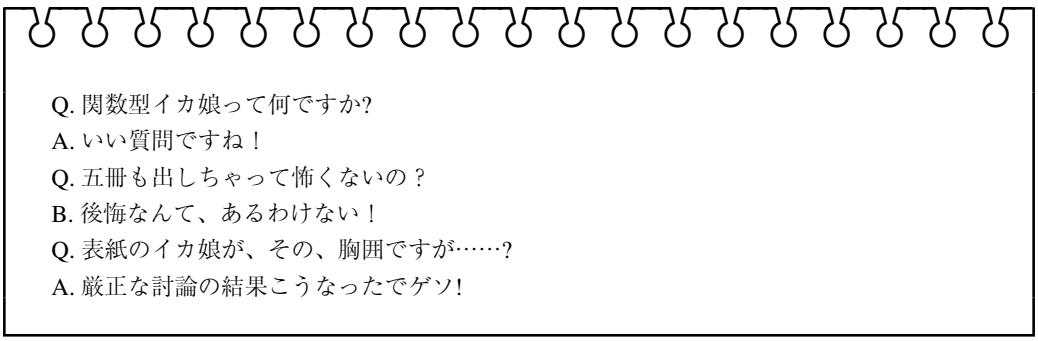

簡約!?
ヘカ娘
GO!

目次

第0章

まえがき

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. 五冊も出しちゃって怖くないの？
B. 後悔なんて、あるわけない！
Q. 表紙のイカ娘が、その、胸囲ですが……?
A. 厳正な討論の結果こうなったでゲソ!

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。それ以上のことは特にないでゲソ。

この本は、コミックマーケット 80 での「簡約! λ カ娘」、コミックマーケット 81 での「簡約!? λ カ娘 (二期)」、さらにコミックマーケット 82 での「簡約!? λ カ娘 (算)」、に続く、さらにさらにコミックマーケット 83 での「簡約!? λ カ娘 4」、に続く、五冊目の関数型イカ娘の本でゲソ。アニメ 3 期の放映が近いことを祈りつつ、関数型言語で地上を侵略しなイカ!

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。

第1章

めたせぴ☆ふぁうんでーしょん

— @master_q

1.1 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。

1.2 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

[*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

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。

「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 1.1)。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」

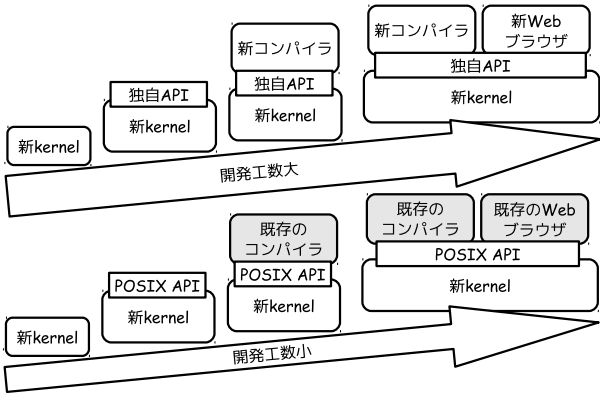


図 1.1: POSIX API を採用することで OS 開発の工数を削減

「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 1.2)。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。

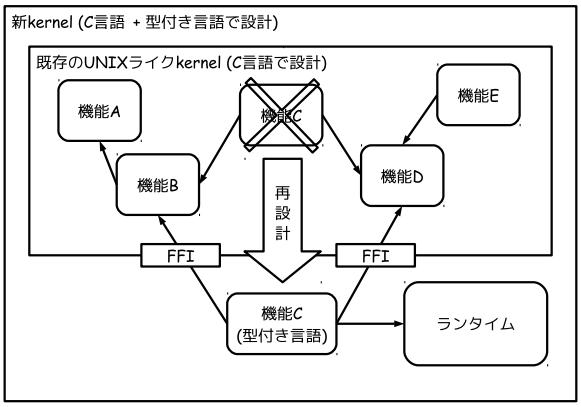


図 1.2: スナッチ設計: 既存 kernel を少しずつ設計置換

[*4] 闘うプログラマー http://www.amazon.co.jp/dp/4822247570
[*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

「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。

1.3 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……

[*6] 達人プログラマー http://www.amazon.co.jp/dp/4894712741

「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。

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 1.1: “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。

[*7] Jhc Haskell Compiler http://repetae.net/computer/jhc/

1.4 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 1.3) 。

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。

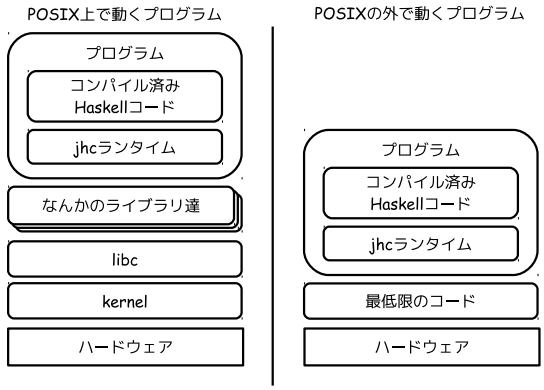


図 1.3: POSIX 内と外の Haskell プログラムの想像イメージ

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。

1.4.1 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 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。

[*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。

[*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。

```
>> 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 1.4)。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。

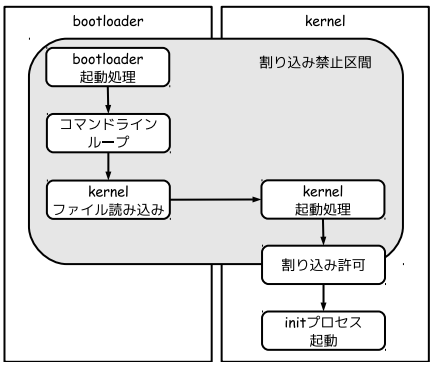


図 1.4: bootloader は割り込み禁止で実行される

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。

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 1.5)。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

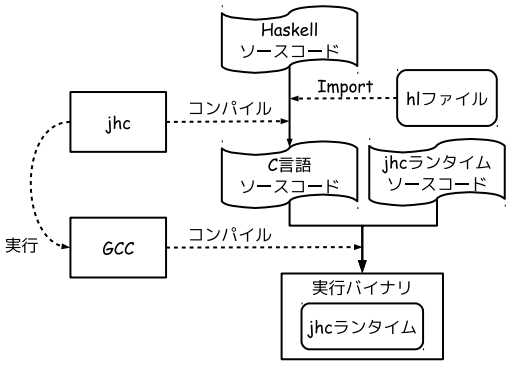


図 1.5: jhc のコンパイルフロー

[*11] https://github.com/ajhc/ajhc/tree/arafura/rts
[*12] http://ajhc.metasepi.org/manual_ja.html#オプション

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。

```
$ 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。

```
$ 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。

1.4.2 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 1.6)

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 1.7)。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。

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。

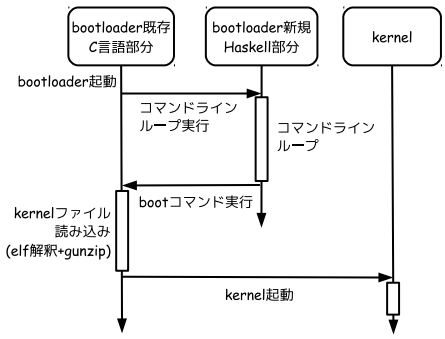


図 1.6: bootloader の一部を Haskell で再設計

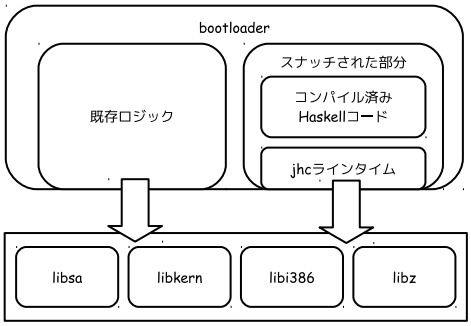


図 1.7: bootloader で使えるライブラリ

```
// 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);
}
```

[*13] https://github.com/ajhc/ajhc/blob/v0.8.0.1/rts/rts/jhc_rts.c#L164

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);
        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");
```

[*14] 最初のスケッチのソースコード https://gitorious.org/metasepi/netbsd-arafura

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。

1.5 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

[*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

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。
「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。

1.6 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

[*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`

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？

1.6.1 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。

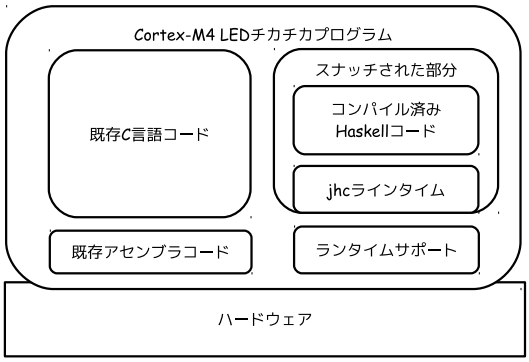


図 1.8: Corntex-M4 上のプログラムのスナッチ

1.6.2 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 1.8 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

[*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`

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。

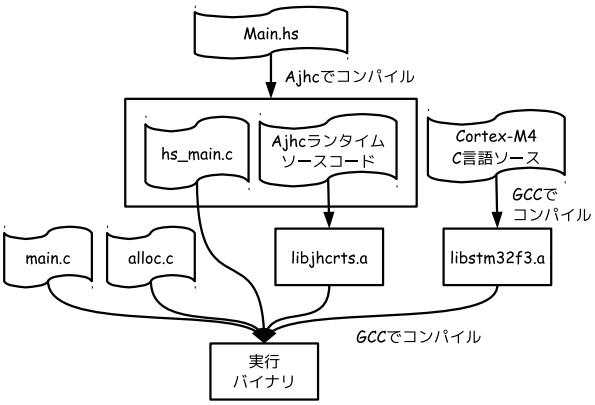


図 1.9: Corntex-M4 クロスコンパイル環境

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。

そこで図 1.9 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。

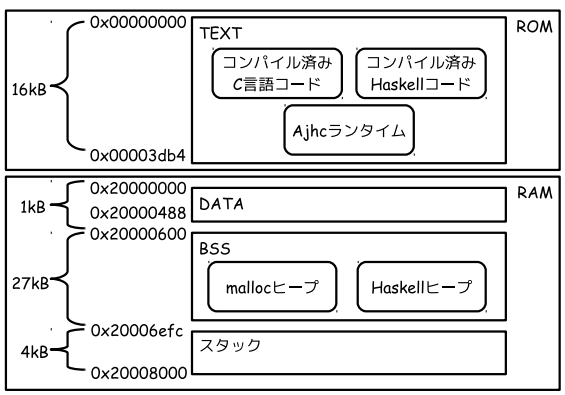
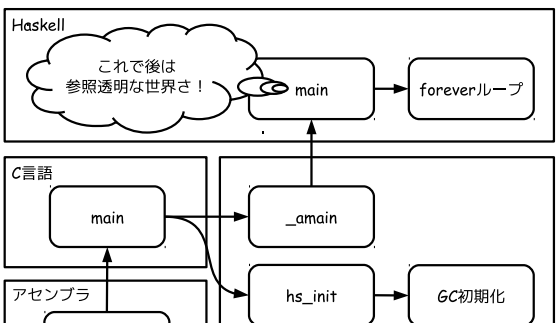


図 1.10: Cortex-M4 上で動作する実行バイナリのメモリマップ例

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 1.10 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。

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 1.11)。

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 1.12)。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。

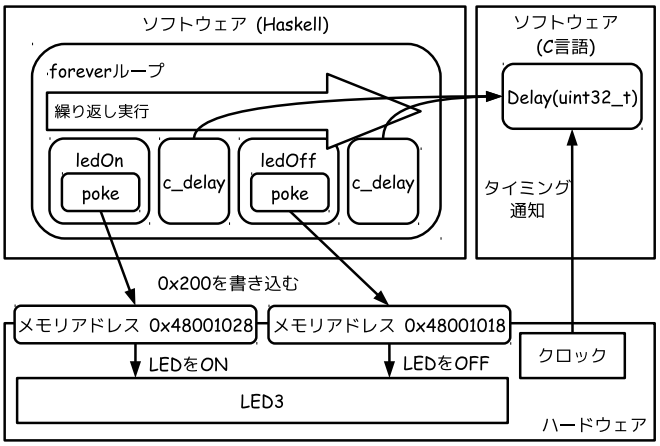


図 1.12: LED チカチカプログラムの動作

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

 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。

[*24] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Foreign-Ptr.html#v:nullPtr

```
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。

[*25] https://github.com/ajhc/demo-cortex-m3#write-haskell-code

1.6.3 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 1.13)。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。

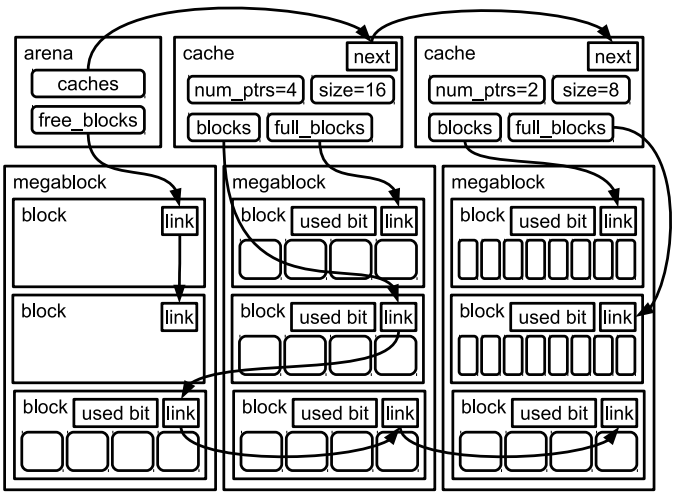


図 1.13: jgc での Haskell ヒープの管理

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

[*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

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。

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 1.14)。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。

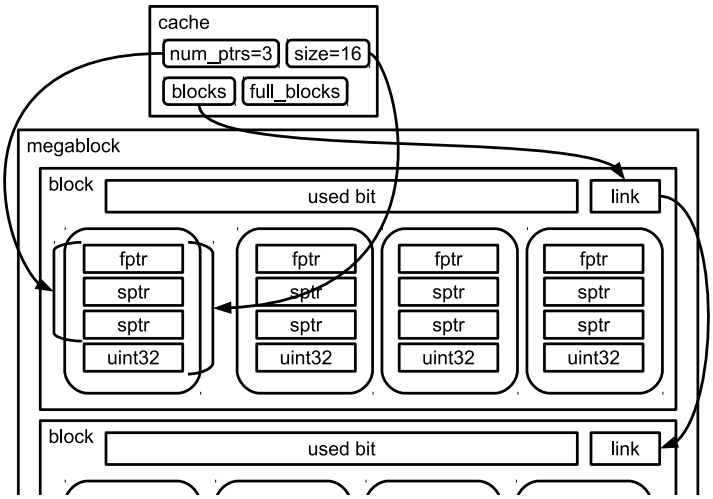


図 1.14: jgc でのチャンクの構造は cache で定義される

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。

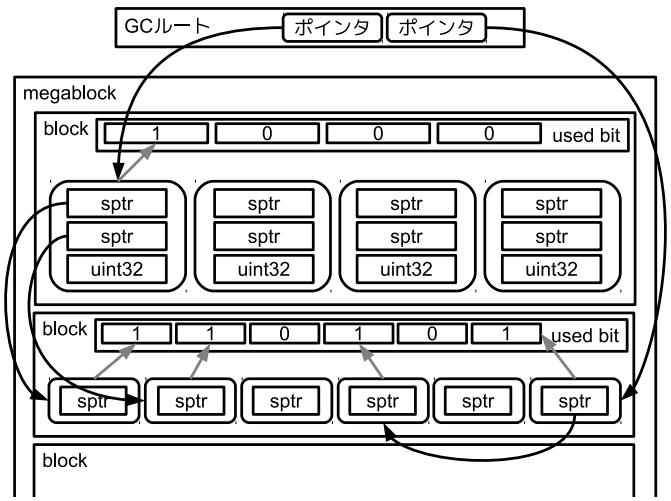


図 1.15: jgc の GC マーキング

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 1.15)。

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

[*28] http://ajhc.metasepi.org/manual_ja.html#ランタイムシステム

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

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 1.16)。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。

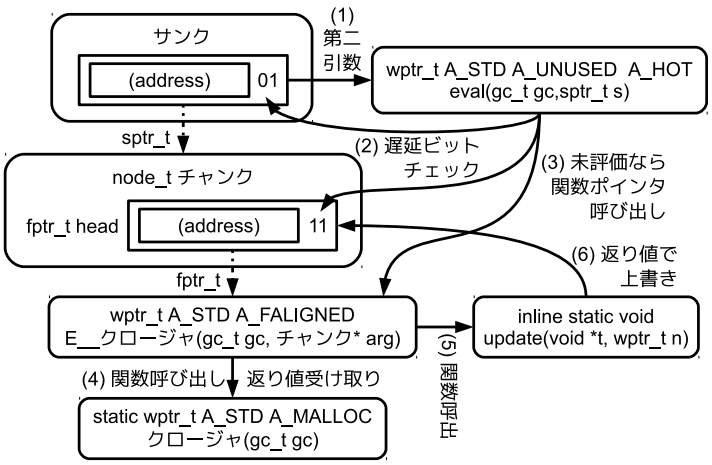


図 1.16: サンク評価の一例

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
```

[*29] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489i/CIHBJCDC.html

```
* 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。

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？

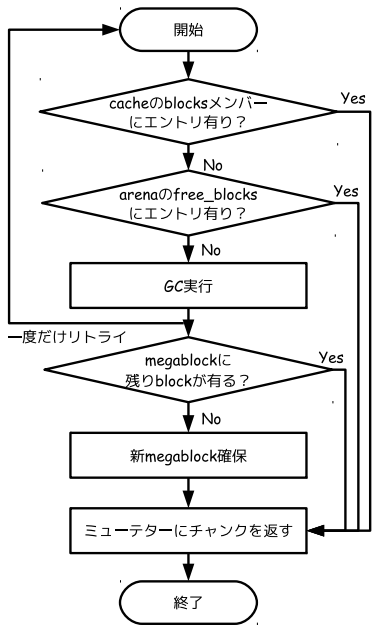


図 1.17: s_alloc 関数フローチャートと GC 後のリトライ追加

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 1.17 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。

[*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

```
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 1.18)。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。

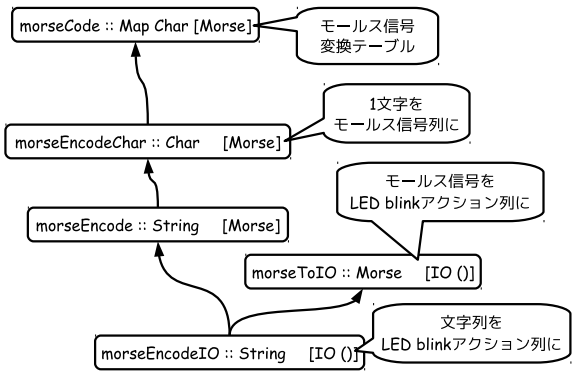


図 1.18: モールス信号 LED 表示デモの関数

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。

1.7 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 1.19 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)

[*34] http://repetae.net/computer/jhc/big-picture.pdf

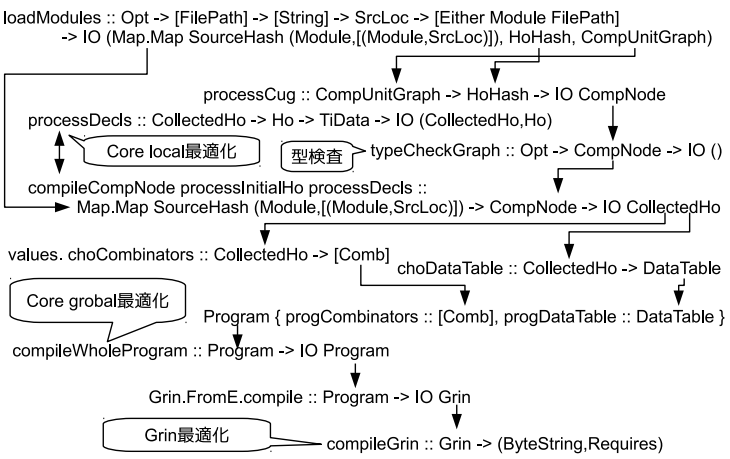


図 1.19: jhc コンパイルパイプラインの全体像

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。

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。

[*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`

```
-- 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)
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

[*38] http://ajhc.metasepi.org/manual_ja.html#型プラグマ

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？」

1.8 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 1.20 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!

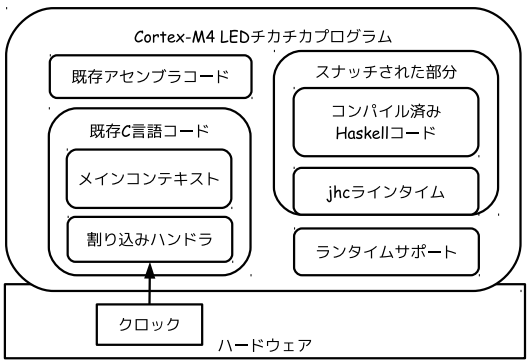


図 1.20: 二度目のスケッチでスナッチできなかった部分

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 1.21)。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。

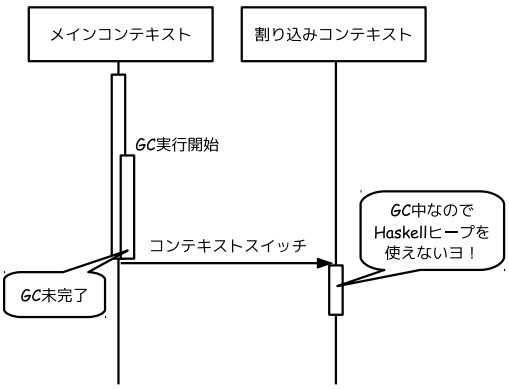


図 1.21: GC 実行中の割り込みコンテキストへのスイッチ

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。

[*39] ChibiOS/RT free embedded RTOS `http://www.chibios.org/dokuwiki/doku.php`

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。

1.8.1 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
```

[*40] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.c#L198

```
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
        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 1.22)。

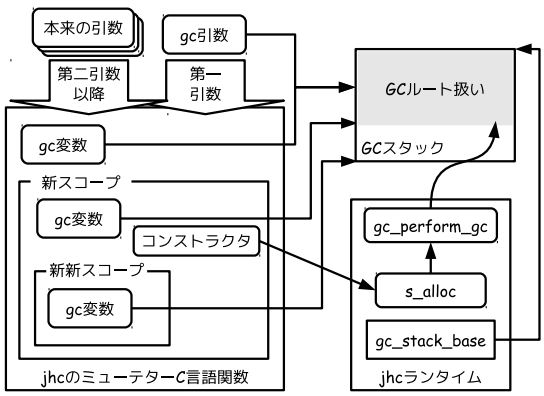


図 1.22: GC スタックとミューテータ

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。

[*41] https://github.com/ajhc/ajhc/blob/v0.8.0.4/rts/rts/gc_jgc.h#L53

1.8.2 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。

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 1.23)。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。

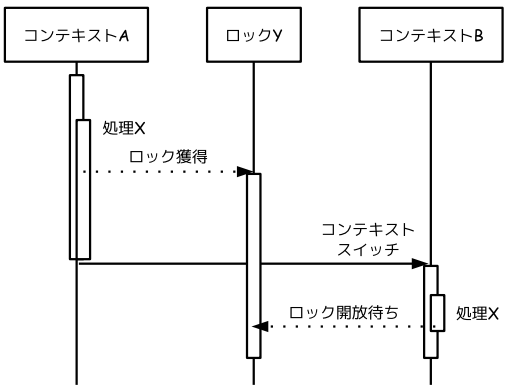


図 1.23: 割り込みと自コンテキストによる二重 lock

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

[*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

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 1.24)。

```
// 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。

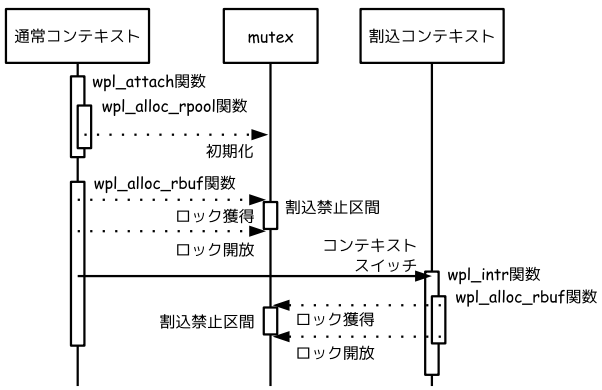


図 1.24: mutex_enter の ipl について

[*44] sigwait(2) http://netbsd.gw.com/cgi-bin/man-cgi?sigwait++NetBSD-current

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。

1.8.3 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。

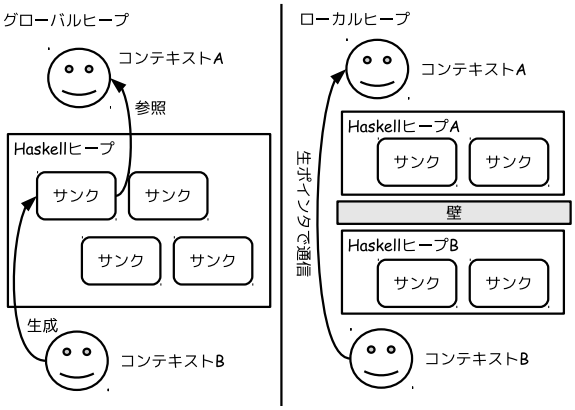


図 1.25: グローバルヒープとローカルヒープ

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 1.25)。グローバルヒープはこれまで議論

[*45] A garbage collector for C and C++ `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

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。

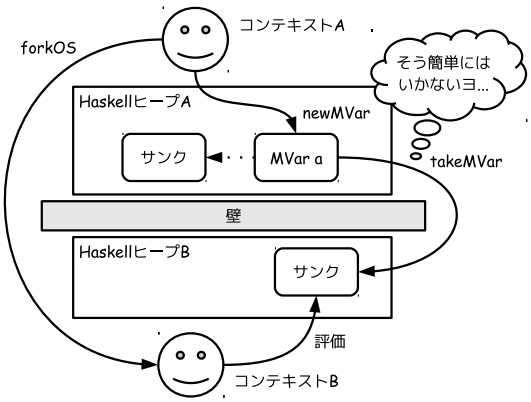


図 1.26: MVar に繋いだサンクは誰が評価すべきなのか？

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 1.26)。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 1.27)。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] 。

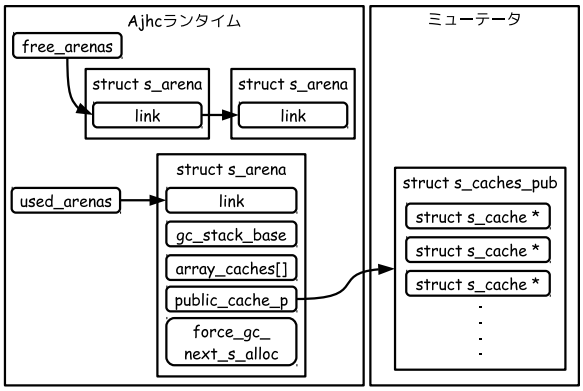


図 1.27: s_arena 構造体と GC スタックの構造と配置

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;
};
```

[*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

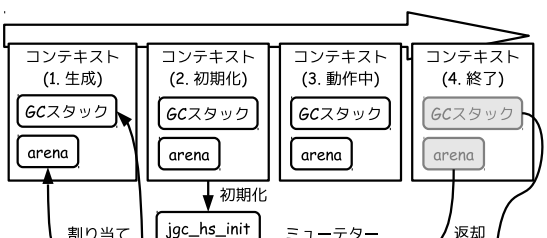
```
// --snip--
void
jhc_hs_init(gc_t gc,arena_t arena)
{
        alloc_public_caches(arena,sizeof(struct s_caches_pub));
        find_cache(&public_caches(arena)->cCJhc_Prim_Prim_$x3a,arena,
                   TO_BLOCKS(sizeof(struct sCJhc_Prim_Prim_$x3a)),2);
        find_cache(&public_caches(arena)->cFR$__fJhc_Basics_$pp,arena,
                   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 1.28)。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。

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 1.29)。

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] 。

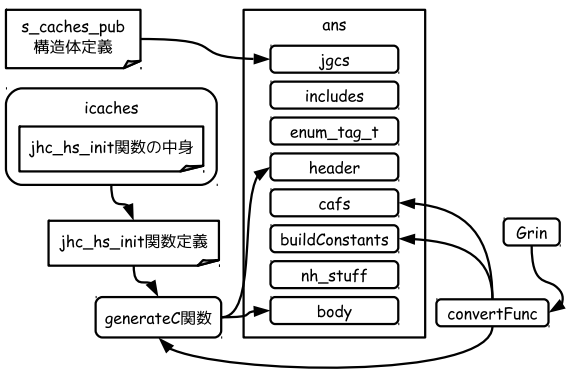


図 1.29: grin → C 変換

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]

[*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 。

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 1.30)。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。

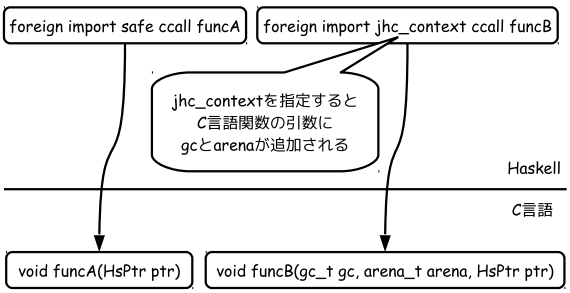


図 1.30: C 言語の関数呼び出しに gc と arena を追加する

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 。

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 1.31 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。

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 1.31: コンパイルフラグによる mutex の実装差異

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。

1.8.4 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。

*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
*58 執筆時点では Travis CI を使っている https://travis-ci.org/ajhc/ajhc

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 1.32)。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……

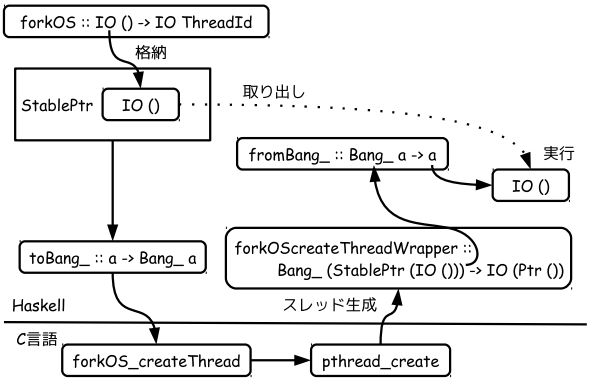


図 1.32: forkOS を pthread で実装

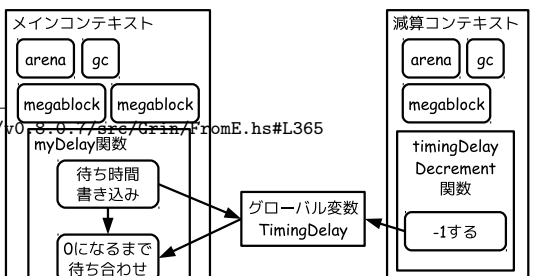
```
ajhc: Grin.FromE.compile'.ce in function: FE@.CCall.forkOScreateThreadWrapper
can't grok expression: <fromBang_ x108042543::IO ()> x62470112
```

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_ 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 1.33)。つまり割り込みコンテキストの



[*59] https://github.com/ajhc/ajhc/blob/v0.8.0.7/src/Grin/FromE.hs#L365

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 timingDelay-Decrement 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。

1.8.5 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 1.34 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められる。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。

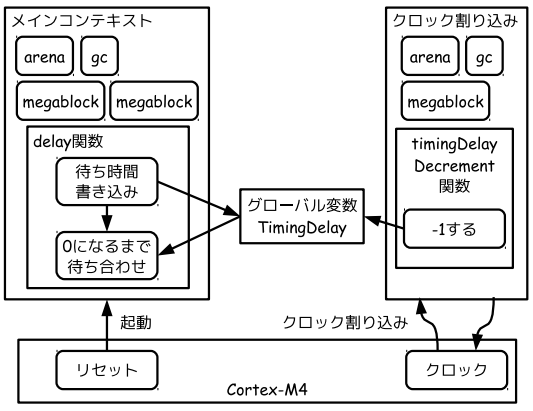


図 1.34: 割り込みハンドラを Haskell でスナッチ

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!

1.9 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。

1.10 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

[*60] https://github.com/ajhc/demo-cortex-m3/tree/master/stm32f3-discovery

- 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 1.35: ”ビンタンビール”

1.11　参考文献

- 図 1.35: 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2章

jhcコピペ

— @master_q

2.1 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ん！！！ あぁあああああ…ああ…あっあっー！ 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あ！！ はぁあああああん！！ 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! sssssaaahssssssaaah.. 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!! Halkegini-aaaaaa!! 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! Yesssss! 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!

2.2 参考文献

- ルイズコピペ `http://dic.pixiv.net/a/`**ルイズコピペ**

第3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

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。

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。

3.1　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……」

3.2　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！》「▂▅▇▓▒(ˊωˋ)▒▓▇▅▂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**かたもるふゐずむ**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**りすとかたもるふゐずむ** $h \in A* \to B$ 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**かたもるふゐずむ**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**かたもるふゐずむ**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**ふゐるたー**正志是皆**かたもるふゐずむ**の一族なり

$$\begin{aligned} \mathtt{length} &= (\!|0, \oplus|\!) \quad \text{where} \quad a \oplus n = 1 + n \\ \mathtt{filter}\ p &= (\!|\mathtt{Nil}, \oplus|\!) \\ &\quad \text{where} \quad a \oplus as \\ &\qquad\qquad \mid \quad p\ a = \mathtt{Cons}(a, as) \\ &\qquad\qquad \mid \neg p\ a = as \end{aligned}$$

再歸關數を手ずからに實裝せずして**かたもるふゐずむ**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」

3.3　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ぃ？」

「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」

$$(\!|\varphi|\!)_{\mathrm{F}} = \mu(\varphi \xleftarrow{\mathrm{F}} \mathtt{out}) \tag{12}$$

$$[\!(\psi)\!]_{\mathrm{F}} = \mu(\mathtt{in} \xleftarrow{\mathrm{F}} \psi) \tag{13}$$

$$[\![\varphi, \psi]\!]_{\mathrm{F}} = \mu(\varphi \xleftarrow{\mathrm{F}} \psi) \tag{14}$$

$$\{\!|\xi|\!\}_{\mathrm{F}} = \mu(\lambda \mathtt{f}.\xi \circ (\mathtt{id} \triangle \mathtt{f})_{\mathrm{F}} \circ \mathtt{out}) \tag{15}$$

「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！」

3.4 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、かたもるふぬずむ喝鈷（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4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。

4.1 プロローグ

4.1.1 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。

4.1.2　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——。

4.1.3 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

[*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。

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——。

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。

4.2　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。

4.2.1 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

$$\boxed{g \circ f(x) = g(f(x))}$$

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

$$\boxed{X \xrightarrow{f} Y \text{ かつ } Y \xrightarrow{g} Z \text{ ならば } X \xrightarrow{g \circ f} Z}$$

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。

4.2.2　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

$$\boxed{\left\{ \langle x, f(x) \rangle \,\middle|\, x \in X \right\} \ (\subseteq X \times Y)}$$

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：

$$\boxed{\textbf{(func)} \quad \forall x \in X \ \ \exists! y \in Y \ \ \langle x, y \rangle \in f.}$$

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：

$$\boxed{\begin{array}{c}\boxed{x\text{に対して、}Y\text{の元}y\text{が1つだけ存在して、}\langle x,y\rangle \in f\text{となる}}\\ \uparrow \\ X\text{の全ての元}x\text{について！}\end{array}}$$

……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？」

4.2.3 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

$$\boxed{\mathrm{Map}(X,Y) := \left\{\ X \text{を定義域とし} Y \text{を値域とする関数}\ \right\}}$$

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」

4.2.4 $\mathrm{Map}(X,Y)$ 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——」

$$\boxed{f \in \mathrm{Map}(X,Y),\ g \in \mathrm{Map}(Y,Z) \quad \Longrightarrow \quad g \circ f \in \mathrm{Map}(X,Z).}$$

「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」

4.2.5 $\mathrm{Map}(\varnothing,Y) = ?$

「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

$$\boxed{\begin{array}{l} t \in \varnothing \times Y \\ \quad \Leftrightarrow \text{なんかの } t_1 \text{ と } t_2 \text{ によって } t = \langle t_1, t_2 \rangle \text{ と表せて、しかも } \langle t_1, t_2 \rangle \in \varnothing \times Y \text{ となってる} \end{array}}$$

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

$$\boxed{\forall t \ \ [t \notin \varnothing \times Y]}$$

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

$$\boxed{f \in \mathrm{Map}(\varnothing, Y) \Rightarrow f \subseteq \varnothing \times Y = \varnothing}$$

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

$$\boxed{\mathrm{Map}(\varnothing, Y) \subseteq \{\varnothing\}}$$

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.76) から」

$$\boxed{\varnothing \in \mathrm{Map}(\varnothing, Y) \Leftrightarrow \forall x \in \varnothing \ \ \exists! y \in Y \ \ \langle x, y \rangle \in \varnothing}$$

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

$$\boxed{\forall x \in \varnothing \quad A(x)}$$

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

$$\boxed{\forall x \in \varnothing \quad B(x)}$$

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。

4.3 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。

4.3.1 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

$$\boxed{V = \{X_i \mid i \in I\}}$$

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。

4.3.2 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

$$f \colon X \to Y \quad \text{または} \quad X \xrightarrow{f} Y$$

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 $\mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ$ 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『$\circ$』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 $f\colon X \to Y_1$ で $g\colon Y_2 \to Z$ 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」

「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

$$\boxed{\mathrm{id}_X\colon X \to X}$$

となってるはずですね。次に、条件 **(cat5)** を考えると、関数 $f\colon X \to Y$ 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

$$\boxed{\begin{array}{rcl}\mathrm{id}_X\colon X & \longrightarrow & X \\ x & \longmapsto & x\end{array}}$$

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――」

[*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。

「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 4.1)：

圏 $\mathbf{Ens}_V$ の言葉	集合論の言葉
対象 X	集合 $X\,(\in V)$
射 $f\colon X \to Y$	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 $\mathrm{id}_X = \{\langle x, x\rangle \mid x \in X\}$
$f\colon X \to Y$ と $g\colon Y \to Z$ の合成射 $g \circ f$	合成関数 $g \circ f\colon X \to Z$
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 4.1: 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」

4.3.3 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 4.2)：

「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」

[*3] CWM p.12, (訳書では p.14)

圏 **Hask** の言葉	Haskell の言葉
対象 X	型 `X`
射 $f: X \to Y$	関数 `f :: X -> Y`
対象 X に対応する恒等射 id_X	関数 `id :: X -> X`
$f: X \to Y$ と $g: Y \to Z$ 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 4.2: Haskell の圏 **Hask** と Haskell の言葉の対応 (1)

「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

$$
\begin{aligned}
&\texttt{Int} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->Int)} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->(Int->Int))} \in \mathrm{Ob}(\mathbf{Hask})\\
&\texttt{(Int->(Int->(Int->Int)))} \in \mathrm{Ob}(\mathbf{Hask})\\
&\cdots
\end{aligned}
$$

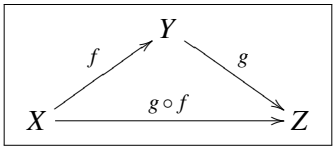
という型の系列があって、この系列だけでも、すでに無限個の型が含まれてるよね」

「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」

4.3.4 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。

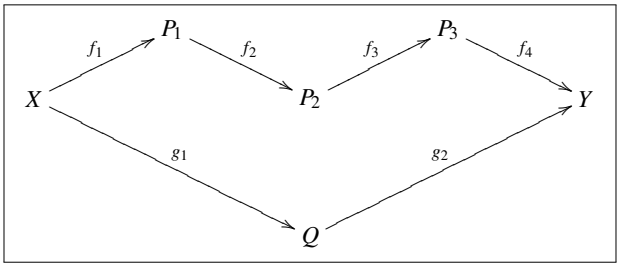
「さて、例えば射の合成の様子を書くときに射の矢印が斜めでもいいことにすれば



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」

[*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

「なるほどなぁ」

4.4 関手

4.4.1 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.83 で定義されてたやつですね。要するに

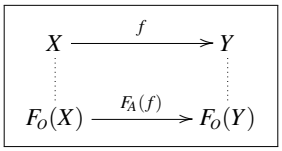
$$\langle g, f \rangle \in \mathrm{Comp}\big(\mathrm{Arr}(\mathbf{C})\big) \iff g \circ f \text{ が定義できる}$$

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」

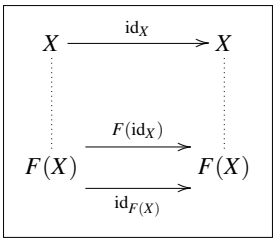
「なるほど、習うより慣れろってわけですね。じゃあまず条件 **(func1)** からやってみますか」

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」

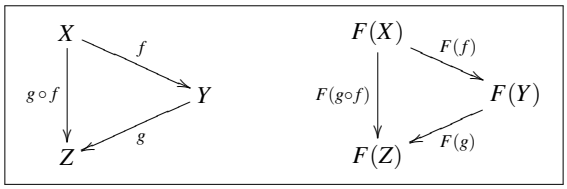
「なるほど、話を戻すと条件 **(func2)** は、今の図式が可換になる事だと言い換えられるわけですね！」

「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：

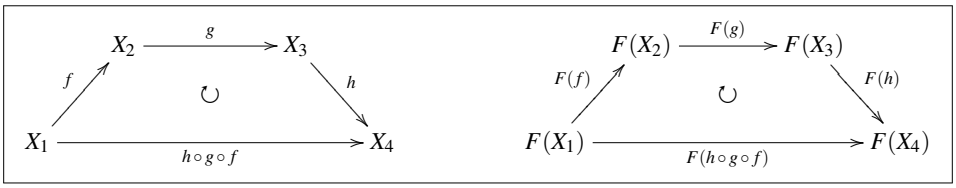


「射の等式 $F(g)\circ F(f)=F(g\circ F)$ が成立する事が右の図式の可換性として表現されてるね。今の図式だと、右側の可換図式は『左側の可換図式を関手 F で移したもの』と思えるよね」

「……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.88) 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 4.3)：

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 4.3: 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」

4.4.2 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 $\mathbf{C}, \mathbf{D}, \mathbf{E}$ 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：

$$\mathbf{C} \xrightarrow{F} \mathbf{D} \xrightarrow{G} \mathbf{E}$$

「――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 $\mathbf{C}$ 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

(assoc) $(H\cdot G)\cdot F = H\cdot(G\cdot F)$

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……

(funcexp1) $F^1 := F.$
(funcexp2) $F^{n+1} := F\cdot F^n \qquad (^\forall n > 0).$

――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

$$\forall m \in \mathbb{N}\left[\forall n \in \mathbb{N} \qquad F^m \cdot F^n \;=\; F^{m+n}\right].$$

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

$$\forall n \in \mathbb{N} \qquad F^1 \cdot F^n \;=\; F^{1+n}$$

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

$$\forall n \in \mathbb{N} \qquad F^{m_0} \cdot F^n \;=\; F^{m_0+n}$$

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

$$\begin{aligned}
&F^{m_0+1} \cdot F^n = (F \cdot F^{m_0}) \cdot F^n && :\textbf{(funcexp2)} \text{ より} \\
&= F \cdot (F^{m_0} \cdot F^n) && :\textbf{(assoc)} \text{ より} \\
&= F \cdot F^{m_0+n} && : \text{帰納法の仮定より} \\
&= F^{m_0+n+1} && :\textbf{(funcexp2)} \text{ より} \\
&= F^{(m_0+1)+n}.
\end{aligned}$$

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) $I_A : f \longmapsto f.$

[*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。

「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——。

$$
\begin{aligned}
&(m = m_0 + 1 \text{ のときの左辺}) \\
&= \left(F^{m_0+1}\right)_A \left(\mathrm{id}_{F^n(X)}\right) \\
&= \left(F^{m_0} \cdot F^1\right)_A \left(\mathrm{id}_{F^n(X)}\right) \\
&= \left(\left(F^{m_0}\right)_A \circ \left(F^1\right)_A\right) \left(\mathrm{id}_{F^n(X)}\right) && : \textbf{(fcomp2)} \text{ より} \\
&= \left(F^{m_0}\right)_A \left(\left(F^1\right)_A \left(\mathrm{id}_{F^n(X)}\right)\right) \\
&= \left(F^{m_0}\right)_A \left(\mathrm{id}_{F^1 \cdot F^n(X)}\right) && : \textbf{(func2)} \text{ より} \\
&= \left(F^{m_0}\right)_A \left(\mathrm{id}_{F^{n+1}(X)}\right) \\
&= \mathrm{id}_{F^{m_0+(n+1)}} = \mathrm{id}_{F^{(m_0+1)+n}} && : \text{帰納法の仮定より} \\
&= (m = m_0 + 1 \text{ のときの右辺})
\end{aligned}
$$

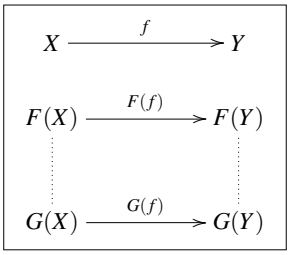
となって確かに示せました」

「よくできました」

4.5　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

$$\theta : F \xrightarrow{\bullet} G$$

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

$$\boxed{F(X) \xrightarrow{\theta_X} G(X)}$$

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：

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \theta_X}\downarrow & \circlearrowleft & \downarrow{\scriptstyle \theta_Y} \\ G(X) & \xrightarrow[G(f)]{} & G(Y) \end{array}$$

「——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——

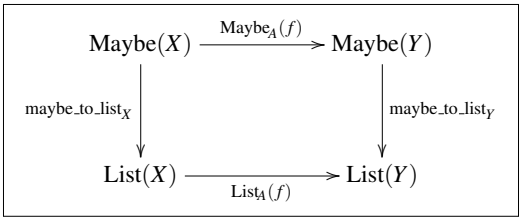
```
maybe_to_list :: Maybe a -> [a]
```

——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」

$$\begin{array}{ccc} \mathrm{Maybe}\ X & \xrightarrow{\mathrm{fmap}\ f} & \mathrm{Maybe}\ Y \\ {\scriptstyle \mathrm{maybe\_to\_list}}\downarrow & & \downarrow{\scriptstyle \mathrm{maybe\_to\_list}} \\ [X] & \xrightarrow[\mathrm{fmap}\ f]{} & [Y] \end{array}$$

「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

$$\mathrm{id}\colon I \xrightarrow{\bullet} I$$

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」

4.6 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 $f\colon \mathbb{R} \to \mathbb{R}$ 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

[*8] 実数 x が関数 $f\colon \mathbb{R} \to \mathbb{R}$ 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。

```
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]
```

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

$$\begin{array}{rcl} \mathrm{curry}\colon \mathrm{Map}(A\times B, C) & \longrightarrow & \mathrm{Map}\big(A, \mathrm{Map}(B,C)\big) \\ f & \longmapsto & \mathrm{curry}(f) \end{array}$$

を適用したものはこんな形になるよね」

$$\mathrm{curry}(\mathrm{function\_with\_state})\colon X \to \mathrm{Map}\big(S, Y\times S\big)$$

「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

$$\begin{aligned} &\mathrm{State}^S_O(X) := \mathrm{Map}(S, X\times S), \\ &\mathrm{State}^S_A(f)\colon \mathrm{Map}(S, X\times S) \longrightarrow \mathrm{Map}(S, Y\times S) \end{aligned}$$

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」

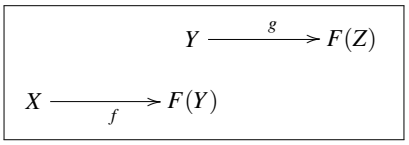
4.6.1 抽象化された計算とその合成

「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」

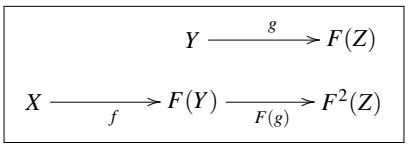
「例えばどんな場合ですか？」

「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、

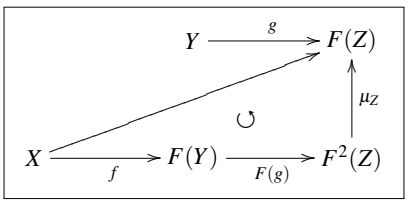


みたいのができます。あれ……これを合成しても $X \to F^2(Z)$ にしかならないですね。うまく行きそうだとおもったんですけど」

「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

$$\mu : F^2 \xrightarrow{\bullet} F$$

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 $f : X \to F(Y)$ と射 $g : Y \to F(Z)$ の『合成』として

$$\mu_Z \circ F(g) \circ f : X \to F(Z)$$

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

$$\begin{array}{ccc} \mathrm{Hom}(Y,F(Z)) \times \mathrm{Hom}(X,F(Y)) & \longrightarrow & \mathrm{Hom}(X,F(Z)) \\ \langle g, f \rangle & \longmapsto & \mu_Z \circ F(g) \circ f \end{array}$$

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 $\mu : F^2 \xrightarrow{\bullet} F$ 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

$$\eta : I \xrightarrow{\bullet} F$$

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：

$$\textbf{(eta-nat-1)}\ \forall X \in \mathrm{Ob}(\mathbf{C}) \quad \eta_X \in \mathrm{Hom}(X, F(X)),$$
$$\textbf{(mu-nat-1)}\ \forall X \in \mathrm{Ob}(\mathbf{C}) \quad \mu_X \in \mathrm{Hom}\big(F^2(X), F(X)\big).$$

このとき、集合 $\overline{O}, \overline{A}$ を次のように定義する：

$$\overline{O} := \mathrm{Ob}(\mathbf{C}),$$
$$\overline{A} := \bigcup_{X,Y \in \mathrm{Ob}(\mathbf{C})} \mathrm{Hom}(X, F(Y)) \left(= \big\{ f \,\big|\, \exists X, Y \in \mathrm{Ob}(\mathbf{C})\ \ f \in \mathrm{Hom}(X, F(Y)) \big\}\right).$$

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

$$\mathrm{Comp}(\overline{A}) := \bigcup_{X,Y,Z \in \mathrm{Ob}(\mathbf{C})} \mathrm{Hom}\big(Y, F(Z)\big) \times \mathrm{Hom}\big(X, F(Y)\big) \left(= \left\{ \langle g, f \rangle \in \overline{A} \times \overline{A} \;\middle|\; \overline{\mathrm{dom}}(g) = \overline{\mathrm{cod}}(f) \right\}\right)$$

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)** $\eta\colon I \xrightarrow{\bullet} F$　(η が自然変換であること)
- **(K2)** $\mu\colon F^2 \xrightarrow{\bullet} F$　(μ が自然変換であること)
- **(K3)** $\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X, F(Y))$　　$\eta_Y \ \bar{\circ}\ f = f.$　(η_Y の左単位条件)
- **(K4)** $\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X, F(Y))$　　$f \ \bar{\circ}\ \eta_X = f.$　(η_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) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y) \qquad F(f) \circ \eta_X = \eta_Y \circ f$

(M2) $\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \forall f \in \mathrm{Hom}(X,Y) \qquad F(f) \circ \mu_X = \mu_Y \circ F^2(f)$

(M3) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad \mu_X \circ F(\eta_X) = \mathrm{id}_{F(X)}.$

(M4) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad \mu_X \circ \eta_{F(X)} = \mathrm{id}_{F(X)}.$

(M5) $\forall X \in \mathrm{Ob}(\mathbf{C}) \qquad \mu_X \circ F(\mu_X) = \mu_X \circ \mu_{F(X)}.$

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、$\mathrm{Kleisli}(\mathbf{C}) := \langle \overline{O},\overline{A},\overline{\mathrm{dom}},\overline{\mathrm{cod}},\eta,\bar{\circ}\rangle$ 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」

4.6.2　条件 (K1) が (M1) を導くこと

「条件 **(K1)** は $\eta : I \overset{\bullet}{\longrightarrow} F$ 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」

4.6.3　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

$$\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,Y) \qquad \mu_Y \circ F^2(f) = F(f) \circ \mu_X$$

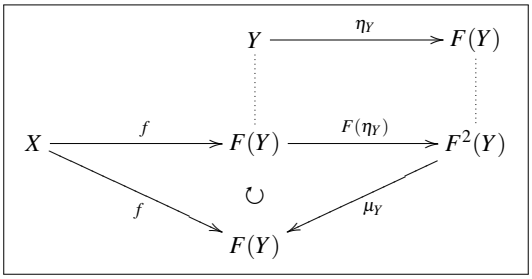
が出てきて、確かに条件 **(K2)** から条件 **(M2)** が導けます。あれ……なんかひょっとしてすごく簡単？」

4.6.4　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

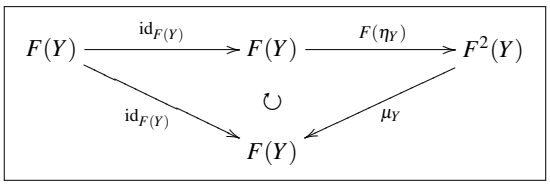
$$\forall X,Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X,F(Y)) \qquad \eta_Y \ \bar{\circ}\ f = f$$

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

$$\boxed{\mu_Y \circ F(\eta_Y) = \mathrm{id}_{F(Y)}}$$

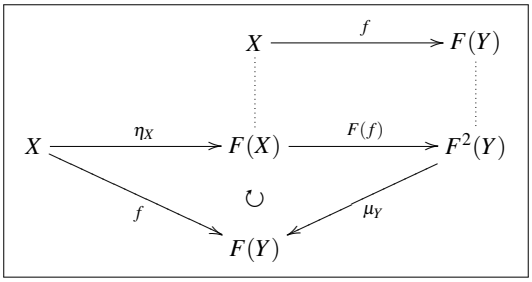
となる。よって、確かに条件 **(K3)** から条件 **(M3)** が導けたね」

4.6.5 条件 (K4) が (M4) を導くこと

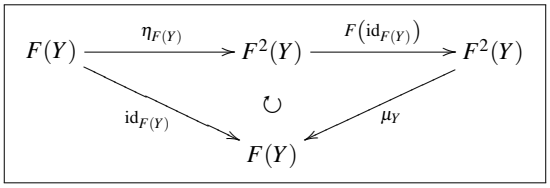
「条件 **(K4)** から条件 **(M4)** を導くところをやってみましょう。まず、もう一度条件 **(K4)** を書いてみると

$$\boxed{\forall X, Y \in \mathrm{Ob}(\mathbf{C})\ \ \forall f \in \mathrm{Hom}(X, F(Y)) \qquad f \circ \eta_X = f}$$

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.93) 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」

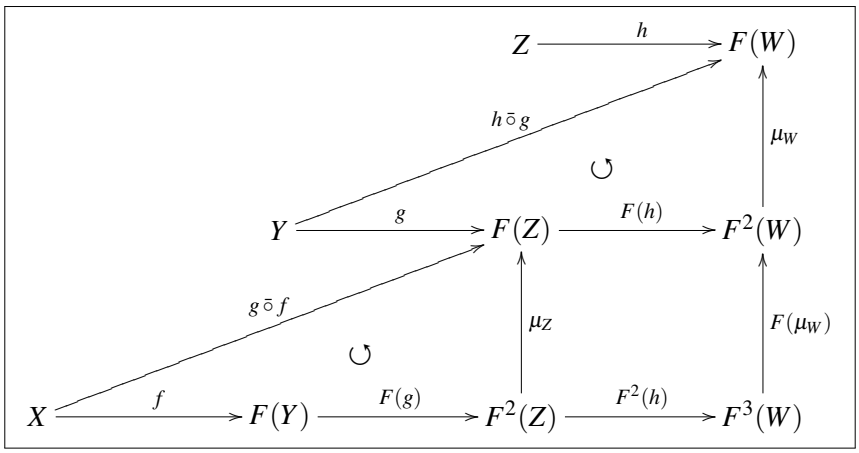
4.6.6　条件 (K5) が (M5) を導くこと

「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

$$\boxed{\forall \langle h,g\rangle, \langle g,f\rangle \in \mathrm{Comp}(\overline{A}) \qquad h \mathbin{\bar{\circ}} (g \mathbin{\bar{\circ}} f) = (h \mathbin{\bar{\circ}} g) \mathbin{\bar{\circ}} f.}$$

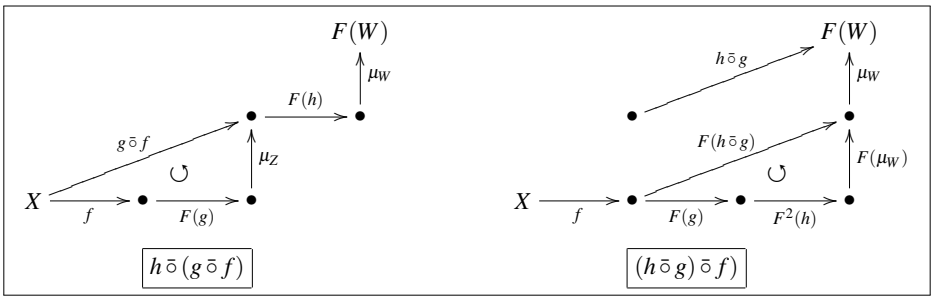
だけど、この条件は次の図式の中に埋め込まれてる」



「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、$h \bar{\circ} (g \bar{\circ} f)$ と $(h \bar{\circ} g) \bar{\circ} f$ 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」

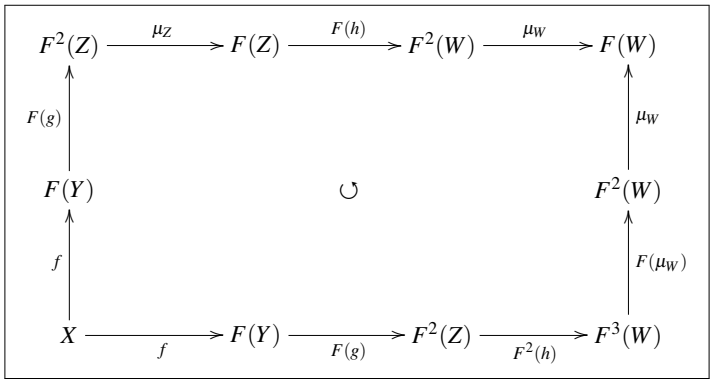
「なるほど、全体の図式はちゃんと見せてもらったし、確かに経路がわかりやすい気がします」

「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、$h \bar{\circ} (g \bar{\circ} f)$ と $(h \bar{\circ} g) \bar{\circ} f$ 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……」

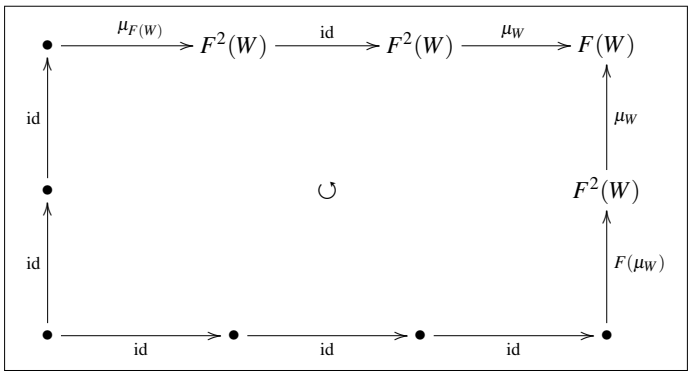
「じゃあ上の図式から $h \circ (g \circ f)$ と $(h \circ g) \circ f$ の部分を拾ってきて展開して可換図式にしちゃおう」そう言って私は次のような可換図式を書いた。



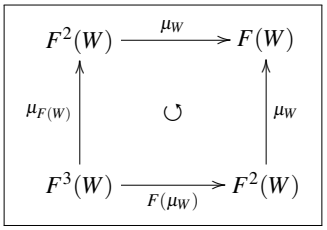
「翼さん、この続きやらせてください。

$$X := F^3(W),\ Y := F^2(W),\ Z := F(W),$$
$$f := \mathrm{id}_{F^3(W)},\ g := \mathrm{id}_{F^2(W)},\ h := \mathrm{id}_{F(W)}$$

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.93) 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？」

4.6.7　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 4.4: 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。

4.7　カンザスでないどこか

4.7.1　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——」

[*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。

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——。

4.7.2　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。

4.8　もう一つの世界

4.8.1　◆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。

4.8.2　◆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。

4.8.3 ◆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。

4.9 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。

4.10　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。

4.10.1　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。

4.10.2　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5章

λカ娘探索？

— @ark_golgo

5.1 プロローグ

図 5.1: 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 5.1)。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。

5.2 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 5.2)。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。

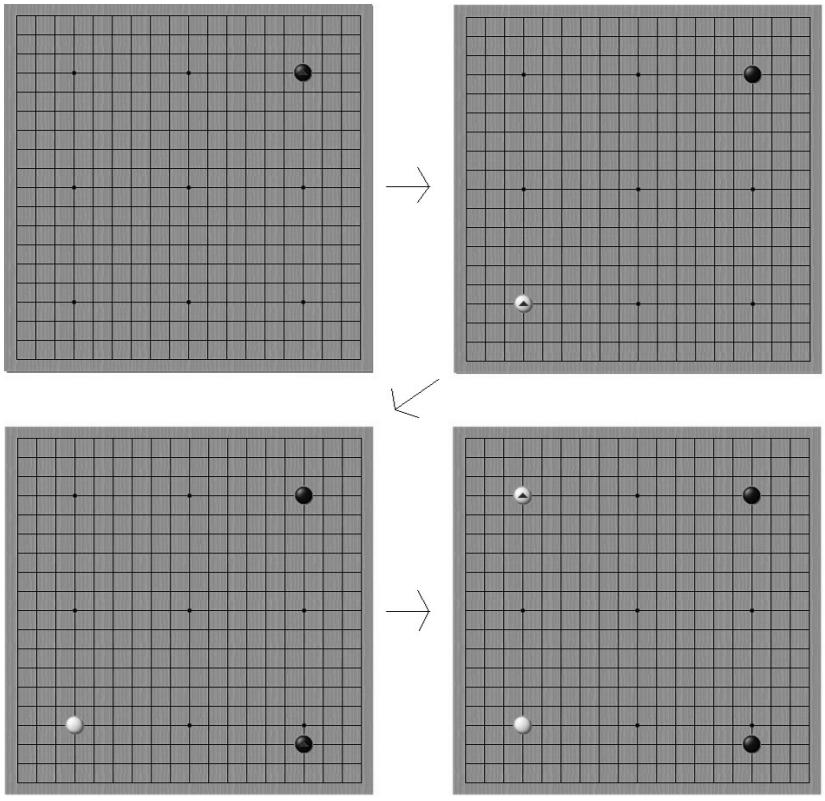


図 5.2: 交互に打つ

2. 相手の石を囲むと取れる (図 5.3。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。

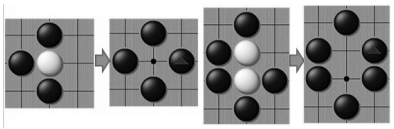


図 5.3: 石を取る

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 5.4)(『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)。

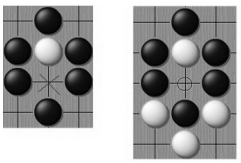


図 5.4: 白は×には打てないが○には打てる

4. 相手より大きな陣地を囲えば勝ち (図 5.5)。

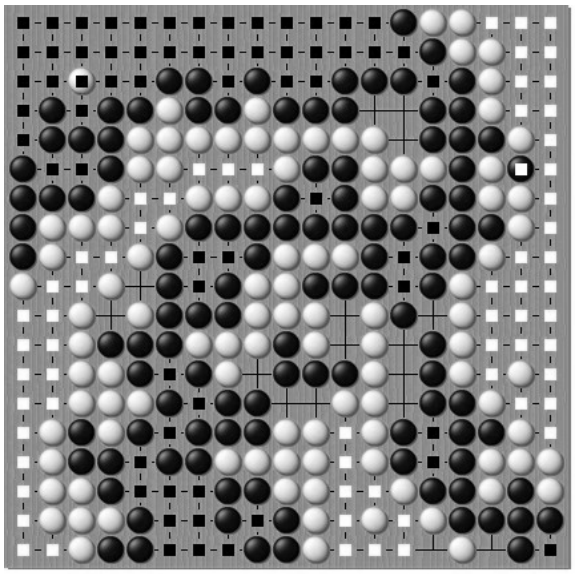


図 5.5: ■が黒の陣地、□が白の陣地。黒 3 目勝ち

5.3 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 5.6 のようになるでゲソ! 」

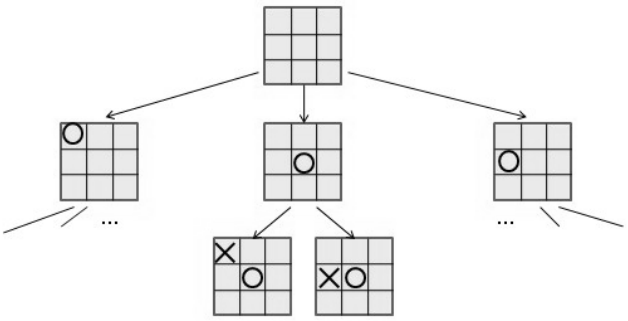


図 5.6: Tic-Tac-Toe(三目並べ) の探索の一部

「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! 」

5.4 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」

「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 ！」

5.5 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。

5.6 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 5.7、5.8、5.9 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。

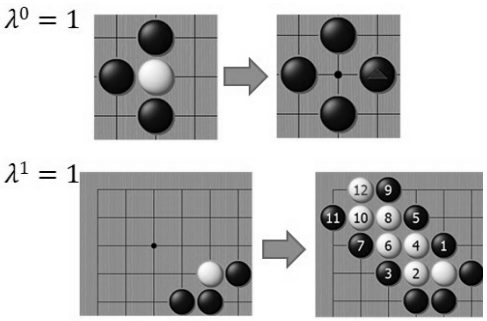


図 5.7: 取れる場合

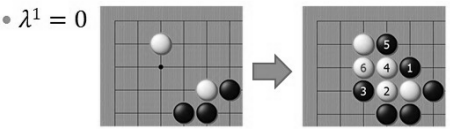


図 5.8: 取れない場合

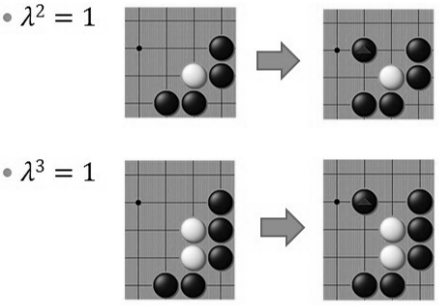


図 5.9: 相手が 2 回,3 回パスをすると取れるような手の方が、
相手が 1 回,2 回パスをしただけで取れる手より有効な場合もある。

- 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 5.10 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!」

- $\lambda^n = 1 \Rightarrow \lambda^i = 1(n \leq i)$
- $\lambda^n = 0 \Rightarrow \lambda^j = 0(1 \leq j \leq n)$

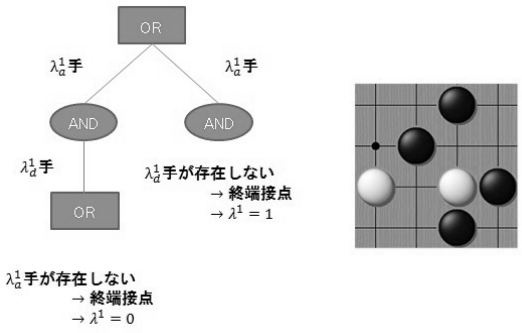


図 5.10: 探索の例

5.7 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、

$$\mathrm{pn}^l(n) = \mathrm{pn}^{l-1}(c_{l-1}) + \mathrm{pn}^l(c_l) \quad \text{(合計戦略) または}$$
$$\mathrm{pn}^l(n) = \mathrm{pn}^l(c_l) \quad \text{(最高次戦略)}$$
$$\mathrm{dn}^l(n) = \min\left(\mathrm{dn}^{l-1}(c_{l-1}), \mathrm{dn}^l(c_l)\right)$$
$$\mathrm{pn}^i(c_i) = \sum_{j=1}^{k} \mathrm{pn}^i(n_j)$$
$$\mathrm{dn}^i(c_i) = \min_{j=1,\ldots,k} \mathrm{dn}^i(n_j)$$

- (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。

5.8 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! 」

5.9 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!」

5.10 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 5.1 は@dif_engine さんに書いていただきました。

第6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。

6.1 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」

「ぇ? 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!」

[*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

「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……」
「……」
　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。

6.2　構文解析と文脈自由言語

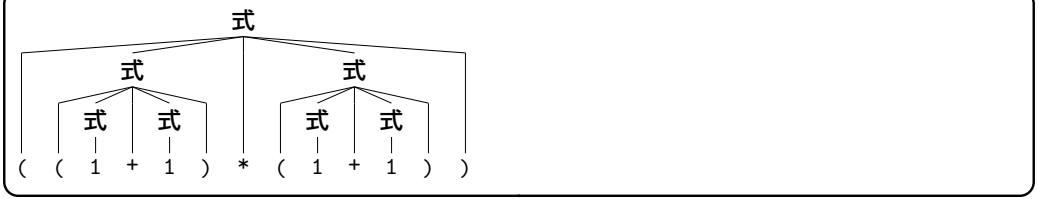
「とりあえず、分かっているとは思うけど基本的なことを確認していくわね」
「はい、分かっていてもコヨネの声で解説を聞きたい、です」
「……たとえば式

```
((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」

6.3 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」

[*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)。

$$
\begin{aligned}
\text{文脈自由文法 } G_{\text{expr}} &= (\Sigma_{\text{expr}}, N_{\text{expr}}, P_{\text{expr}}, S_{\text{expr}}) \\
\Sigma_{\text{expr}} &= \{\texttt{1}, \texttt{+}, \texttt{(}, \texttt{)}\} \\
N_{\text{expr}} &= \{S, E\} \\
P_{\text{expr}} &= \{\ S \to E, \\
&\qquad E \to E\texttt{+(}E\texttt{)}, \\
&\qquad E \to \texttt{1}\} \\
S_{\text{expr}} &= S
\end{aligned}
$$

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、

$\underline{\texttt{1}}\texttt{+(1)+(1)}$
$\to E\texttt{+(}\underline{\texttt{1}}\texttt{)+(1)}$
$\to \underline{E\texttt{+(}E\texttt{)}}\texttt{+(1)}$
$\to E\texttt{+(}\underline{\texttt{1}}\texttt{)}$
$\to \underline{E\texttt{+(}E\texttt{)}}$
$\to \underline{E}$
$\to S$

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)

```
      E
      |
構文木: 1   ▷ +(1)+(1)
```

↓ shift します。

状態 3

スタック: E+

カーソル位置: 1+ ▷ (1)+(1)

```
      E
      |
構文木: 1  + ▷ (1)+(1)
```

↓ shift します。

状態 4

スタック: E+(

カーソル位置: 1+(▷ 1)+(1)

```
      E
      |
構文木: 1  +( ▷ 1)+(1)
```

↓ shift します。

状態 5

スタック: E+(1

カーソル位置: 1+(1 ▷)+(1)

```
      E
      |
構文木: 1  +(1 ▷ )+(1)
```

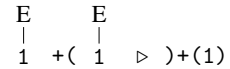
↓ reduce: 下線部分が $E \to 1$ の右辺と一致するので reduce します。

状態 6

スタック: E+(E

カーソル位置: 1+(1 ▷)+(1)

構文木:



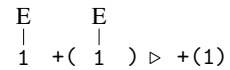
↓ shift します。

状態 7

スタック: $\underline{E\texttt{+(}E}$

カーソル位置: 1+(1) ▷ +(1)

構文木:



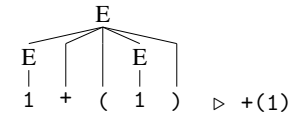
↓ reduce: 下線部分が $E \to E\texttt{+(}E\texttt{)}$ の右辺と一致するので reduce します。

状態 8

スタック: E

カーソル位置: 1+(1) ▷ +(1)

構文木:



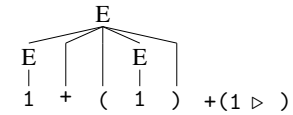
↓ 3 回 shift します。

状態 11

スタック: $E\texttt{+(}\underline{1}$

カーソル位置: 1+(1)+(1 ▷)

構文木:



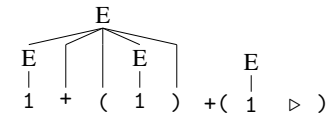
↓ reduce: 下線部分が $E \to 1$ の右辺と一致するので reduce します。

状態 12

スタック: E+(E

カーソル位置: 1+(1)+(1 ▷)

構文木:



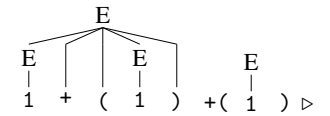
↓ shift します。

状態 13

スタック: $\underline{E\texttt{+(}E\texttt{)}}$

カーソル位置: 1+(1)+(1) ▷

構文木:



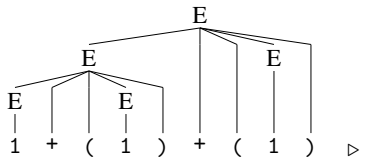
↓ reduce: 下線部分が $E \to E\texttt{+(}E\texttt{)}$ の右辺と一致するので reduce します。

状態 14

スタック: $\underline{E}$

カーソル位置: `1+(1)+(1)` ▷

構文木:



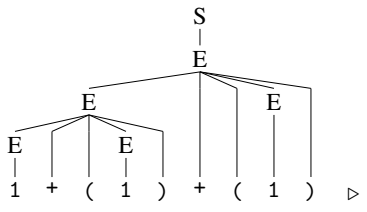
↓ reduce: 下線部分が $S \to E$ の右辺と一致するので reduce します。

状態 15

スタック: S

カーソル位置: `1+(1)+(1)` ▷

構文木:



6.4 LR 構文解析

6.4.1 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」

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」

[*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`

「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」

6.4.2　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」

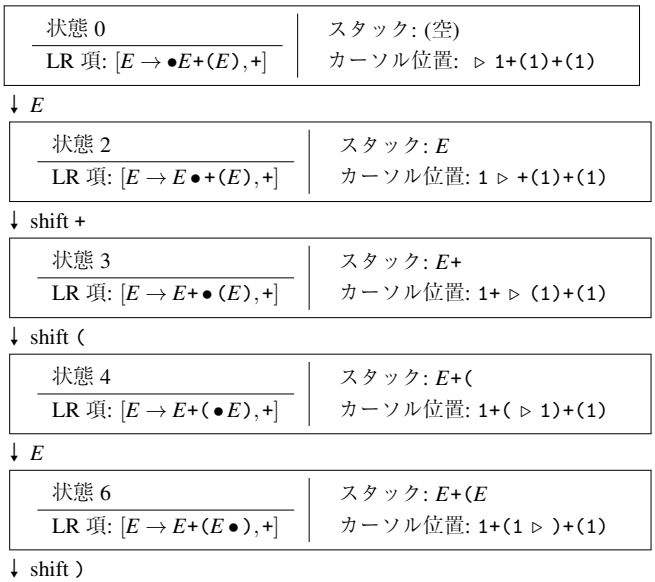
[*7] 正しくは LR(k) item とか LR(1) item なのでしょうが、以下では (k) の部分を省いてこう書きます

「『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……」
……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



状態 7 LR 項: $[E \to E\texttt{+(}E\texttt{)}\bullet, \texttt{+}]$	スタック: $E\texttt{+(}E\texttt{)}$ カーソル位置: `1+(1)` ▷ `+(1)`

↓ reduce $E \to E\texttt{+(}E\texttt{)}$

6.4.3 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 $E \to \texttt{1}$ が使えそうなので、一度元の LR item から離れて、『$E \to \texttt{1}$ を reduce したい気持ち』になります。具体的には、LR item

$$[E \to \bullet\texttt{1}, \texttt{+}]$$

つまり『これから `1` を読んで、$E \to \texttt{1}$ 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、$E \to \texttt{1}$ 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 LR 項: $[E \to \bullet E\texttt{+(}E\texttt{)}, \texttt{+}]$	スタック: (空) 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 $E \to \texttt{1}$ のシーケンスに入る

状態 0′ LR 項: $[E \to \bullet\texttt{1}, \texttt{+}]$	スタック: (空) カーソル位置: ▷ `1+(1)+(1)`

↓ shift `1`

状態 1 LR 項: $[E \to \texttt{1}\bullet, \texttt{+}]$	スタック: `1` カーソル位置: `1` ▷ `+(1)+(1)`

↓ reduce $E \to \texttt{1}$

↓ reduce したら元の生成ルールのシーケンスに戻る

状態 2 LR 項: $[E \to E \bullet \texttt{+(}E\texttt{)}, \texttt{+}]$	スタック: E カーソル位置: `1` ▷ `+(1)+(1)`

↓ shift `+`

状態 3 LR 項: $[E \to E\texttt{+}\bullet \texttt{(}E\texttt{)}, \texttt{+}]$	スタック: E`+` カーソル位置: `1+` ▷ `(1)+(1)`

↓ shift `(`

状態 4 LR 項: $[E \to E\texttt{+(}\bullet E\texttt{)}, \texttt{+}]$	スタック: 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 $E \to \texttt{1}$ のシーケンスに入る

状態 4′ LR 項: $[E \to \bullet\texttt{1}, \texttt{)}]$	スタック: E`+(` 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

状態 0	スタック: ⟨,#0⟩
LR 項: $\#0 = [E \to \bullet E$+(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: ⟨,#0⟩
LR 項: $\#0' = [E \to \bullet 1, +]$	カーソル位置: ▷ 1+(1)+(1)

↓ shift 1

状態 1	スタック: ⟨,#0⟩,⟨1,#1⟩
LR 項: $\#1 = [E \to 1\bullet, +]$	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

↓ 具体的にはまず reduce 元の記号 1 をスタックから pop して

↓ その下にあった要素 ⟨,#0⟩ から LR 項 $\#0 = [E \to \bullet E$+(E),+]

↓ を取得して戻り (状態 0 と似ているがカーソル位置だけは進んでいる)

状態 1^+	スタック: ⟨,#0⟩
LR 項: $\#0 = [E \to \bullet E$+(E),+]	カーソル位置: 1 ▷ +(1)+(1)

↓ reduce 先の非終端記号 E を入力する (スタックに積んで LR 項も遷移する)

状態 2	スタック: ⟨,#0⟩,⟨E,#2⟩
LR 項: $\#2 = [E \to E\bullet$+(E),+]	カーソル位置: 1 ▷ +(1)+(1)

↓ shift +

状態 3	スタック: ⟨,#0⟩,⟨E,#2⟩,⟨+,#3⟩
LR 項: $\#3 = [E \to E+\bullet$ (E),+]	カーソル位置: 1+ ▷ (1)+(1)

↓ shift (

状態 4	スタック: ⟨(,#0⟩, ⟨E,#2⟩, ⟨+,#3⟩, ⟨(,#4⟩
LR 項: #4 = [E → 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 → 1 のシーケンスに入る

状態 4′	スタック: ⟨(,#0⟩, ⟨E,#2⟩, ⟨+,#3⟩, ⟨(,#4⟩
LR 項: #4′ = [E → •1,)]	カーソル位置: 1+(▷ 1)+(1)

↓ shift 1

状態 5	スタック: ⟨(,#0⟩, ⟨E,#2⟩, ⟨+,#3⟩, ⟨(,#4⟩, ⟨1,#5⟩
LR 項: #5 = [E → 1•,)]	カーソル位置: 1+(1 ▷)+(1)

↓ reduce E → 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: ⟨(,#0⟩, ⟨E,#2⟩, ⟨+,#3⟩, ⟨(,#4⟩, ⟨E,#6⟩
LR 項: #6 = [E → E+(E •), +]	カーソル位置: 1+(1 ▷)+(1)

↓ shift)

状態 7	スタック: ⟨(,#0⟩, ⟨E,#2⟩, ⟨+,#3⟩, ⟨(,#4⟩, ⟨E,#6⟩, ⟨),#7⟩
LR 項: #7 = [E → E+(E)•, +]	カーソル位置: 1+(1) ▷ +(1)

↓ reduce E → E+(E)

「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。

6.4.4 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 → 1 のシーケンスに入る』
に対応して

$$[E \to \bullet 1,)]$$

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 → E+(E) のシーケンスに入る』
に対応する

$$[E \to \bullet E+(E),)]$$

という LR 項も可能なはずよね。
結果的には [E → •1,)] しか使えないんだけど、状態 4′ でのカーソル位置までで 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 $E \to \texttt{1}$ 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]、• 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

$[E \to \bullet \texttt{1}, \texttt{)}]$

[*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

$$[E \to \bullet E\texttt{+(}E\texttt{),)}]$$

を追加するわ。$[E \to E\texttt{+(}\bullet E\texttt{),\$}]$ と $[E \to E\texttt{+(}\bullet E\texttt{),+}]$ の中で、$\bullet$ の次の E の次に来るのは) だったので (First()$) = First()+) = {)})、先読みトークンは) になっているわ。

更に、$[E \to \bullet E\texttt{+(}E\texttt{),)}]$ 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

$$[E \to \bullet \texttt{1},\texttt{+}]$$
$$[E \to \bullet E\texttt{+(}E\texttt{),+}]$$

を追加します。

ここで追加する LR 項が尽きるので、

$$\begin{aligned}\text{closure}(\{[E \to E\texttt{+(}\bullet E\texttt{),\$}],[E \to E\texttt{+(}\bullet E\texttt{),+}]\}) = \{&[E \to E\texttt{+(}\bullet E\texttt{),\$}],\\ &[E \to E\texttt{+(}\bullet E\texttt{),+}],\\ &[E \to \bullet E\texttt{+(}E\texttt{),)}],\\ &[E \to \bullet E\texttt{+(}E\texttt{),+}],\\ &[E \to \bullet \texttt{1},\texttt{)}],\\ &[E \to \bullet \texttt{1},\texttt{+}]\}\end{aligned}$$

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」

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...

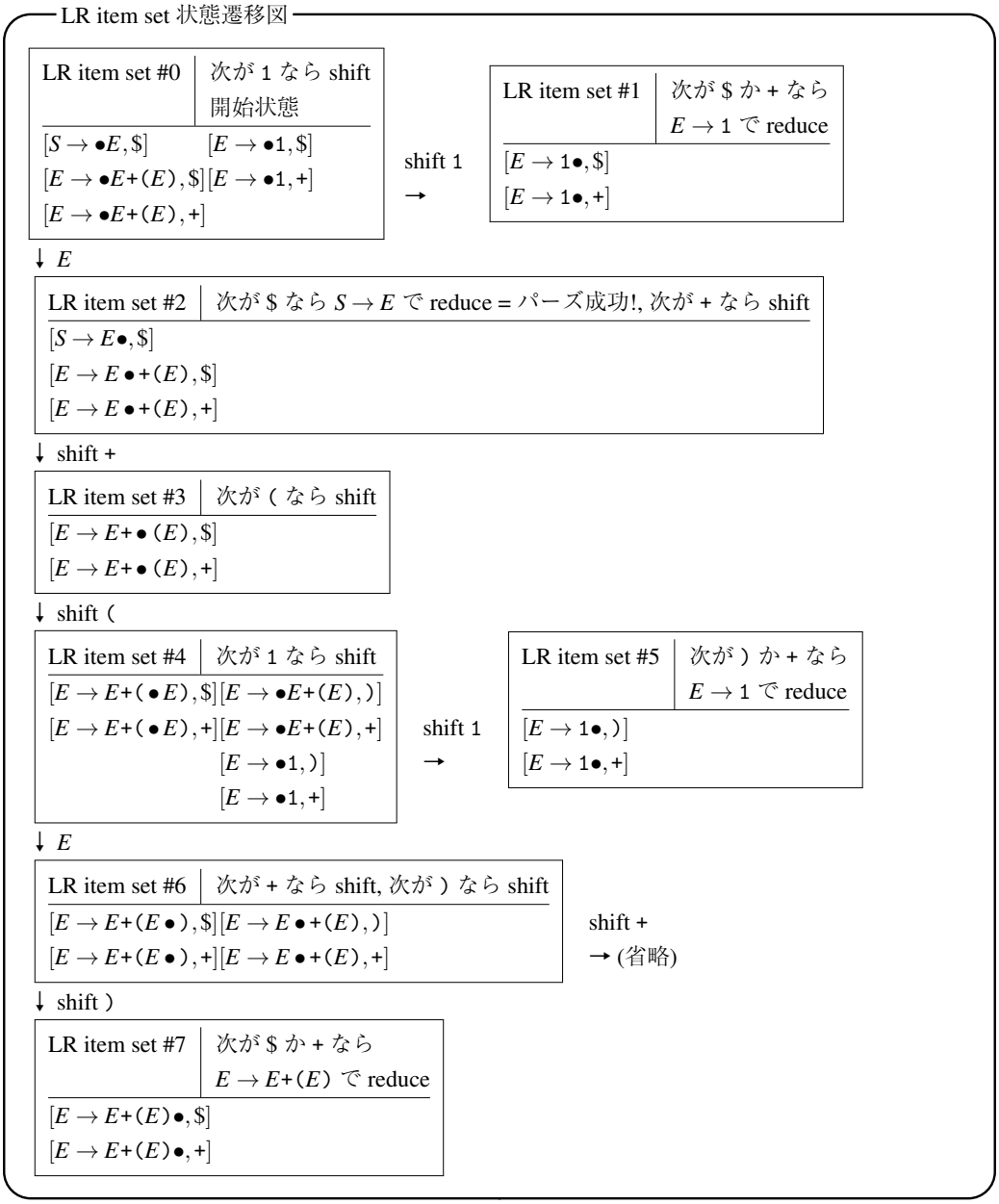
↓ reduce $A \to \gamma$ (前半: pop する)

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」



「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: $\langle, \#0\rangle, \langle \texttt{1}, \#1\rangle$ 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: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle$ カーソル位置: `1+` ▷ `(1)+(1)`

↓ shift `(`

状態 $4 = 4'$ / LR item set: #4	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle$ カーソル位置: `1+(` ▷ `1)+(1)`

↓ shift `1`

状態 5 / LR item set: #5	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle \texttt{1}, \#5\rangle$ カーソル位置: `1+(1` ▷ `)+(1)`

↓ reduce $E \to$ `1`

状態 6 / LR item set: #6	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle E, \#6\rangle$ カーソル位置: `1+(1` ▷ `)+(1)`

↓ shift `)`

状態 7 / LR item set: #7	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle E, \#6\rangle, \langle \texttt{)}, \#7\rangle$ 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: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle \texttt{1}, \#5\rangle$ カーソル位置: `1+(1)+(1` ▷ `)`

↓ reduce $E \to$ `1`

状態 12 / LR item set: #6	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle E, \#6\rangle$ カーソル位置: `1+(1)+(1` ▷ `)`

↓ shift `)`

状態 13 / LR item set: #7	スタック: $\langle, \#0\rangle, \langle E, \#2\rangle, \langle \texttt{+}, \#3\rangle, \langle \texttt{(}, \#4\rangle, \langle E, \#6\rangle, \langle \texttt{)}, \#7\rangle$ 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7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7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
```

7.1 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。

“‘haskell 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)
```

7.2 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。

7.3 Iso と Prism

7.3.1 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)
```

7.4 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。

`haskell 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。

`haskell _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。

7.5 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

関数型イカ娘本#6を書かなイカ!!

まさかの 6 冊目計画! 関数型イカ娘 6 冊目を一緒に書かなイカ?
よく分からない電波を受信しながら妄想と関数型の混ざった素敵本を書く企画でゲソ。
関数型イカ娘と書いたでゲソがイカ娘以外のネタも歓迎するでゲソ!
みんなで楽しく関数型本を書かなイカ!?

参加方法

Skype 常設チャットで雑談とか打ち合わせとかしています。Skype で `xhlkogitsune` にメッセージを送るか、著者 (次ページ参照) までご連絡ください。参加希望の方、あるいは参加を考えている方はお気軽にどうぞ。

スケジュール (暫定)

2013/11/24(日) 深夜　表締め切り/ドラフト提出期限。
2013/12/01(日) 深夜　裏締め切り/未推敲原稿提出期限。ほぼ完成した原稿を出してください。
2013/12/07(土) 深夜　裏裏締め切り/最終締め切り・推敲完了原稿提出期限。
2013/12/29(日)〜31(火)　コミックマーケット 85 (予測)

Call For Articles

Functional Ikamusume book series provides a forum for researchers, developers, and anime-watchers to publish their latest work, articles, and "mousou (妄想)" on the design, implementations, principles, and uses of functional programming and Ikamusume. The book series covers the entire spectrum of work, from practice to theory, from frank introduction of functional programming to latest research work, from Ikamusume with some functional flavor to functional programming with some Ikamusume flavor. Let's enjoy writing articles on functional programming in frank "geso" style de-geso!!
Past books are written entirely in Japanese, and books are planned to be sold in Japan, but we also accept articles in English. You don't have to be worried about what is Functional Ikamusume —— she is just Ikamusume who loves functional programming, as you love it.

How to participate

If you are interested in writing Functional Ikamusume articles, please contact `xhlkogitsune` on Skype or the authors listed in the next page on twitter. You can use Japanese or English for discussion.

Important Dates

November 24th (Sun), 2013　Draft Deadline
December 1st (Sun), 2013　Submission Deadline
December 7th (Sat), 2013　Refinement Deadline/Final Hard Deadline
December 29th-31th, 2013　Comic Market 85 at Ariake, Tokyo, JAPAN where we will sell our books

会員名簿じゃないカ?

@xhl_kogitsune (6 章)

きつねさんと一緒に構文解析のお勉強をしました。楽しかったです。

@master_q (1, 2 章)

John の Haskell コードは世界一のナポリタン!

@nushio (3 章)

ほむほむを出すところまで書けなくて残念でした。

@dif_engine (4 章)

くぅ〜疲れましたｗ 「蓮」「殺」からなるハスケルスレイヤーの物語はこれにて完結です！

@ark_golgo （5 章）

今回初めて同人誌なるものに原稿を出させていただきました。いい経験でした。皆さんありがとうございました。

@fumieval （7 章）

Haskell は手続き型言語ですが、それが何か？

「簡約（かんやく）！？ λ（らむだ）カ娘（か むすめ） Go!」

2013 年 8 月 12 日 コミックマーケット 84 初版発行

2013 年 8 月 15 日 第 2 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, @master_q, @nushio, @dif_engine, @ark_golgo, @fumieval

表紙：@dif_engine