゲソ！　それの一部を次の節で説明するでゲソ。」

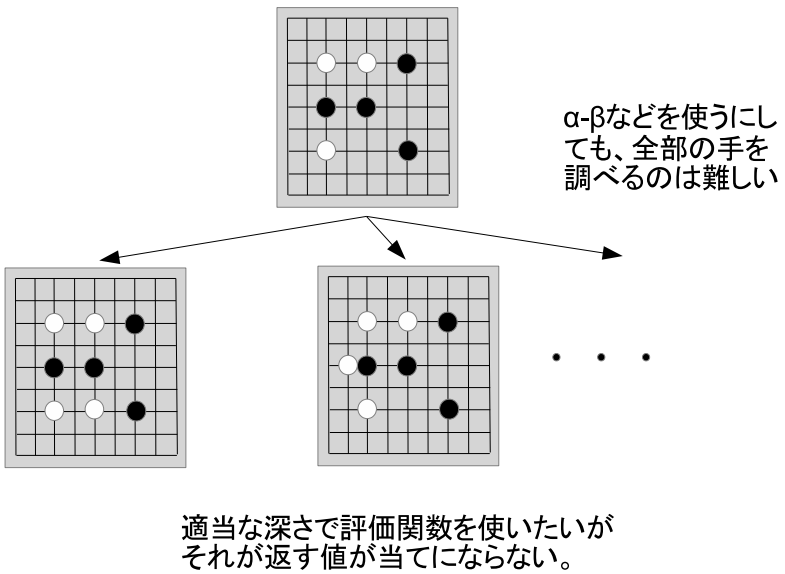


図 30.10: 囲碁で α-β は上手く機能しない

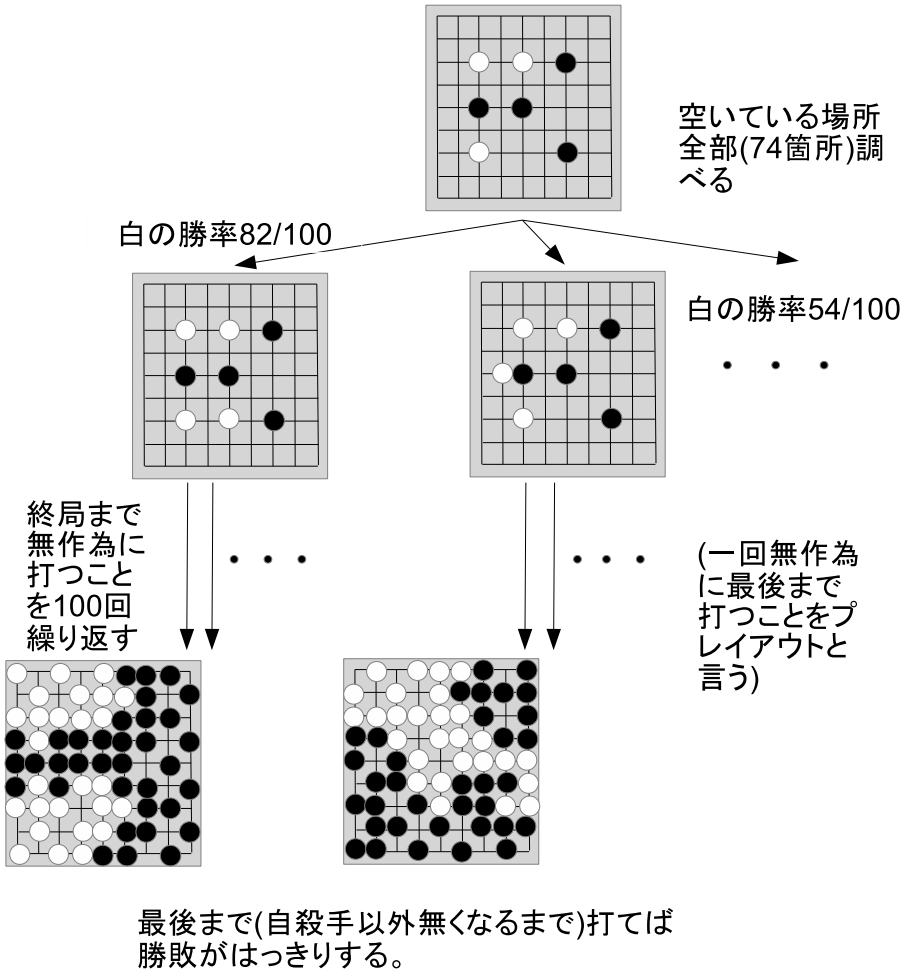


図 30.11: 囲碁における原始モンテカルロ法。

30.8 モンテカルロ木探索

「原始モンテカルロ法だけでも弱い囲碁プログラムが出来るでゲソが、原始モンテカルロ法に木探索を組み合わせるともっと強くなるんじゃなイカ？ 図 30.12 を使って、モンテカルロ法における木探索を説明するでゲソ。まずは普通に、各候補手に対して何度かプレイアウトを行って結果を集計するでゲソ。それから、ある候補手に対してのプレイアウト数が閾値 (1〜数百のことが多い) を超えたら、そこにノードを作るでゲソ (それまではプレイアウト数を集計するための仮のノードを作っておく)。そうすると、深さ 1 のところにノードがたくさん出来るでゲソ。そうなったら、以降のプレイアウトを行う際に、そのノードの勝率とプレイアウト数に注目するでゲソ。そして、『勝率が高く、まだプレイアウト数が少ない』ノード (候補手) を選

んで、そこからまたプレイアウトをするでゲソ。これを何段も繰り返すと、モンテカルロとして有望なところは深いところまでノードが作られるでゲソ！　そして、上記の $\alpha-\beta$ 探索と同様の効果が得られるようになるでゲソ！　これがモンテカルロ木探索でゲソ。」

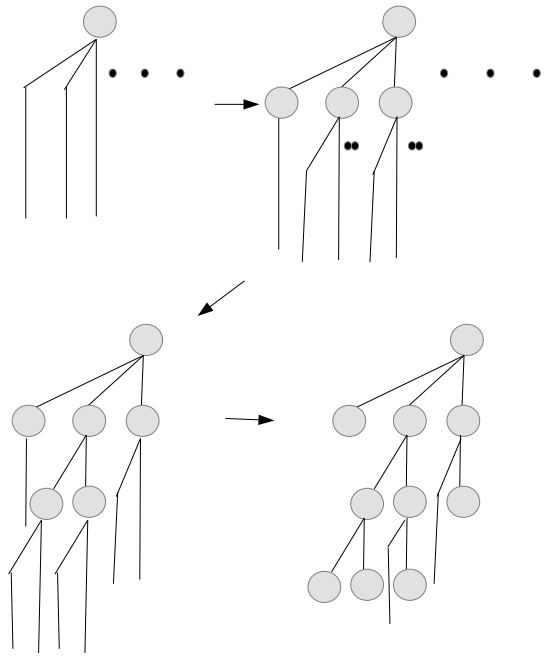


図 30.12: モンテカルロ木探索のイメージ。最初はルートノードしかないが、プレイアウトを繰り返すにしたがって、ノードが出来る。ノードが出来たら、有望そうな (勝率が高く、プレイアウト回数がまだ少ない) ノードを中心にプレイアウトすることになる。その結果、有望そうなところだけ、どんどんノードが深く作られ、木となる。

30.9　モンテカルロ木探索を用いた囲碁ソフトの現状

「モンテカルロ木探索を用いることで、囲碁ソフトは飛躍的に強くなったでゲソ！　当初は 9 路盤でしか有効でないと思っていた人もいるようでゲソが、たくさんの工夫によって、19 路盤でもかなり強くなったでゲソ。現在最強の囲碁ソフトは『Zen』と『CrazyStone』だと言われているでゲソが、どちらもネット碁の KGS というところで 5d〜6d[*2] くらいにランクされているでゲソ。これは日本基準だとアマチュアの 7〜8 段くらいでゲソ。Zen は 4 子置いて (Zen 側が 4 段級差有利) トッププロの武宮プロに勝ったことがあるでゲソ。CrazyStone も 4 子置いてトッププロの石田プロに勝ったことがあるでゲソ！　また、Zen は 3 子置いて高校チャンピオンの大表さんに勝ったこともあるでゲソ。」

「そんなに強いのかね！　儂もトッププロと雑誌の企画で何度か打ったことがあるが、3 子置いただけでは勝ったことないし、4 子置いてようやく五分五分という感じじゃったな。」

「でも、アマチュアトップレベルの多賀さんとハンデなしで打ってみたこともあるでゲソが、その時はボコボコにされていたでゲソ……。図 30.13 がそのボコボコにされた棋譜でゲソ。黒が CrazyStone でゲソが、なんと上辺の攻め合いを CrazyStone が理解出来ていなくて、図の局面でも CrazyStone は自分が優勢と思っていたでゲソ！　でも、実際には 100 目以上の大差で多賀さんが優勢でゲソ！　ここから碁の神様が CrazyStone の代わりに打ったとしても、逆転は不可能でゲソ！　仕方なくオペレーターが投了したでゲソ……。このように、現在の囲碁ソフトは強いでゲソが、攻め合いのような細く長い読みを必要とする局面が弱点なんじゃ

[*2] KGS では対戦成績によって段級位がつく。5d というのは 5 段ということ。ただし、日本のアマチュアの 5 段よりも評価が厳しいと言われている。なお、日本と同様に級もあり、例えば 1 級なら 1k となる。

なイカ？　これを解決するために世界中の研究者が研究しているでゲソが、未だ有効な解決策は見つかっていないでゲソ。もちろんこの文章の著者も研究しているでゲソ！」
「なるほどのう。確かに『適当に最後まで打って』形勢判断するという仕組みだと、一本道で長い手順は見落としてしまうわけじゃな。」
「ちなみに、モンテカルロ将棋も研究されているでゲソ。でも、まだまだ α-β ベースのプログラムには勝てないようでゲソ。なぜ囲碁でモンテカルロ法が上手く働くのか、理由は大きく二つあるでゲソ。まず一つは、囲碁は一手の手の価値の差が小さいから、ランダムシミュレーションの結果の平均が形勢の良い近似になりやすいからでゲソ。もう一つは、囲碁はランダムに打ってもほぼ間違いなく終局するゲームだからでゲソ。」
「言われてみれば確かにそうじゃな。将棋だとそもそも適当に打っていたら詰まないし、詰みを見落とすか見落とさないかだけで結果が逆転してしまうからの。」

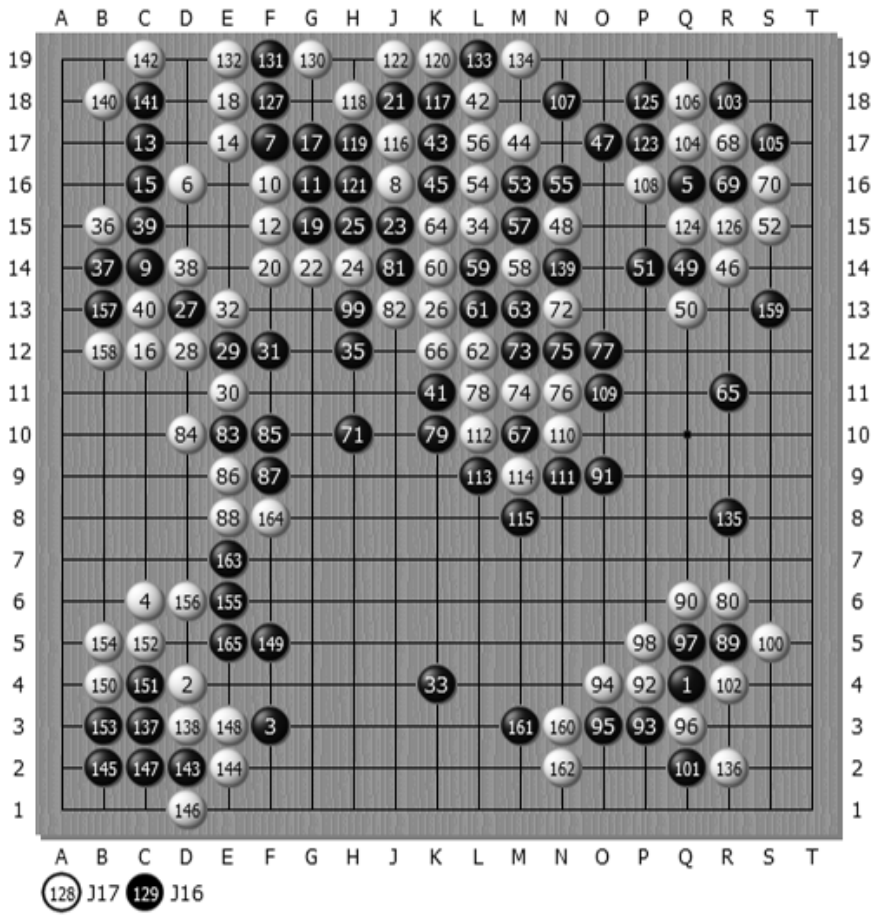


図 30.13: 囲碁ソフト、トップアマに完敗!

30.10　エピローグ

「いやはや囲碁ソフト試してみたぞい。最初は試しにお主と同様 3 子置かせて (3 段級差ソフト有利) みたのじゃが、コロッと負かされてしまったワイ。ただ、何局か打っているうちに弱点が分かってしまって、最近は 5 子 (5 段級差ソフト有利) でも勝てるようになったワイ。」
「5 子でゲソか……。相変わらずすごいでゲソね……。爺さんプロより強いんじゃなイカ？」
「いやいや、プロが試さないような変な手を試しているうちに弱点が分かったというだけじゃよ。しかし、やはりお主のように、人間と打っている方が楽しいの。」
「(私は人間ではなくλカ娘でゲソが、つっこまないでおくでゲソ)。じゃあ今日も打つでゲソ!」
　λカ娘の挑戦は終わらない……。

30.11　参考文献

- コンピュータ囲碁-モンテカルロ法の理論と実践-(松原仁　編集、美添一樹、山下宏共著 共立出版)
- ゲーム計算メカニズム (小谷善行 編著、岸本章宏、柴原一友、鈴木豪 共著　コロナ社)
- Combining Online and Offline Knowledge in UCT(Sylvain Gelly, David Silver, Proceedings of the 24th International Conference on Machine Learning)
- Computing Elo Ratings of Move Patterns in the Game of Go(Rémi Coulom, ICGA Computer Games Workshop, Amsterdam, The Netherlands, June 2007)

第31章

僕のカノジョはスナッチャー

— @master_q

31.1 川のほとり

『下におろして、垂直にまっすぐ持ち上げて敵を打ち抜くんでゲソ。ここ、両脇が大事なんでゲソ。そーして、打つべし、打つべし、打つべし、ぐるーんでゲソー!』
「なんなの、それ？」
『スナッチを制する者は、世界を制すでゲソ。最初のターゲットを決めるのが難しいんでゲソ』
「またサバゲーの話？」
『ん？ 何やってんでゲソ？』
「型推論の宿題」
『なんで大学でやんないんでゲソ？』
「かっこう悪いから」
『ワシのもやってくれなイカー』
「Dvorak 配列のままだよ」
『ぬ？ おぬしもスナッチやればいいじゃなイカ。なんでいつも ThinkPad 持ってるんでゲソ？』
「自分こそ、なんでいつもここに来るんだよ。」
『それはでゲソね、…あれ、なんででゲソ？』
「お気楽だね。イカ墨ください」
『出してないでゲソ』
「あのさ、なんでいつもこんなこと…」
『人間の指はおもしろいでゲソー』
「ん？」
『ワシはこうしてないと溢れちゃうじゃなイカ』
「溢れちゃうって、どうなっちゃうんだ？」
『……きっと…すごいことになるでゲソ……』

すごいことなんてない。ただ当たり前のことしか起こらない。僕達の大学から見えるオンボロなビル、ソフトハウス–メタセピのオフィスができたとき、プログラマたちは大騒ぎだった。毎日決まった時刻に鳴り出すスナッチのメロディーが、僕にはなんだか不吉なサイレンのように聞こえる。それはうっすらと広がり、世界を覆っていく。

31.2 カノジョのこと

僕のカノジョはスナッチャーだ。スナッチャーというのは他の生物の一部を置換して自分の体内に取り込むエイリアンの名前だ。いつから彼等がこの世界に現われたのかはわかっていない。とにかく現在では僕たち人間とスナッチャーが同居して暮していた。

僕が大学で授業を受けていると、カノジョが突然声をかけてきた。どうも授業の宿題が解けないらしい。僕は他人と会話するのが苦手なのでおっくうだったが、簡単に鍵となるアイデアを伝えた。次の日もその次の日もカノジョは僕に声をかけてきた。そのうち、僕達は毎週いっしょにハイキングにいくようになった。毎週どこにハイキングに行くのか、実のところそのネタを考えるのも僕にはおっくうだった。紙と鉛筆がそ

れまで僕の友達だったから。

カノジョはスナッチャーなので、ときおり僕の一部をスナッチしたくなるようだ。どこが好かれているのかわからないが、カノジョは僕のことをお気に入りだ。もちろん僕だってスナッチされるのは御免だ。カノジョだって僕を完全にスナッチしてカノジョ自身の中に飲み込んでしまったら、僕という存在がなくなってしまう。一緒にハイキングに行く相手がいなくなる。そんなのつまらないと思っているんだろう。今のところ、カノジョは僕のことを本気でスナッチしてこようとは思わないようだ。

カノジョのお気に入りのタブレットは Nexus 7 だ。このタブレットは Android OS で動いていて、Android NDK を使えば C 言語だけでアプリケーションを書くこともできる。カノジョはスピード狂だったから、Android のアプリはいつも C 言語で書いていた。でも僕は C 言語よりも Haskell の方が気に入っている。型の強い言語というのは設計時の安心感が段違いだからだ。せっかくなので、Haskell で面白いアプリをカノジョのために書いてあげたいと最近は思うようになった。

31.3 Snatch-driven development

ところが Android NDK を使っても Haskell で Android アプリケーションを書くのは簡単ではない。たしかに Android NDK を使えば POSIX API を使った C 言語プログラミングができる。そのため GHC を使ったクロス開発が可能だ。しかし GHC だとバイナリサイズが巨大になる傾向がある。簡単なロジックでも数十 MB ものコードサイズになってしまうこともある。また、Android はアクティビティという少しかわったしくみでプログラムの状態管理をしている。このようなドメイン固有の作法を守りつつゼロからアプリケーションを設計することは簡単ではないのだ。もし C 言語のまっとうな実装があれば、そのコードの設計を推測して Haskell で書き直すこともできる。しかしプログラムの規模が大きくなるにつれてこの作業は難しくなる。というのも”C 言語のコードから設計を推測する” ことが完全に完了しないと” 設計を Haskell で書き下す” ことができないからだ。当然この作業が完全に完了しないと動作確認はできない。小さなユニットテストはできるが、そのユニットテストが設計から正しく意図されたものであるかも自信が持てない。この流れを少しずつ進行させる方法はないだろうか？

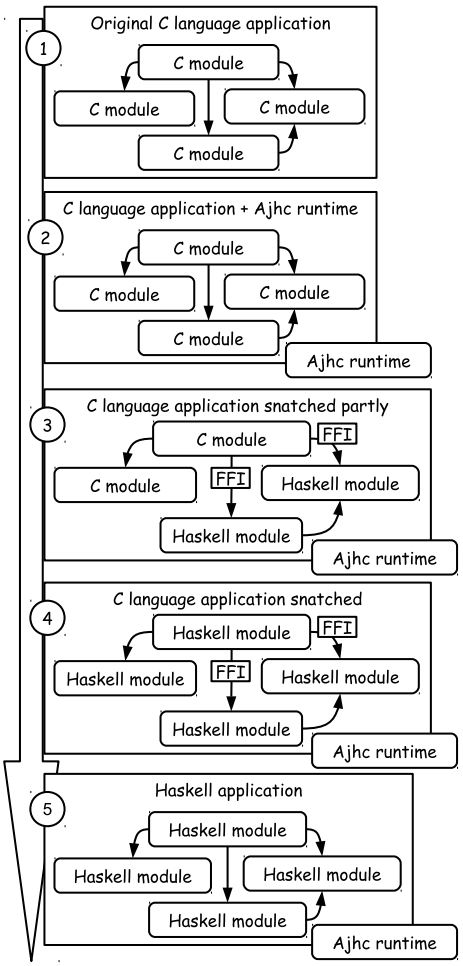


図 31.1: Snatch-driven development の概要

ところで、ソフトウェアには “Snatch-driven development” という開発手法がある。Ajhc という Haskell コンパイラを組み合わせて使えば、C 言語の設計を動作可能な状態をたもったまま Haskell で書き直すことが可能だ。また Ajhc には大変小さなコードサイズのバイナリを吐くという特性もある。以前からこの手法に興味があった。カノジョへプレゼントする Android アプリを作るのに、C 言語で書かれた簡単なアプリケーションを Haskell でスナッチした後、その Haskell の設計を成長させるような方法を取ることはできないだろうか？ Android アプリを C 言語から Haskell で書き直すことは、スナッチ設計を身に付ける上で良いトレーニングになるだろう。

スナッチのやり方を簡単に解説しておこう (図 31.1)。スナッチは多くの場合以下のステップを取る。下のステップの番号はそのまま図 31.1 の番号にそのまま対応している。

Step 1 まず既存の動作可能な C 言語のソースコードを用意する。この C 言語ソースコードは実行可能、もしくはテスト可能であればなんでも良い。

Step 2 その C 言語ソースコードにとりあえず Ajhc コンパイラのランタイム部分だけを埋め込む。この状態でまず実行/テストしてみて問題ないことを確認する。

Step 3 その後、プログラム内のモジュールの内 Haskell 化できそうな箇所を選んで設計置換をする。この時、Haskell と C 言語の間は FFI を使って接続する。つまり Haskell の型を使わず C 言語の ABI でモジュール間を接続する。

Step 4 この設計置換をくりかえしてプログラム全体を Haskell 化する。

Step 5 全体を Haskell 化しおわった状態で動くことを確認できたら、最後に FFI を取り去って Haskell module 同士を型によるインターフェイスで接続する。

これで元の C 言語の設計がそのままの形をたもったまま Haskell 化できたことになる。

31.4 Android NDK とそのサンプルアプリケーション

さっそく Android アプリスナッチの準備をはじめよう。何を用意すればいいだろうか？

- Android NDK 開発環境とその概要理解
- スナッチする Android NDK サンプルソースコード

こんなところだろうか。順番に手をつけていこう。

31.4.1 Android NDK 開発環境

Android NDK [*1] というのは先にも言った通り Android のアプリケーションを C 言語で書くための開発環境だ。詳細ははぶくが Android は通常の UNIX ライク OS と異なり、プロセスではなくプロセスの上に構築されたアクティビティというしくみでアプリケーションを管理している。通常はこのアクティビティは Java 言語で設計することになっている。特殊なライフサイクルがあるので生の C 言語では扱いにくいのだ。ところが NativeActivity というしくみを使うと、このアクティビティを C 言語のみで設計することが可能になる。しかし Android フレームワークから渡されるイベントにすばやく応答するために、Android NDK プログラミングでは POSIX スレッドを使う必要がある。もちろん誰もが知っている通り、POSIX スレッドを使ったプログラミングは大変難しい。デッドロックを防止するために非常に注意深い設計が求められることなどがその理由だ。

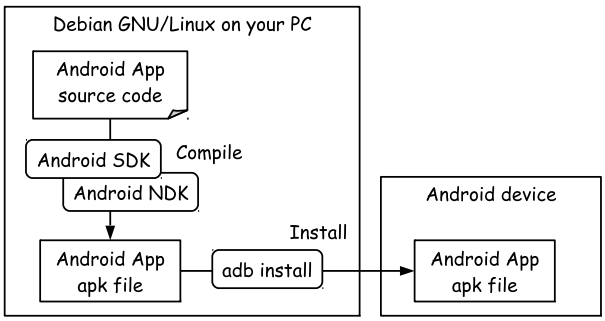


図 31.2: Android NDK アプリのクロス開発環境

そこで、Android NDK ではこの POSIX スレッドを使ったアクティビティ制御のラッパーとして android_native_app_glue を提供している。android_native_app_glue はアクティビティのライフサイクルを C 言語でも書きやすくしてくれるミドルウェアのようなものだ。android_native_app_glue を使えば設計者は POSIX スレッドを意識することなく NativeActivity アプリケーションを設計できる。

Android NDK 開発環境の設定に入る前に、開発環境のおおざっぱなイメージをつかんでおこう (図 31.2)。通常のアプリケーションとは違い、Android 上のアプリケーションは Android OS 上ではなく別の PC 上でコンパイルする。ぼくは Debian GNU/Linux を使っているので、Intel アーキティクチャ上の Debian に Android SDK と Android NDK をインストールして、ARM アーキティクチャの Android OS 向けのアプリケーションパッケージである apk ファイルをビルドできる環境を整える必要がある。apk ファイルができあがったら Android SDK に付属している adb というコマンドを使って apk ファイルを Android デバイスにインストールする。これで Android デバイスのランチャー画面から自作の Android アプリケーションのアイコンが見えるようになるはずだ。

[*1] http://developer.android.com/tools/sdk/ndk/index.html

僕はさっそく Android NDK 開発環境をととのえるために愛用の ThinkPad を開いた。[*2]まず必要なファイルをそろえよう。以下のファイルをダウンロードする。

- Android SDK (adt-bundle-linux-x86_64-20130917.zip) [*3]
- Android NDK (android-ndk-r9-linux-x86_64.tar.bz2) [*4]

そして Linux 用の JVM をインストールした後、Android SDK/NDK のインストールを行なう。

```
$ cd $HOME
$ unzip -x adt-bundle-linux-x86_64-20130917.zip
$ mv adt-bundle-linux-x86_64-20130917/sdk $HOME/android-sdk
$ rm -f adt-bundle-linux-x86_64-20130917
$ export PATH=$PATH:$HOME/android-sdk/tools:$HOME/android-sdk/platform-tools
$ tar xf android-ndk-r9-linux-x86_64.tar.bz2
$ mv android-ndk-r9 android-ndk
$ export PATH=$PATH:$HOME/android-ndk
$ sudo apt-get install openjdk-7-jdk ant lib32z1-dev lib32stdc++6
$ android update sdk --no-ui --force
```

Android SDK/NDK のセットアップが終わったので、なにかサンプルアプリケーションをビルドしてみよう。さっき展開した Android NDK の中に native-activity という NativeActivity を使ったサンプルアプリケーションが含まれている。どうやら非常に規模の小さいアプリケーションのようだ。今回はこの native-activity をスナッチしよう。さっそくスナッチの前にこのサンプルをビルドして apk ファイルを作ってみよう。

```
$ cd $HOME/android-ndk/samples/native-activity
$ android update project -p ./ -n native-activity -t android-10
$ ndk-build NDK_DEBUG=1
$ ant debug
$ file bin/native-activity-debug.apk
bin/native-activity-debug.apk: Java Jar file data (zip)
```

Android NDK アプリケーションのコンパイルは非常に簡単にできるようだ。このビルドで生成された apk ファイルに Android にインストールするアプリケーションが入っている。最後にこの apk ファイルを実機にインストールしてみよう。まず Android デバイスと PC を USB ケーブルで繋ぎ、adb コマンドで接続確認をする。

```
$ sudo adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
0X5X4X0X0X3X1X0X        device
```

“List of devices attached” の後にデバイスが見えれば接続に成功している。デバイスが見えたことを確認できたら続いて adb コマンドで apk ファイルをインストールしよう。

```
$ sudo adb install -r bin/native-activity-debug.apk
1221 KB/s (156823 bytes in 0.125s)
        pkg: /data/local/tmp/native-activity-debug.apk
Success
```

[*2] Debian GNU/Linux amd64 sid 2013/11/01 時点。GHC のバージョンは 7.6.3。Ajhc のバージョンは 0.8.0.9。

[*3] http://developer.android.com/sdk/index.html

[*4] http://developer.android.com/tools/sdk/ndk/index.html

インストールが終わったら、Android デバイスのランチャーから”NativeActivity” というアプリを立ち上げてみよう。アプリの黒い画面を指でなぞると色が変化するはずだ。この”NativeActivity” アプリはタッチセンサーの入力に応じて画面に一枚だけ表示しているポリゴンの色を変化させる OpenGL ES のデモアプリケーションだ。

31.4.2 Android NDK サンプルソースコード

さっきコンパイルした Android NDK サンプルソースコード”$HOME/android-ndk/samples/native-activity” の中身をざっくり調べておこう。

```
native-activity
|-- AndroidManifest.xml
|-- default.properties
|-- jni
|   |-- Android.mk
|   |-- Application.mk
|   `-- main.c
`-- res
    `-- values
        `-- strings.xml
```

設定ファイル群については置いておくとして、なんと C 言語のソースコードは 1 ファイルしか入っていないようだ。詳しいソースコード解析はスナッチしながらするとして、おおざっぱに main.c ファイルの中をのぞいてみよう。[*5]

```
// -- snip -- Some #include lines
struct saved_state {
// -- snip --
struct engine {
// -- snip --
static int engine_init_display(struct engine* engine) {
// -- snip --
static void engine_draw_frame(struct engine* engine) {
// -- snip --
static void engine_term_display(struct engine* engine) {
// -- snip --
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
// -- snip --
static void engine_handle_cmd(struct android_app* app, int32_t cmd) {
// -- snip --
void android_main(struct android_app* state) {
    struct engine engine;

    memset(&engine, 0, sizeof(engine));
    state->userData = &engine;
    state->onAppCmd = engine_handle_cmd;
    state->onInputEvent = engine_handle_input;
// -- snip --
```

[*5] native-activity サンプルソースコードとこの記事に記載されているその他のソースコードのライセンスは Apache License 2.0 http://www.apache.org/licenses/LICENSE-2.0。

構造体の定義が 2 つ。static 関数が 5 つ。エントリ関数が 1 つ。というシンプルな構成だ。android_native_app_glue.h を include することで生の Android NDK ではなく android_native_app_glue ラッパーを使っていることがわかる。static 関数はエントリ関数の頭でコールバックとして設定する。Android フレームワーク側からのイベントをこのコールバックで処理する。つまり android_native_app_glue を使った Android NDK アプリケーションはイベントドリブンだということになる (図 31.3)。ここまでで図 31.1 の Step 1 が完了し、Step 2 への準備が整った。

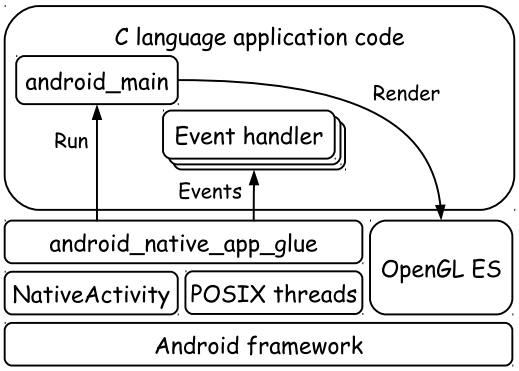


図 31.3: Android NDK アプリの構造

31.5 スナッチ 1: 何もしない Haskell プログラム

さぁ最初のスナッチを開始しよう。この章は図 31.1 の Step 2 に対応する。実はスナッチ設計においてもっとも難しいのはこの最初のスナッチだ。C 言語のコードと Haskell のコードを連携させるだけであれば楽なのだが、最初のスナッチではその連携をさせるためのビルドプロセスを先に作らなければならないからだ。なぜ独自のビルドプロセスが必要になるのだろうか？ 今の Android NDK アプリケーションのビルドシステムは C 言語のビルドしか考えていない。このビルドシステムに Haskell ソースコードのビルドさせる必要がある。Haskell ソースコードのビルドに必要な要素とは何なのだろうか？ おおざっぱに以下に列挙してみよう。

- スナッチした Haskell ソースコード
- Haskell ソースコードを C 言語化するための ajhc コマンド呼び出し
- ajhc コマンドが自動生成するランタイムの実行に必要なスタブ関数群
- 元の C 言語ソースコードである main.c と、ajhc コマンドが自動生成したランタイム、Haskell から変換された C 言語ソース[*6]、スタブ関数群の 4 つをコンパイルするための gcc コマンド呼び出し

今回作成するビルドプロセスはややこしいので図 31.4 にまとめてみた。ここで説明するビルドプロセスはこのスナッチをする中で Makefile にまとめることにする。このビルドシステムでは元の C 言語のソースコードは jni ディレクトリに、Haskell のソースコードは hs_src ディレクトリに配置する。Haskell ソースはまずはじめに Ajhc によってコンパイルされて hs_build ディレクトリに C 言語のファイルとして出力される。この時、Ajhc は Haskell ソースコード由来の C 言語ファイル hs_main.c だけではなく、ランタイムのソースコード hs_build/lib/*.c と hs_build/rts/*.c も吐き出す。その後、サンプルコードに付属していた ant のビルドシステムが走り、これまで登場した C 言語のソー

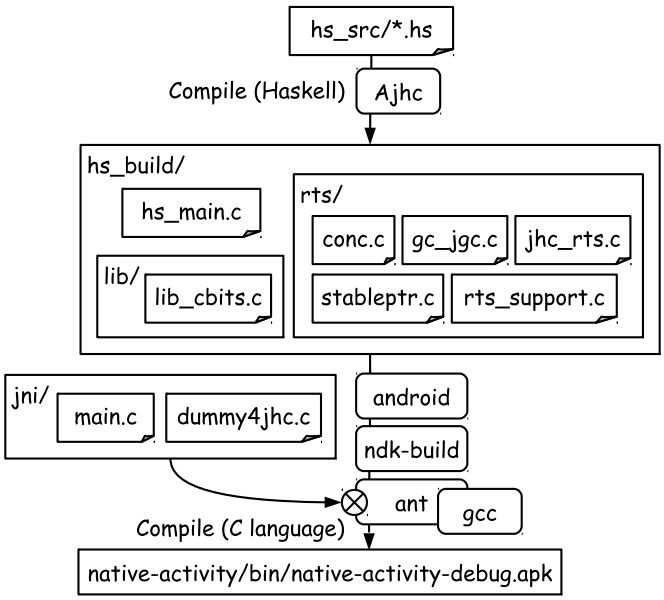


図 31.4: Android NDK Haskell アプリのビルドプロセス

[*6] Ajhc コンパイラは Haskell ソースコードを C 言語ソースコードに変換した後、その C 言語ソースコードと Ajhc ランタイムソースコードを GCC にコンパイルさせることで実行バイナリを得る

スコードを全てまとめてコンパイルし、Android アプリケーションである apk ファイルを作成する。

スナッチを開始する前に Ajhc Haskell コンパイラをインストールしよう。

```
$ sudo apt-get install haskell-platform libncurses5-dev gcc m4
$ export PATH=$HOME/.cabal/bin:$PATH
$ cabal update
$ cabal install ajhc
$ ajhc --version
ajhc 0.8.0.10 (ef04922345d3503235620c86405af838f0271347)
compiled by ghc-7.6 on a x86_64 running linux
```

まず、これまで調べたサンプルアプリケーションのビルド手順を Makefile にまとめておこう。これで make コマンドを実行するだけで C 言語のサンプルアプリケーションがビルドできるようになった。

```
APPNAME = native-activity
ANDROID_TARGET = android-10
CDIR = jni
CSRC = $(wildcard $(CDIR)/*.c $(CDIR)/*.h)

all: bin/$(APPNAME)-debug.apk

bin/$(APPNAME)-debug.apk: $(CSRC)
        android update project -p ./ -n $(APPNAME) -t $(ANDROID_TARGET)
        ndk-build NDK_DEBUG=1
        ant debug
```

Makefile の文法に慣れていなければ、この Makefile の記述についての理解を飛ばしてしまっても良い。ここでは図 31.4 のビルドプロセスについて理解できていれば十分だからだ。

図 31.1 の Step 2 によると最初のスナッチではまず Ajhc のランタイム部分だけを既存 C 言語プログラムに埋め込むのだった。これは具体的には何をすべきだろうか？ Ajhc ランタイムだけを動作させることはできない。そのランタイムの上で動作させる Haskell プログラムが必要になる。もっとも簡単な Haskell プログラムとはなんだろう。hs_src/Main.hs というファイルに何もしない Haskell プログラムを書こう。

```
main = return ()
```

この何もしない Haskell プログラムを C 言語のエントリ関数から呼び出してみる。jni/main.c ファイルを修正しよう。

```
$ git diff jni/main.c
diff --git a/jni/main.c b/jni/main.c
index 2e19635..4767594 100644
--- a/jni/main.c
+++ b/jni/main.c
@@ -234,6 +234,17 @@ void android_main(struct android_app* state) {
     // Make sure glue isn't stripped.
     app_dummy();

+    {
+        // Init & run Haskell code.
+        int hsargc = 1;
+        char *hsargv = "q";
```

```
+        char **hsargvp = &hsargv;
+
+        hs_init(&hsargc, &hsargvp);
+        _amain();
+        // hs_exit(); /* FALLTHROUGH */
+    }
+
     memset(&engine, 0, sizeof(engine));
     state->userData = &engine;
     state->onAppCmd = engine_handle_cmd;
```

jni/main.c ファイルの_amain() 関数によって Haskell の main 関数が呼び出される。当然、これだけでコンパイルが通るわけはない。Ajhc のランタイムはいくつかの POSIX の関数に依存しているためだ。しかし Android アプリでは標準入出力などは使わないため、それらの関数のいくつかはダミーの空関数にすることができる。そこで、その関数群のダミー実装を提供する jni/dummy4jhc.c ファイルを作る。

```
#include "jhc_rts_header.h"
int jhc_printf_stderr(const char *format, ...) { return 0; }
int jhc_fputs_stderr(const char *s) { return 0; }
int jhc_fflush_stdout() { return 0; }
void jhc_print_profile(void) { return 0; }
int jhc_utf8_getchar(void) { return 0; }
int jhc_utf8_getc(FILE *f)  { return 0; }
int jhc_utf8_putchar(int ch)  { return 0; }
int jhc_utf8_putc(int ch, FILE *f)  { return 0; }
```

Ajhc では JHC_USE_OWN_STDIO フラグをコンパイル時に渡した場合には上記の関数を独自実装に置き換えることができる。JHC_USE_OWN_STDIO フラグを使わなかった場合には上記の関数群は POSIX を使った標準実装になる。Ajhc のランタイムのファイル群とさっき作った dummy4jhc.c ファイルをコンパイル対象に追加するために、jni/Android.mk ファイルを修正する。

```
$ git diff jni/Android.mk
diff --git a/jni/Android.mk b/jni/Android.mk
index 9e64d80..9727688 100644
--- a/jni/Android.mk
+++ b/jni/Android.mk
@@ -17,8 +17,16 @@ LOCAL_PATH := $(call my-dir)
 include $(CLEAR_VARS)

 LOCAL_MODULE    := native-activity
-LOCAL_SRC_FILES := main.c
+LOCAL_SRC_FILES := main.c dummy4jhc.c  ../hs_build/hs_main.c \
+       ../hs_build/rts/rts_support.c ../hs_build/rts/jhc_rts.c \
+       ../hs_build/lib/lib_cbits.c ../hs_build/rts/gc_jgc.c \
+       ../hs_build/rts/stableptr.c ../hs_build/rts/conc.c
+LOCAL_C_INCLUDES := hs_build/cbits hs_build
+LOCAL_CFLAGS     := -std=gnu99 -D_GNU_SOURCE -falign-functions=4 \
+       -ffast-math -fno-strict-aliasing -DNDEBUG -D_JHC_GC=_JHC_GC_JGC \
+       -D_JHC_CONC=_JHC_CONC_PTHREAD -D_JHC_USE_OWN_STDIO
 LOCAL_LDLIBS    := -llog -landroid -lEGL -lGLESv1_CM
```

```
+LOCAL_ARM_MODE  := arm
 LOCAL_STATIC_LIBRARIES := android_native_app_glue

 include $(BUILD_SHARED_LIBRARY)
```

最後に Ajhc を使った Haskell から C 言語への変換を Makefile で自動化しよう。

```
$ git diff Makefile
diff --git a/Makefile b/Makefile
index 423e04a..dd6bd7a 100644
--- a/Makefile
+++ b/Makefile
@@ -3,13 +3,23 @@ ANDROID_TARGET = android-10
 CDIR = jni
 CSRC = $(wildcard $(CDIR)/*.c $(CDIR)/*.h)

+HSBUILD = hs_build
+HSDIR = hs_src
+HSSRC = $(wildcard $(HSDIR)/*.hs)
+
 all: bin/$(APPNAME)-debug.apk

-bin/$(APPNAME)-debug.apk: $(CSRC)
+bin/$(APPNAME)-debug.apk: $(HSBUILD)/hs_main.c $(CSRC)
        android update project -p ./ -n $(APPNAME) -t $(ANDROID_TARGET)
        ndk-build NDK_DEBUG=1
        ant debug

+$(HSBUILD)/hs_main.c: $(HSSRC)
+       mkdir -p $(HSBUILD)
+       ajhc -fffi -fpthread --include=hs_src \
+            --tdir=$(HSBUILD) -C -o $@ $(HSDIR)/Main.hs
```

これでサンプルアプリケーションは” 一度何もしない Haskell コードを実行してから起動する” ようになった。一見、今回のスナッチには何の意味もないように見える。しかしこの最初のスナッチは実は無から有を作り出す大事な作業だ。今回のスナッチをする前はアプリケーションのソースコードは全部 C 言語で書かれていた。この C 言語コードを Haskell 化しようとしても、Haskell コードを配置する場所がなかった。なぜなら Haskell コードが一行も書かれていなかったからだ。また Ajhc のランタイムもなかった。最初のスナッチによってたった一行だけの Haskell コードと Ajhc ランタイムをアプリケーションが内在するようになった (図 31.5)。

スナッチが終わったらビルドが通ることと、元のアプリケーションと同様の動作を確認する必要がある。今回は何もスナッチせず、単に何もしない Haskell 関数を呼び出しているだけだ。それでも Ajhc ランタイムの初期化でエラーが発生する可能性は否定できない。幸運にもこの時点でコンパイルの異常と動作の異常は見られなかった。

これで Haskell コードを置く場所ができ、またその Haskell コードが実行可能であることが確かめられた。少なくとも Ajhc ランタイムの初期化には成功している。この後は、Haskell コードを含むアプリケーションがコンパイル可能/実行可能な状態を維持しているか確かめながらスナッチを進めることになる。

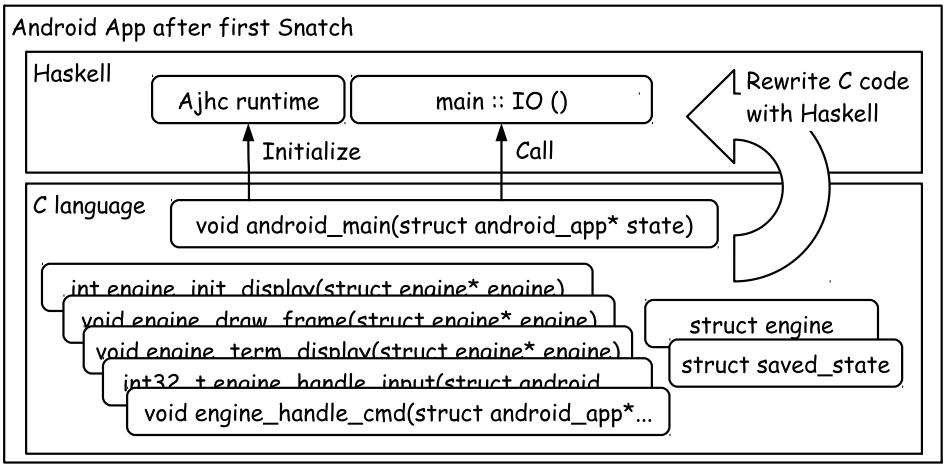


図 31.5: 最初のスナッチで C コードを Haskell 化できるようになった

31.6 スナッチ 2: 最初のターゲット

これで Android 上で Haskell コードを動かす準備ができた。つまり C 言語のコードを Haskell でスナッチする準備が整ったわけだ。図 31.1 の Step 3 に進むことにしよう。さて、まずどこから手をつけるべきだろうか？ 最初に調べた時に内部の構造は関数と構造体定義だけの簡単なものだった。強い型を好む僕としては C 言語の構造体定義に Haskell の型を付けたいと思うところだが、おそらく構造体定義はほうぼうの関数で共用しているはずなので、ある程度スナッチが進行した後でないと Haskell の型を付けても影響範囲が大きいだろう。まずは一番簡単な関数を Haskell 化してみよう。行数と構造から見て一番簡単な C 言語の関数は engine_handle_input() のようだ。

```
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
    struct engine* engine = (struct engine*)app->userData;
    if (AInputEvent_getType(event) == AINPUT_EVENT_TYPE_MOTION) {
        engine->animating = 1;
        engine->state.x = AMotionEvent_getX(event, 0);
        engine->state.y = AMotionEvent_getY(event, 0);
        return 1;
    }
    return 0;
}
```

この関数はどのような処理を行なっているのだろうか？

1. struct android_app から struct engine へのポインタを取り出す
2. AInputEvent_getType() 関数で AInputEvent からイベント種別を取得
3. AMotionEvent_getX() と AMotionEvent_getY() で AInputEvent から xy 座標を取得
4. struct engine のメンバーを書き換える

どうやら関数の処理の中心には struct engine がいてデータを仲介しているようだ。さっき構造体よりも関数をスナッチ対象にすると決めたが、この関数を見るかぎり struct engine の方を先にスナッチするべきに見える。

Haskell の FFI の使い方を復習しておこう。構造体のスナッチには Foreign.Storable.Storable クラス[*7] [*8] を使う。Storable クラスのインスタンスは poke と peek という 2 つのアクションを定義する。これらのアクションの中で、C 言語の構造体と Haskell の型との相互変換を記述する。すると、当該の C 言語の構造体へのポインタを peek すると Haskell ヒープに確保された Haskell の型が得られ、また C 言語の構造体へのポインタへ Haskell の型を poke すると Haskell の型に内在していたデータが C 言語の構造体への書き込まれる。

もう少しだけ説明しておこう。Haskell では関数とデータの境界はあいまいだ。しかし C 言語では手続とデータ、つまり関数と構造体は明確に分かれている。つまり C 言語のスナッチにおいては、手続のスナッチは foreign import を使って C 言語の関数を Haskell から呼び出し可能にすればいい。構造体のスナッチは Storable クラスを使って C 言語の構造体と Haskell の型の相互変換のルールを記述すればいい。スナッチがあまり進行していない時点では、この論理をあまり崩さない方がよい。既存の C 言語の要素をスナッチするときに邪魔になることが多い。もし完全にスナッチが完了して C 言語側の構造体を使わなくなったら、Haskell の型に Storable クラスを継承させる必要はない。また C 言語の関数を完全に Haskell の関数に置き換えてしまったら foreign import は不要となり、直接 Haskell 関数同士を接続できることになる。

手を動かす前に自分がどんなコードを設計したいのか想像してみよう (図 31.6)。通常、可能な場面ではボトムアップ設計よりもトップダウン設計でやった方が効率が良いはずだ。この設計に必要なテクニックは 3 つに大別される。

1. スナッチ対象になっている C 言語関数の Haskell 化
2. Storable クラスで C 言語の構造体に Haskell の型を付ける
3. スナッチ対象が使用している C 言語の関数群を foreign import

一度のスナッチでどこから手をつけるべきかは場合による。トップダウンに設計したい場合には 1,2,3 の順番。ボトムアップに設計したい場合には 3,2,1 の順番だろう。今回のスナッチではスナッチ対象になっている C 言語の関数の規模も小さいのでトップダウン設計を採用することにする。C 言語の関数の規模が大きい場合には、まず既存の C 言語関数を foreign import してしてから C 言語の関数の中身を一度にスナッチするのではなく少しずつ Haskell 関数化している手もある。

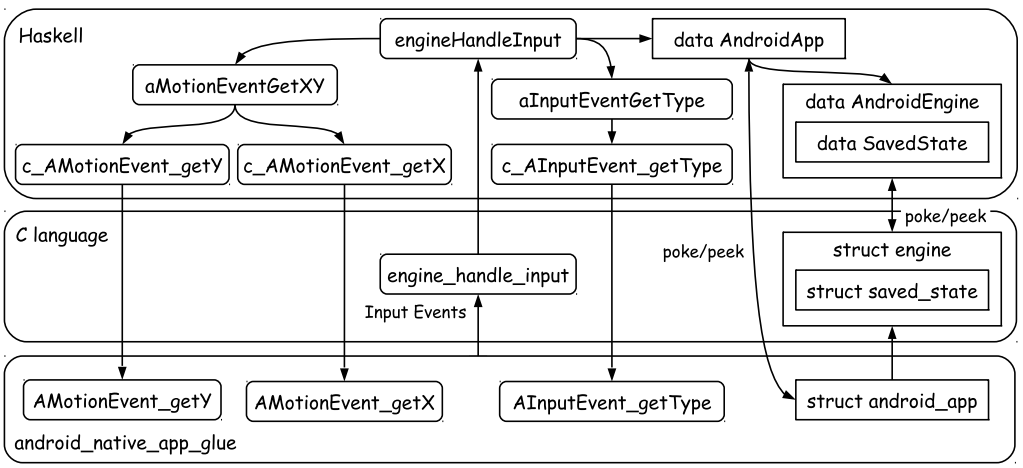


図 31.6: スナッチ後の engine_handle_input() 関数のイメージ

31.6.1 スナッチ対象になっている C 言語関数の Haskell 化

先の図 31.6 を参考にしてソースコードをトップダウンに書いてみよう。例えば hs_src/Main.hs はこんな感じだろうか。

[*7] Foreign.Storable.Storable http://hackage.haskell.org/package/base/docs/Foreign-Storable.html

[*8] 参考: http://itpro.nikkeibp.co.jp/article/COLUMN/20080805/312151/?ST=develop&P=3

```
{-# LANGUAGE ForeignFunctionInterface #-}
import Foreign.Ptr
import Foreign.Storable
import AndroidNdk

main = return ()

type FuncHandleInput =
  Ptr AndroidApp -> Ptr AInputEvent -> IO Int
foreign export ccall "engineHandleInput"
  engineHandleInput :: FuncHandleInput
engineHandleInput :: FuncHandleInput
engineHandleInput appP evP = aInputEventGetType evP >>= go
  where
    go AInputEventTypeMotion = do
      engP <- appUserData `fmap` peek appP
      peek engP >>= updateEngine >>= poke engP
      return 1
    go _ = return 0
    updateEngine eng = do
      (x, y) <- aMotionEventGetXY evP
      let stat = engState eng
      return $ eng { engAnimating = 1
                   , engState = stat { sStateX = truncate x,
                                       sStateY = truncate y }}
```

どー見ても C 言語の engine_handle_input() 関数をそのまま写しただけだ。スナッチ設計では C 言語から Haskell に変換している段階ではこれ以上抽象化を行なわない方が良い。いくらこの段階で抽象化したところで、まだこの関数の外部と C 言語のインターフェイスで繋ってしまっているため局所最適化になってしまうのだ。Haskell を使った抽象を楽しむのは完全にスナッチが終わった後に取っておくことにしよう。

この Haskell で書かれた engineHandleInput 関数は foreign export で C 言語から呼び出せるようになっている。main.c から呼び出すとしたらこんな感じだろうか。つまり engine_handle_input() 関数の中身がごっそり Haskell 化できることになる。

```
static int32_t engine_handle_input(struct android_app* app,AInputEvent* event){
    return engineHandleInput(app, event);
}
```

31.6.2 Storable クラスで C 言語の構造体に Haskell の型を付ける

さて、こんな実装に移行するためにはどうやって Storable を使えば良いだろうか。なんとなくそこそこ大きな実装になる予感がする。ちゃんとしたインターフェイスを切った方がいいだろう。まず hs_src/AndroidNdk.hs ファイルに AndroidNdk というモジュールを作ろう。

```
module AndroidNdk ( module AndroidNdk
                  , module AndroidNdk.Storable
                  , module AndroidNdk.Import ) where
import AndroidNdk.Storable
import AndroidNdk.Import
```

図 31.3 によると今回のサンプルアプリの下のレイヤーは android_native_app_glue と OpenGL ES のようだ。この AndroidNdk モジュールには android_native_app_glue 由来の要素を格納することにする。また AndroidNdk モジュールは NativeActivity を使った全てのアプリケーションを抽象化できるようには設計しない。現時点では NativeActivity のしくみとアプリケーション群からの要求が見えないからだ。AndroidNdk モジュールの中身は Storable と Import の二種類に分かれている。C 言語の構造体を Storable でラップするのが AndroidNdk.Storable モジュール。C 言語の関数を foreign import するのが AndroidNdk.Import モジュールだ。

まずなにはともあれデータ型、つまり構造体を Storable でラップしないことには関数を作れない。今回のサンプルアプリケーションで出てくる構造体は 3 つあるが、その包含関係は以下のようなっている。

```
struct android_app ∋ struct engine ∋ struct saved_state
```

ということは最初に Storable インスタンスにするべきなのは struct saved_state だろう。struct engine を Strorable インスタンスにしようとしても先に struct saved_state が Haskell から取り扱えねばならないからだ。hs_src/AndroidNdk/Storable.hs ファイルを作成して、SavedState 型を作ろう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module AndroidNdk.Storable (
  AndroidApp(..),
  AndroidEngine(..),
  SavedState(..)
  ) where
import Foreign.Storable
import Foreign.Ptr

-- struct saved_state
foreign import primitive "const.sizeof(struct saved_state)"
  sizeOf_SavedState :: Int
foreign import primitive "const.offsetof(struct saved_state, x)"
  offsetOf_SavedState_x :: Int
foreign import primitive "const.offsetof(struct saved_state, y)"
  offsetOf_SavedState_y :: Int
data SavedState =
  SavedState { sStateX :: Int
             , sStateY :: Int }
instance Storable SavedState where
  sizeOf    = const sizeOf_SavedState
  alignment = sizeOf
  poke p sstat = do
    pokeByteOff p offsetOf_SavedState_x $ sStateX sstat
    pokeByteOff p offsetOf_SavedState_y $ sStateY sstat
  peek p = do
    x <- peekByteOff p offsetOf_SavedState_x
    y <- peekByteOff p offsetOf_SavedState_y
    return $ SavedState { sStateX = x, sStateY = y }
```

ここで少し気になることがある。struct saved_state は以下のように C 言語側で定義されていた。

```
struct saved_state {
    float angle;
    int32_t x;
```

```
    int32_t y;
};
```

しかし今回作った SavedState 型ではこれら構造体メンバーの内で x と y のみ Haskell 側から使えるようにしている。angle メンバーは SavedState 型に含まれない。angle メンバーに有意な値が入っている struct saved_state へのポインタに SavedState 型を poke した場合、angle メンバーの内容が消えてしまうようなことはないのだろうか？ 結論から言うと使い方に気をつければ angle メンバーが消えることはない。図 31.7 を見てほしい。この図では stateC という C 言語構造体の実体に Haskell 側の SavedState 型のインスタンスである stateHs を poke している。SavedState 型では x と y メンバーに対応する sStateX/sStateY レコードしか定義されていない。そのため、この poke では stateC の x と y メンバーのみ更新されることになる。angle メンバーはそのまま残る。この Storable インスタンスの使い方で注意しなければいけないのは、struct saved_state のメンバー全てが SavedState 型に写されている訳ではないので一旦 C 言語型から渡ってきたポインタを peek したら、必ず同じポインタに対して poke しないと Haskell 側で定義した SavedState 型において欠損しているメンバーは不定値になってしまうことだ。

ところでこのような問題があるのに構造体の一部しか Haskell の型をつけないのだろうか。いくつか理由があるな大きな理由は C 言語の関数を少しずつスナッチするのと同じように、C 言語の構造体メンバーの少しずつスナッチした方が安全だからだ。構造体の規模が大きい場合にはその全てのメンバーをスナッチしようとすると誤りが入ってしまう可能性が大きくなる。構造体メンバーの内の一部だけスナッチし、残ったメンバーは C 言語側でのみ取り扱うような設計をしたい場合今回のようなテクニックを使うことができる。本来であれば Ajhc に GHC の vector ライブラリ[*9] を移植して使うべきだろう。

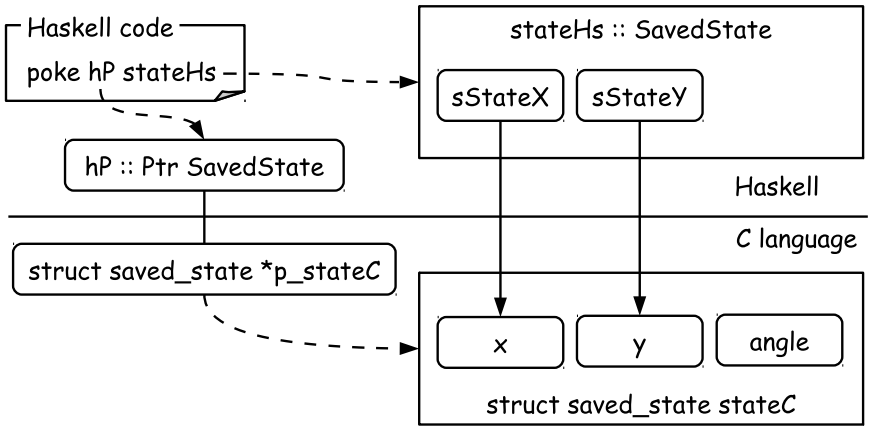


図 31.7: ポインタに対する部分的な poke

また foreign import primitive “const.HOGEHOGE” [*10]という見慣れないキーワードが見える。このキーワードは jhc 独自のもので、HOGEHOGE の位置にあるテキストをコンパイル結果の C 言語ソースコードに直接埋め込むキーワードだ。これは少し危険だが場面によっては大変重宝する。今回のスナッチでは struct saved_state のサイズと構造体先頭からメンバー変数へのオフセットを知るのに使っている。

続いて struct android_app と struct engine も Storable インスタンスにしよう。hs_src/AndroidNdk/Storable.hs に追記する。

```
-- struct engine
foreign import primitive "const.sizeof(struct engine)"
  sizeOf_Engine :: Int
foreign import primitive "const.offsetof(struct engine, animating)"
  offsetOf_Engine_animating :: Int
```

[*9] http://hackage.haskell.org/package/vector
[*10] http://ajhc.metasepi.org/manual_ja.html#foreign プリミティブ

```
foreign import primitive "const.offsetof(struct engine, state)"
  offsetOf_Engine_state :: Int
data AndroidEngine =
  AndroidEngine { engAnimating :: Int
                , engState     :: SavedState }
instance Storable AndroidEngine where
  sizeOf    = const sizeOf_Engine
  alignment = sizeOf
  poke p eng = do
    pokeByteOff p offsetOf_Engine_animating $ engAnimating eng
    pokeByteOff p offsetOf_Engine_state     $ engState eng
  peek p = do
    animating <- peekByteOff p offsetOf_Engine_animating
    state     <- peekByteOff p offsetOf_Engine_state
    return $ AndroidEngine { engAnimating = animating
                           , engState     = state }
-- struct android_app
foreign import primitive "const.sizeof(struct android_app)"
  sizeOf_AndroidApp :: Int
foreign import primitive "const.offsetof(struct android_app, userData)"
  offsetOf_AndroidApp_appUserData :: Int
data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine }
instance Storable AndroidApp where
  sizeOf    = const sizeOf_AndroidApp
  alignment = sizeOf
  poke p app = do
    pokeByteOff p offsetOf_AndroidApp_appUserData $ appUserData app
  peek p = do
    userData <- peekByteOff p offsetOf_AndroidApp_appUserData
    return $ AndroidApp { appUserData = userData }
```

31.6.3　スナッチ対象が使用している C 言語の関数群を foreign import

C 言語の構造体に対する Storable インスタンスが完成したので、今度は C 言語の関数群を hs_src/AndroidNdk/Import.hs に foreign import しよう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module AndroidNdk.Import where
import Foreign.Ptr
import Foreign.C.Types

data AInputEventType =
  AInputEventTypeKey | AInputEventTypeMotion | AInputEventTypeFail
  deriving(Eq)
newtype {-# CTYPE "AInputEvent" #-} AInputEvent = AInputEvent ()

foreign import primitive "const.AINPUT_EVENT_TYPE_KEY"
  c_AINPUT_EVENT_TYPE_KEY :: Int
foreign import primitive "const.AINPUT_EVENT_TYPE_MOTION"
  c_AINPUT_EVENT_TYPE_MOTION :: Int
```

```
foreign import ccall "c_extern.h AInputEvent_getType"
  c_AInputEvent_getType :: Ptr AInputEvent -> IO Int
foreign import ccall "c_extern.h AMotionEvent_getX"
  c_AMotionEvent_getX :: Ptr AInputEvent -> CSize -> IO Float
foreign import ccall "c_extern.h AMotionEvent_getY"
  c_AMotionEvent_getY :: Ptr AInputEvent -> CSize -> IO Float
```

CTYPE "AInputEvent"という型プラグマ[*11]を使った。このプラグマは data や newtype で定義した型に対して C 言語側から見た型を付与する。今回のサンプルアプリケーションの中では AInputEvent という C 言語の型の中身をさわらず、AInputEvent へのポインタをとりまわすだけだ。そんな時はいちいち Storable 型を定義する必要はなく上記の AInputEvent 型のように中身のない型を定義して CTYPE 型プラグマを使えばいい。

このままでもスナッチは継続できるだろう。しかし foreign import しただけの関数は生の C 言語 API のままなので使いにくく、また型安全からは遠い。そこで簡単なラッパーを hs_src/AndroidNdk/Import.hs に追加して強い型を付けよう。

```
aInputEventGetType :: Ptr AInputEvent -> IO AInputEventType
aInputEventGetType p = do
  i <- c_AInputEvent_getType p
  return $ inputEventType i

inputEventType :: Int -> AInputEventType
inputEventType e | e == c_AINPUT_EVENT_TYPE_KEY    = AInputEventTypeKey
                 | e == c_AINPUT_EVENT_TYPE_MOTION = AInputEventTypeMotion
                 | otherwise = AInputEventTypeFail

aMotionEventGetXY :: Ptr AInputEvent -> IO (Float, Float)
aMotionEventGetXY ep = do
  x <- c_AMotionEvent_getX ep 0
  y <- c_AMotionEvent_getY ep 0
  return (x, y)
```

最後にさっきから出てきていた c_extern.h ヘッダに main.c の中にあった include 行と構造体定義を移動させ、main.c から include する。つまり構造体と関数インターフェイス定義を main.c と Haskell コードで共有するのだ (図 31.8)。これで、OpenGL ES の API も Haskell 側からたたけるようになったはずだ。まだこの段階ではその要求はないけれど。

今回のスナッチはこれで一区切りだ。今回のスナッチの途中でもビルド確認はしてきたが、スナッチの終わりビルドと動作確認をしておこう。やはり Android デバイス上で正常に動作しているように見える。次のスナッチに進もう。

31.7 スナッチ 3: Haskell コードの近傍をさらにスナッチ

さっきは C 言語の engine_handle_input() 関数の中身をスナッチした。しかし C 言語のエントリ関数から Haskell のコードを呼び出したままだと、main.c はスナッチが完了してもエントリ関数が残ったままになってしまう。せっかくなら engine_handle_input() 関数そのものを消してしまいたい。どうしたら良いだろう？ この関数を使っているコードはどこだろうか。

```
void android_main(struct android_app* state) {
// -- snip --
    memset(&engine, 0, sizeof(engine));
```

[*11] http://ajhc.metasepi.org/manual_ja.html#型プラグマ

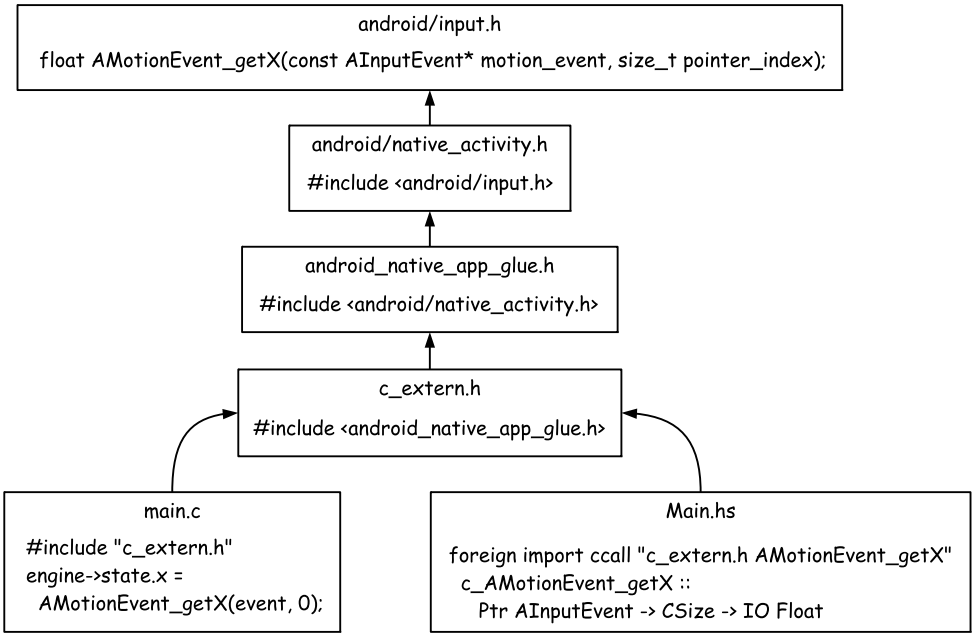


図 31.8: c_extern.h ヘッダで API 定義を C と Haskell で共有

```
    state->userData = &engine;
    state->onAppCmd = engine_handle_cmd;
    state->onInputEvent = engine_handle_input;
    engine.app = state;
```

なるほど。android_main() 関数にて engine_handle_input() の関数ポインタを struct android_app メンバーに代入している。この部分を Haskell 化することはできないだろうか。ここでもトップダウン設計をしてみよう。単純に考えると struct android_app へのポインタを Haskell に渡して onInputEvent メンバーを更新してもらえばいいことになる。

```
$ git diff jni/main.c
diff --git a/jni/main.c b/jni/main.c
index d3a037d..711f3bc 100644
--- a/jni/main.c
+++ b/jni/main.c
@@ -205,7 +198,7 @@ void android_main(struct android_app* state) {
     memset(&engine, 0, sizeof(engine));
     state->userData = &engine;
     state->onAppCmd = engine_handle_cmd;
-    state->onInputEvent = engine_handle_input;
+    initAndroidAppFunc(state);
     engine.app = state;

     // Prepare to monitor accelerometer
```

こんな initAndroidAppFunc 関数はどうなるだろうか？ hs_src/Main.hs ファイルに定義を追加しよう。

```
foreign export ccall "initAndroidAppFunc"
  initAndroidAppFunc :: Ptr AndroidApp -> IO ()
```

```
initAndroidAppFunc :: Ptr AndroidApp -> IO ()
initAndroidAppFunc appP = peek appP >>= updateApp >>= poke appP
  where
    updateApp app = return $ app { appOnInputEvent = p_engineHandleInput }
```

問題は p_engineHandleInput というモノだ。これは FunPtr 型にすべきだがどうやって作ればいいだろうか？ GHC であれば foreign import ccall "wrapper"を使うだろう[*12]。ところが現状の Ajhc では wrapper にはバグがあり使えないようだ[*13] [*14]。しょうがない。手で関数ポインタを作ろう。hs_src/Main.hs を編集する。

```
$ git diff hs_src/Main.hs
diff --git a/hs_src/Main.hs b/hs_src/Main.hs
index beb330a..75e0af1 100644
--- a/hs_src/Main.hs
+++ b/hs_src/Main.hs
@@ -5,10 +5,10 @@ import AndroidNdk

 main = return ()

-type FuncHandleInput =
-  Ptr AndroidApp -> Ptr AInputEvent -> IO Int
 foreign export ccall "engineHandleInput"
   engineHandleInput :: FuncHandleInput
+foreign import ccall "&engineHandleInput"
+  p_engineHandleInput :: FunPtr FuncHandleInput
 engineHandleInput :: FuncHandleInput
 engineHandleInput appP evP = aInputEventGetType evP >>= go
   where
```

一度 export することで"engineHandleInput" という名前のシンボルを作り、再度関数ポインタを import する。なにか自作自演っぽいが、wrapper が使えないのでしょうがない。

次に onInputEvent メンバーを Haskell から取り扱えるようにするために Storable インスタンスを変更しよう。関数の型定義である FuncHandleInput エイリアスは Main.hs でも Storable.hs でも使うので、Main.hs から Storable.hs に移動させる。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 58d5db0..8f259cb 100644
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -2,10 +2,15 @@
 module AndroidNdk.Storable (
   AndroidApp(..),
   AndroidEngine(..),
-  SavedState(..)
+  SavedState(..),
+  FuncHandleInput
   ) where
```

[*12] http://hackage.haskell.org/package/base/docs/Foreign-Ptr.html#t:FunPtr

[*13] FFI functions declared "wrapper" are not supported - Issue #64 https://github.com/ajhc/ajhc/issues/64

[*14] みんながこの記事を読んでいるころまでに対策できていたら素敵でゲソ! (げふぅ!)

```
 import Foreign.Storable
 import Foreign.Ptr
+import AndroidNdk.Import
+
+type FuncHandleInput =
+  Ptr AndroidApp -> Ptr AInputEvent -> IO Int

 -- struct saved_state
 foreign import primitive "const.sizeof(struct saved_state)"
```

最後に AndroidApp 型に onInputEvent メンバーに対応する appOnInputEvent レコードを追加しよう。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 58d5db0..8f259cb 100644
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -55,12 +60,18 @@ foreign import primitive "const.sizeof(struct android_app)"
   sizeOf_AndroidApp :: Int
 foreign import primitive "const.offsetof(struct android_app, userData)"
   offsetOf_AndroidApp_appUserData :: Int
-data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine }
+foreign import primitive "const.offsetof(struct android_app, onInputEvent)"
+  offsetOf_AndroidApp_appOnInputEvent :: Int
+data AndroidApp = AndroidApp { appUserData :: Ptr AndroidEngine
+                             , appOnInputEvent :: FunPtr FuncHandleInput }
 instance Storable AndroidApp where
   sizeOf    = const sizeOf_AndroidApp
   alignment = sizeOf
   poke p app = do
     pokeByteOff p offsetOf_AndroidApp_appUserData $ appUserData app
+    pokeByteOff p offsetOf_AndroidApp_appOnInputEvent $ appOnInputEvent app
   peek p = do
     userData <- peekByteOff p offsetOf_AndroidApp_appUserData
-    return $ AndroidApp { appUserData = userData }
+    onInput <- peekByteOff p offsetOf_AndroidApp_appOnInputEvent
+    return $ AndroidApp { appUserData = userData
+                        , appOnInputEvent = onInput }
```

今回のスナッチは簡単な修正だった。それでも C 言語の関数が一つ消えたのは気持ちがいい。やはり図にまとめて今回のスナッチをふりかえってみよう (図 31.9)。中身だけスナッチ済みだった engine_handle_input() 関数全体を Haskell 化によって C 言語側から消してしまうために当該関数の使用者である android_main() 関数に変更を加えた。結果として android_native_app_glue から直接 Haskell のコールバック関数が呼び出されるようになった。

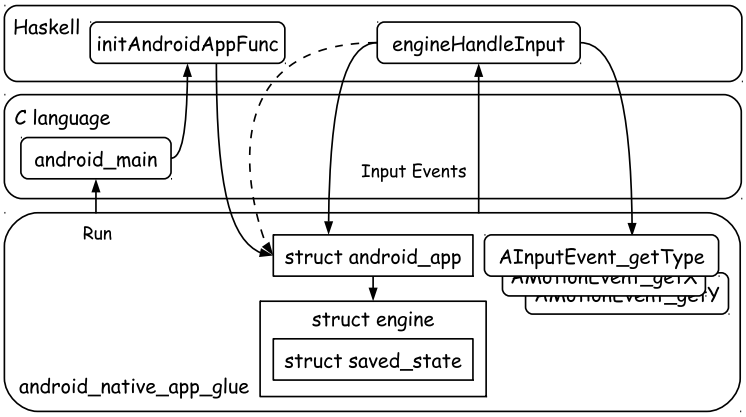


図 31.9: onInputEvent 関数ポインタの初期化をスナッチした結果

簡単な修正だったとはいえ、スナッチの終わりにはビルド確認と動作確認をするべきだ。もしここでエラーが起きていたら今回のスナッチによって不具合が入りこんだことになる。念の為の動作確認だったが、もちろん Android デバイス上での動作は正常に見える。これで今回のスナッチは完了だ。

31.8 こっそりと驚きを届けたい

『さいきん家でゴソゴソなにをやってるんでゲソ？』
「ん。別に」
『めずらしく ThinkPad 開いてるじゃなイカ。Coq の宿題でも出たんでゲソ？』
「そんなんじゃないよ」
『あやしいでげそ……おもしろいことを独り占めするの良くないでゲソ!』
「後でちゃんと説明するよ」

ここでプレゼントを作っているのがバレてしまっては面白さが半減だ。もうちょっと開発速度を上げた方がいいのかもしれない。

31.9 スナッチ 4: OpenGL ES をたたく

さて次はどこをスナッチしようか。engine_handle_input() 関数の近傍のスナッチは終わった。initAndroidAppFunc 関数の近傍のスナッチに進んでも良いが、ここは C 言語側で最も長い android_main() 関数だ。いきなりここを攻めるのは無謀だろう。他に短い C 言語関数を探してみると engine_draw_frame() が見つかった。

```
static void engine_draw_frame(struct engine* engine) {
    if (engine->display == NULL) {
        // No display.
        return;
    }

    // Just fill the screen with a color.
    glClearColor(((float)engine->state.x)/engine->width, engine->state.angle,
            ((float)engine->state.y)/engine->height, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    eglSwapBuffers(engine->display, engine->surface);
}
```

この関数はフレーム毎の描画を OpenGL ES を使って行なう。glClearColor()、glClear()、eglSwapBuffers() という 3 つの関数が見えるが、これらは OpenGL ES の API だ。OpenGL ES のスナッチをするわけでないので、これらの関数は完全にスナッチされた後も C 言語の関数のまま残るだろう。これらの関数を hs_src/OpenGLES.hs に import しよう。

```
{-# LANGUAGE ForeignFunctionInterface #-}
module OpenGLES where
import Data.Word
import Data.Bits
import Foreign.Ptr

type C_GLbitfield = Word32
newtype EGLDisplayT = EGLDisplayT ()
newtype EGLSurfaceT = EGLSurfaceT ()
type EGLDisplay = Ptr EGLDisplayT
type EGLSurface = Ptr EGLSurfaceT

foreign import primitive "const.GL_COLOR_BUFFER_BIT"
  c_GL_COLOR_BUFFER_BIT :: C_GLbitfield
foreign import primitive "const.GL_DEPTH_BUFFER_BIT"
  c_GL_DEPTH_BUFFER_BIT :: C_GLbitfield
foreign import primitive "const.GL_STENCIL_BUFFER_BIT"
  c_GL_STENCIL_BUFFER_BIT :: C_GLbitfield
foreign import ccall "GLES/gl.h glClearColor"
  c_glClearColor :: Float -> Float -> Float -> Float -> IO ()
foreign import ccall "GLES/gl.h glClear"
  c_glClear :: C_GLbitfield -> IO ()
foreign import ccall "EGL/egl.h eglSwapBuffers"
  c_eglSwapBuffers :: EGLDisplay -> EGLSurface -> IO Word32
```

このままでも Haskell でコードを書けるが、C_GLbitfield 型が生の 32bit 値であるので型が弱い。そこで GLBitField 型を作り、[GLBitField] 型でビット指定できるようにしておこう。こうすればビットの意味をあやまって理解してしまうことを防げるはずだ。もっとも今回のスナッチでは gLColorBufferBit しか使わないので combineGLBitField は使うことはない。それでも明らかに型が弱い部分はその都度抽象化をしてやった方がいいだろう。hs_src/OpenGLES.hs に定義を追加する。

```
newtype GLBitField = GLBitField { unGLBitField :: Word32 }
gLColorBufferBit, gLDepthBufferBit, glStenciBufferBit :: GLBitField
gLColorBufferBit  = GLBitField c_GL_COLOR_BUFFER_BIT
gLDepthBufferBit  = GLBitField c_GL_DEPTH_BUFFER_BIT
glStenciBufferBit = GLBitField c_GL_STENCIL_BUFFER_BIT
combineGLBitField :: [GLBitField] -> GLBitField
combineGLBitField = GLBitField . foldr ((.|.) . unGLBitField) 0

glClear :: GLBitField -> IO ()
glClear = c_glClear . unGLBitField

glClearColor :: Float -> Float -> Float -> Float -> IO ()
glClearColor = c_glClearColor
```

```
eglSwapBuffers :: EGLDisplay -> EGLSurface -> IO Word32
eglSwapBuffers = c_eglSwapBuffers
```

これで OpenGL ES の API を Haskell からたたけるようになった。engine_draw_frame() のスナッチしてみよう。

```
$ git diff hs_src/Main.hs
diff --git a/hs_src/Main.hs b/hs_src/Main.hs
index 75e0af1..9527e98 100644
--- a/hs_src/Main.hs
+++ b/hs_src/Main.hs
@@ -1,7 +1,9 @@
 {-# LANGUAGE ForeignFunctionInterface #-}
+import Control.Monad
 import Foreign.Ptr
 import Foreign.Storable
 import AndroidNdk
+import OpenGLES

 main = return ()

@@ -30,3 +32,19 @@ initAndroidAppFunc :: Ptr AndroidApp -> IO ()
 initAndroidAppFunc appP = peek appP >>= updateApp >>= poke appP
   where
     updateApp app = return $ app { appOnInputEvent = p_engineHandleInput }
+
+foreign export ccall "engineDrawFrame"
+  engineDrawFrame :: Ptr AndroidEngine -> IO ()
+engineDrawFrame :: Ptr AndroidEngine -> IO ()
+engineDrawFrame engP = do
+  eng <- peek engP
+  when ((engEglDisplay eng) /= nullPtr) $ do
+    let w     = fromIntegral $ engWidth eng
+        h     = fromIntegral $ engHeight eng
+        s     = engState eng
+        x     = fromIntegral $ sStateX s
+        y     = fromIntegral $ sStateY s
+        angle = sStateAngle s
+    glClearColor (x/w) angle (y/h) 1.0
+    glClear $ gLColorBufferBit
+    void $ eglSwapBuffers (engEglDisplay eng) (engEglSurface eng)
```

SavedState に angle を AndroidEngine に width,height,display,surface を追加しよう。

```
$ git diff hs_src/AndroidNdk/Storable.hs
diff --git a/hs_src/AndroidNdk/Storable.hs b/hs_src/AndroidNdk/Storable.hs
index 8f259cb..cfe24c9 100644
--- a/hs_src/AndroidNdk/Storable.hs
+++ b/hs_src/AndroidNdk/Storable.hs
@@ -8,6 +8,7 @@ module AndroidNdk.Storable (
```

```
 import Foreign.Storable
 import Foreign.Ptr
 import AndroidNdk.Import
+import OpenGLES

 type FuncHandleInput =
   Ptr AndroidApp -> Ptr AInputEvent -> IO Int
@@ -19,19 +20,24 @@ foreign import primitive "const.offsetof(struct saved_s...
   offsetOf_SavedState_x :: Int
 foreign import primitive "const.offsetof(struct saved_state, y)"
   offsetOf_SavedState_y :: Int
+foreign import primitive "const.offsetof(struct saved_state, angle)"
+  offsetOf_SavedState_angle :: Int
 data SavedState =
   SavedState { sStateX :: Int
-             , sStateY :: Int }
+             , sStateY :: Int
+             , sStateAngle :: Float }
 instance Storable SavedState where
   sizeOf    = const sizeOf_SavedState
   alignment = sizeOf
   poke p sstat = do
     pokeByteOff p offsetOf_SavedState_x $ sStateX sstat
     pokeByteOff p offsetOf_SavedState_y $ sStateY sstat
+    pokeByteOff p offsetOf_SavedState_angle $ sStateAngle sstat
   peek p = do
     x <- peekByteOff p offsetOf_SavedState_x
     y <- peekByteOff p offsetOf_SavedState_y
-    return $ SavedState { sStateX = x, sStateY = y }
+    angle <- peekByteOff p offsetOf_SavedState_angle
+    return $ SavedState { sStateX = x, sStateY = y, sStateAngle = angle }

 -- struct engine
 foreign import primitive "const.sizeof(struct engine)"
@@ -40,20 +46,44 @@ foreign import primitive "const.offsetof(struct engine...
   offsetOf_Engine_animating :: Int
 foreign import primitive "const.offsetof(struct engine, state)"
   offsetOf_Engine_state :: Int
+foreign import primitive "const.offsetof(struct engine, width)"
+  offsetOf_Engine_width :: Int
+foreign import primitive "const.offsetof(struct engine, height)"
+  offsetOf_Engine_height :: Int
+foreign import primitive "const.offsetof(struct engine, display)"
+  offsetOf_Engine_display :: Int
+foreign import primitive "const.offsetof(struct engine, surface)"
+  offsetOf_Engine_surface :: Int
 data AndroidEngine =
   AndroidEngine { engAnimating :: Int
-                , engState     :: SavedState }
```

```
+                , engState     :: SavedState
+                , engWidth      :: Int
+                , engHeight     :: Int
+                , engEglDisplay :: EGLDisplay
+                , engEglSurface :: EGLSurface }
 instance Storable AndroidEngine where
   sizeOf    = const sizeOf_Engine
   alignment = sizeOf
   poke p eng = do
     pokeByteOff p offsetOf_Engine_animating $ engAnimating eng
     pokeByteOff p offsetOf_Engine_state     $ engState eng
+    pokeByteOff p offsetOf_Engine_width     $ engWidth eng
+    pokeByteOff p offsetOf_Engine_height    $ engHeight eng
+    pokeByteOff p offsetOf_Engine_display   $ engEglDisplay eng
+    pokeByteOff p offsetOf_Engine_surface   $ engEglSurface eng
   peek p = do
     animating <- peekByteOff p offsetOf_Engine_animating
     state     <- peekByteOff p offsetOf_Engine_state
+    width               <- peekByteOff p offsetOf_Engine_width
+    height              <- peekByteOff p offsetOf_Engine_height
+    eglDisp             <- peekByteOff p offsetOf_Engine_display
+    eglSurf             <- peekByteOff p offsetOf_Engine_surface
     return $ AndroidEngine { engAnimating = animating
-                           , engState     = state }
+                           , engState     = state
+                           , engWidth      = width
+                           , engHeight     = height
+                           , engEglDisplay = eglDisp
+                           , engEglSurface = eglSurf }

 -- struct android_app
 foreign import primitive "const.sizeof(struct android_app)"
```

ここまででわかる通り、Storable の管理は単純作業だ。設計対象によっては自動化できるかもしれない。

最後に C 言語の engine_draw_frame() 関数から Haskell の engineDrawFrame 関数を呼び出せばスナッチは終わりだ。

```
static void engine_draw_frame(struct engine* engine) {
    engineDrawFrame(engine);
}
```

簡単なスナッチだったが図にまとめておこう (図 31.10)。今回はサンプルアプリケーションのスナッチなので、OpenGL ES の中まではスナッチしない。しかし C 言語で書かれた OpenGL ES と Haskell のコードの共存は可能なようだ。

今回のスナッチでは新しい部品である OpenGL ES の呼び出しを Haskell 化した。スナッチに失敗していた場合には画面描画がおかしくなるはずだ。スナッチの完了前にビルドと動作確認をしてみよう。Android デバイスにインストールしての動作確認は正常。画面描画にも問題は見られない。これで次のスナッチに安心して進むことができる。

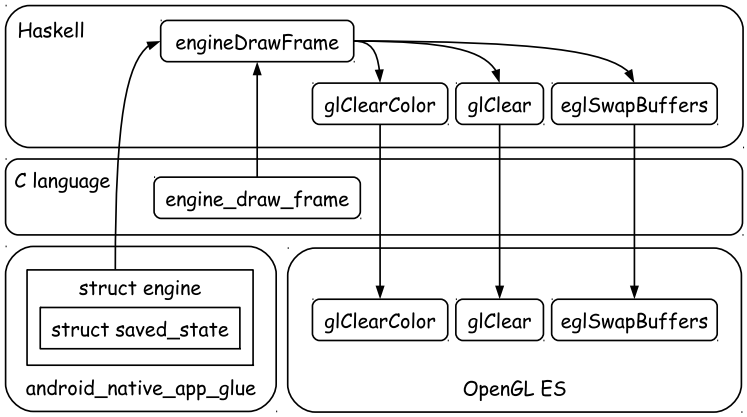


図 31.10: engine_draw_frame() 関数をスナッチした結果

31.10　スナッチ完了

その後数日かけて僕はサンプルアプリケーションのスナッチを続けてみた。たしかに作業に時間はかかったが、スナッチに使ったテクニック自体はこれまで使ってきたものを応用すればなんとかなったようだ。最終的に main.c をこれ以上 Haskell 化できないところまでスナッチを進行させることができた。その結果 main.c には何が残ったのだろうか？

```
void android_main(struct android_app* state) {
    app_dummy(); // Make sure glue isn't stripped.

    // Init & run Haskell code.
    int hsargc = 1;
    char *hsargv = "q";
    char **hsargvp = &hsargv;

    hs_init(&hsargc, &hsargvp);
    androidMain(state);
    hs_exit();
}
```

もはやこの C 言語のエントリコードは有意なロジックとは呼べない。以下 3 つのアクションを順番に呼び出しているにすぎないからだ。Android NDK サンプルソースコードは完全にスナッチされたと考えていいだろう。つまり図 31.1 の Step 5 まで到達できたことになる。

- hs_init(): Ajhc ランタイムの初期化
- androidMain(): Haskell エントリ関数の呼び出し
- hs_exit(): Ajhc ランタイムの終了

31.11　プレゼントを作ろう

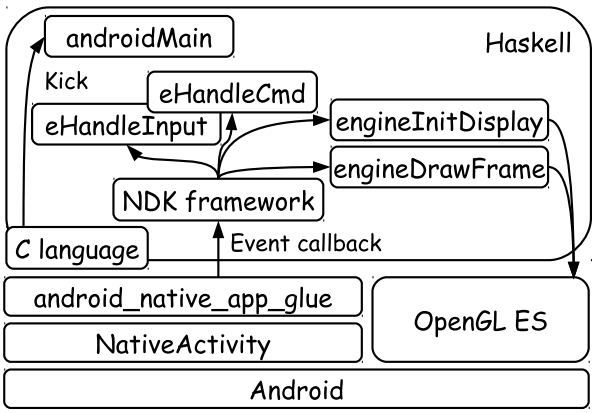


図 31.11: Haskell を使った Android NDK フレームワーク

せっかくスナッチしたので Android NDK フレームワークのようなものを作ってみよう (図 31.11)。結局 C 言語と Haskell の界面であるコールバックが煩雑な処理になりやすいので、その部分を NDK framework として隠した。この抽象化が最善だとは思えないが、そこは型の強い言語、後で再設計するのはそれほど大変ではない。

このスナッチ済みサンプルアプリケーションを改造して何か面白いアプリは作れないだろうか。結局、先のサンプルはイベントをコールバックで受け取って、メインコンテキストで OpenGL ES を使って画面描画をするだけだ。タッチパネルで立方体を回転させるぐらいであれば、それほど苦労しないはずだ。既にスナッチは完了しているので、今回の改造は純粋な Haskell プログラミングで済んだ (図 31.12)。

単純なアプリだがタッチ UI で立方体で動かすのは少し不思議な気分がする。カノジョはこのアプリ[*15] を喜んでくれるだろうか。その様子を想像して、僕は今度の週末が楽しみになった。

31.12　お互いを理解することは小さく傷付け合うこと

ほんとうに簡単なサンプルアプリケーションをスナッチしただけだが、C 言語によるイベントドリブンな設計について良い知見が得られた。イベントドリブンのような Haskell で設計しにくい対象にも FFI を使って少しずつスナッチ設計をすることは可能だった。またスナッチのどの段階においても元のアプリケーションの挙動を確認することができた。完全にスナッチが終わった後は単なる Haskell プログラムとしての再設計することができた。今回はアプリケーションを再設計することで Android NDK を簡単なフレームワークで管理するようにして、新しい立方体アプリケーションにまで成長させることができた。つまり完全なスナッチが終わった後は Haskell 言語の設計効率を活用できるわけだ。もちろんスナッチ後のアプリケーションコードには強い型が付いているため、再設計コストも見積りやすい。

しかしコンピュータアーキティクチャの全てを Haskell でスナッチすることがうれしいかは疑問だ。例えばバーチャルメモリのページデーモンではコンストラクタや GC を使ってはならないことは明白だ。ということは全てを Haskell で書き直せるか？ という問題については既に反例が見つかっているわけだ。NetBSD kernel のような巨大なアプリケーションでも C 言語で記述すべき部分と Haskell で楽に設計できる部分とが同居することだろう。

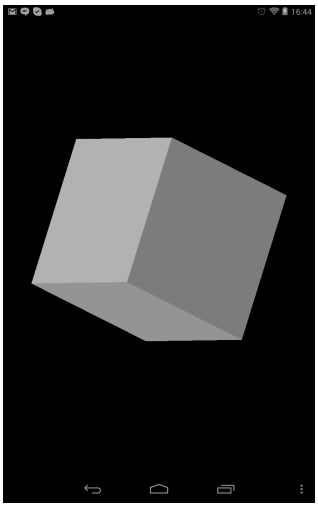


図 31.12: 立方体回転アプリ

……ひょっとすると僕たち生物も同じなのかもしれない。スナッチとは相手を理解するために相手を少し傷付けて、相手の一部を自分の内に取り込むということでもある。お互いを小さく傷付け合いながら、お互いを少しずつ理解する。それが……興味を持つとか、愛するということなのかもしれない。

僕は ThinkPad の画面を閉じた。

31.13　それでも明日はやって来る

「どうして僕の左足をスナッチしちゃったの!？」
『ごめんでゲソ! ごめんでゲソ! ついあんまりにも面白くて……』

[*15] https://play.google.com/store/apps/details?id=org.metasepi.ajhc.android.cube

「これじゃ明日大学行けないじゃないか」
『明日の朝までに絶対元に戻すでゲソ。だから心配は不要じゃなイカ？』
「そういう問題じゃぁ……もういいよ。先に寝るね。おやすみ」

もうちょっとで眠れそうだったのだが、カノジョが布団に入ってきた。僕の背中にカノジョの温もりが伝わってくる。少し震えているようだ。僕はカノジョのてのひらに小さく口づけしてあげた。

「おやすみ。また明日ね」

31.14　ソースコードライセンス

引用した Android NDK の native-activity サンプルソースコードは以下のランセンスに服します。

```
* Copyright (C) 2010 The Android Open Source Project
*
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at
*
*      http://www.apache.org/licenses/LICENSE-2.0
*
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
```

31.15　参考文献

- 『フリクリ』（FLCL） - ガイナックス・Production I.G
- 『スナッチャー』（SNATCHER） - コナミ（現・コナミデジタルエンタテインメント）
- 『ONE LIFE』（ワン・ライフ） - the pillows
- 今回の記事で途中までスナッチした Android NDK サンプルアプリケーションソースコード[*16]
- タッチパネルで立方体を回転させる Android NDK アプリケーションソースコード[*17]
- Real World Haskell – Chapter 17. Interfacing with C: the FFI [*18]
- Metasepi team meeting #6: ”Snatch-driven development” [*19]
- Metasepi team meeting #8’: Haskell apps on Android NDK [*20]

[*16] https://github.com/master-q/trysnatch_androidndk
[*17] https://github.com/ajhc/demo-android-ndk
[*18] http://book.realworldhaskell.org/read/interfacing-with-c-the-ffi.html
[*19] http://www.slideshare.net/master_q/20131020-osc-tokyoajhc
[*20] http://www.slideshare.net/master_q/20131116-osc-fukuokaajhcdroid

·S(S(KS)K)(S(S(KS)K)(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I))))))·S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))·S(S(KS)K)(S(S(KS)K)(M(S(S(KS)K)I)(S(S(KS)K)(S(S(KS)K)I))))·S(S(KS)K)(S(K(S(S(KS)K)(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))).

「簡約(かんやく)！？ λ(らむだ) カ(か)娘(むすめ) 始まりの総集編」

1 巻:「簡約(かんやく)！ λ(らむだ) カ(か)娘(むすめ)」
2 巻:「簡約(かんやく)！？ λ(らむだ) カ(か)娘(むすめ) (二期(かっこにき))」
3 巻:「簡約(かんやく)！？ λ(らむだ) カ(か)娘(むすめ) (算(さん))」
4 巻:「簡約(かんやく)！ λ(らむだ) カ(か)娘(むすめ) (4)」
5 巻:「簡約(かんやく)！？ λ(らむだ) カ(か)娘(むすめ) Go!」
6 巻:「簡約(かんやく)！？ λ(らむだ) カ(か)娘(むすめ) Rock!」

2014 年 5 月 17 日 初版発行

原作：安部真弘「侵略！ イカ娘」より
発行：参照透明な海を守る会
http://www.paraiso-lang.org/ikmsm/
hayashizaki.kitsune+ikmsm@gmail.com
著者：@xhl_kogitsune, @tanakh, @nushio, @master_q,
@dekosuke, @yryozo, @dif_engine, @ark_golgo, @fumieval
表紙：七臣ナオミ 様 http://naomi703703703.blog112.fc2.com/
口絵：尾住(オズ) 様 (1 巻表紙)
七臣ナオミ 様 (表紙別 ver,4 巻表紙)
@dif_engine (2,3,5,6 巻表紙)
挿絵：@nushio (1 巻)
印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/

参照透明な海を守る会