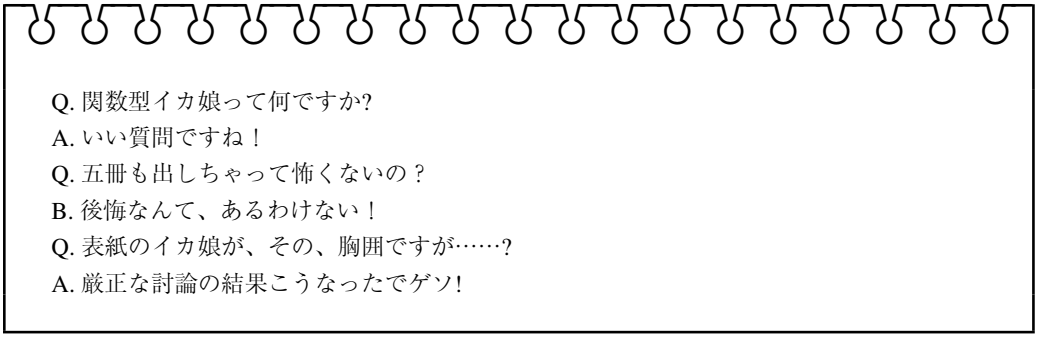
第0章

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。

目次

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」

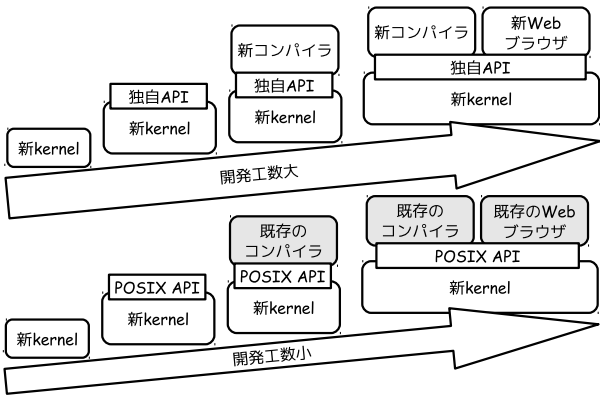


図 1.1: POSIX API を採用することで OS 開発の工数を削減

「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。

「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、

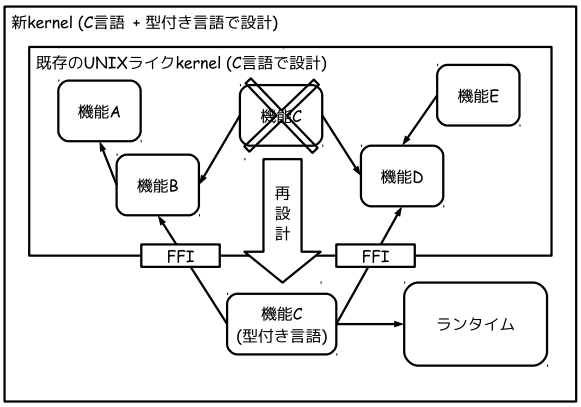


図 1.2: スナッチ設計: 既存 kernel を少しずつ設計置換

[*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

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……
「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

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

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。

「なにいいい！　こんな大事な時にあやつは何をしておるのじゃ！　人類が滅んでしまえばコミケも何もなかろうに！」

駄々をこね、ぶかぶかぬののふくを揺らした艦長はふと気づいてこっちを見る。

「のう客人、お主これを読めるか？」

「何を？」

「これじゃよ」艦長がジェスチャーすると、小さな指からホログラムの光線が艦橋空間を横切って『旧約甘蕉書の 3 ページ』の枠線を点滅させる。

りすと的**かたもるふぬずむ**を産すの事

祖あだむ＝にんじやは弓手に $b \in B$ を持ち馬手に $\oplus \in A \| B \to B$ を掲げ給ひて**りすとかたもるふぬずむ** $h \in A* \to B$ あるべしと宣せ給ふ

$$\begin{aligned} h\,\mathtt{Nil} &= b \\ h(\mathtt{Cons}(a, as)) &= a \oplus (h\ as) \end{aligned} \tag{1}$$

卽ち斯なりぬ◆このごろ都に流行る**はすける**語に於ては foldr とも稱すとぞ◆あだむ＝にんじや之を觀て善とし給ふ◆あだむ＝にんじや甘蕉形の喝鈷にて**かたもるふぬずむ**を表すの沙汰定めおかせ給ひぬ

$$h = (\!|b, \oplus|\!) \tag{2}$$

りすとを引數に取る關數**かたもるふぬずむ**にみな從ひて其數限りなし◆例を擧ぐるに先の征夷對象羣源**れんぐす**賴朝及び數々の名句を殘せし二天流宗家宮本**ふぬるたー**正志是皆**かたもるふぬずむ**の一族なり

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

再歸關數を手ずからに實裝せずして**かたもるふぬずむ**に歸依するこそ**ばぐ**を除き災鬼を祓ふ術なれあなかしこ

ウィンドウには古びた本のページめいたものが映し出されている。艦橋の皆がこっちに注目している。時間の流れが止まったかのような錯覚。俺に何をしろと言うんだ？　戦闘シーンを流す大画面、複数の EMACS ウィンドウに映し出された Haskell コードの断片、一つだけ場違いな古文書……この文書の言語は日本語を基調としているようだが、使われている記号の意味が、それ以前に文意が全くわからない。いや、待てよ、まさかこの状況——

cata という関数を使って、
length と filter を実装しなさい、というクイズ……!!

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。

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」　「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

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

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——」

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

[*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。

第5章

λ力娘探索 ？

— @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

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

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。