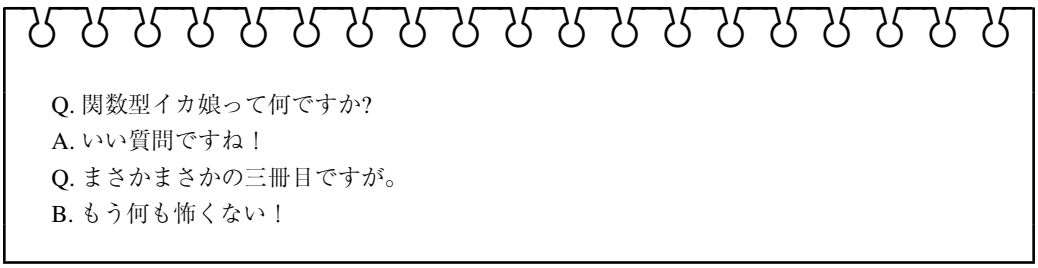

簡約!?
八方娘
(算)

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. まさかまさかの三冊目ですが。
B. もう何も怖くない！

関数型イカ娘とは、「イカ娘ちゃんは 2 本の手と 10 本の触手で人間どもの 6 倍の速度でコーディングが可能な超絶関数型プログラマー。型ありから型なしまでこよなく愛するが特に Scheme がお気に入り。」という妄想設定でゲソ。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)」に続く、三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。

前回までのあらすじ

――『それ』は、人類がプログラミングを始めて以来、世界の各地にてたびたび観測されてきたものであったが、その年の国際学会にて正式に議題として取り上げられ、限りなく仕様の逸脱に近い仕様として、公式に処理されることで一致した。
プロジェクト維持のため、総員が協調し対処にあたるべき認定特異災害『バグ』。
バグはプログラムに潜み、プログラムに接触することで、ゴミへと変えて分解してしまう。
対して、ヒトの行使する通常言語にのっとったデバッグ力は、いかに最新・先鋭を誇っていようと、バグには微々たる効果しか発揮できず、ヒトは往々にして、ただ納期が通り過ぎ、ソフトウェアが納品先へと姿を消すのを待つだけでしかなかった。
ある者は、そんなバグを「まさに災害だ」と評し、またある者は、プロジェクトを襲い、デスマ案件へと変え、やがてプログラマ自身も練炭の塊と崩れ落ちるその様に、「他人を巻き込む自殺願望そのものだな」と吐き捨てるのであった。
物語の舞台は、近未来の日本。日本政府は、公に出来ない暴力装置をいくつかかかえている。神保町にある国立情報学研究所機動部二課は、日露戦争時に英国図書館の指導のもと組織された特務機関を前身としており、一般に周知されている機動部一課と同様、特異災害バグに対する、被害拡大の阻止と事態収拾を担っているのだが、決定的に異なる点がひとつあった。
『型システム』――
天敵バグの駆逐のため、人類が備えうる、唯一絶対の兵器体系。その開発と、その行使である。
型システムは装着者ごとに非常に異なった姿を見せ、強さも様々だが、型システムを身に纏ったものは、対応するバグに対して効率的・有効な攻撃手段を備え、一網打尽にすることが可能となる。
だが、機動部二課が保有する異端技術の結晶でもある強く静的な型推論システムは、バグを殲滅せしめる強力な武装であると同時に、広く普及している既存言語の型システムとは比較にならぬ負担を装着者の脳に強いるもので、激しい拒絶反応を誘発することも多かった。このため、不用意な接触による国民への被害、米国との安全保障条約や、周辺諸外国に対する影響も鑑みられ、日本の型兵器の保有実態は、現在の政府与党判断によって、完全に秘匿されている状態でもある。
誰に知られることなくバグと戦い、ヒトの暮らしを守るニンジャたちが年端もいかぬ少女たちであることを――その正体が、当代トップの競技プログラミングユニットである『ツヴァイウナギ』の２人、[redacted]と[redacted]であることを知る者は、ごく僅かに限定されている。
[redacted]と[redacted]は、この春より、私立バイナリアン程序院高等科に通うこととなった。
憧憬の対象である、アーティスト・[redacted]が通うことでも知られるバイナリアン程序院に通うことは、[redacted]と[redacted]にとって、望外の喜びである。
しかし――そのことが大きな運命の転換になろうとは、まだ気付いてはいない。

ハスケるのを諦めるなッ！
オキャムるのを諦めないで!!

新たなるニンジャソウルの覚醒は、すぐそこにまで迫りつつあった。

第1章

蓮物語

— @dif_engine

物語の概要と目的

集合と写像について一応のことを知ってはいるけれど関数型言語に触れたことがない主人公がHaskell を学びます。同時に主人公は圏論の初歩を学び、圏論を応用すると Haskell の幾つかの重要な型クラスの挙動が理解しやすくなることを知ります。物語を通し、主人公は最小限の圏論の知識を手がかりに Haskell の型を理解しようと試行錯誤することになります。

このような話題をきちんと論じるならば、集合と写像のなす圏を圏論の立場で丁寧に抽象化してから論じることになるでしょう。とはいうものの、圏論をゼロから学び cartesian closed category のあたりまで進めるのはなかなか大変です。そこまで徹底した抽象化を前提としなくても Haskell と圏論の関わりを説明することはある程度可能だと筆者は考えました。物語を通して、圏論についての初歩的な知識を紹介し、Haskell の幾つかの重要な型クラスが従うルールを圏論の立場から説明します。

1.1 プロローグ

怪異——。

そう呼ぶのが如何にもふさわしいように思えた。髪の毛の代わりに頭から青い触手が生えている少女を怪異と呼ばないのなら怪異とは一体何のための言葉だというのだろう。

晴れた夏の午後だった。他の観光客たちが思い思いにビーチライフを満喫する中、海の家で少し遅目の昼食を食べた僕は場違いにもテーブルに本とノートを広げて思案していた。

「——それは Haskell じゃなイカ？」

食べ終えたカレーライスの皿をテーブルから下げるついでに僕のノートを覗きこんだ少女はそんな風に声をかけてきた。少女はイカ娘と名乗った。

「いか——むすめ？」

「そうでゲソ」

風変わりな姿に似つかわしく、やはり名前も風変わりだと思った。他人の名前についてあれこれ言うのは——しかも初対面の相手に対してならなおさらだ——いささか不作法ではあったが僕は素直にそんな感想を述べた。もしかしたら思いがけず怪異に遭遇したことで僕はいくらか動揺していたのかもしれない。

「阿良々木暦（あららぎこよみ）という名前も十分風変わりじゃなイカ？」

これは僕の名前に対する彼女の感想。

とにかく、僕達の会話はそんな風に始まった。
ありふれた出会い。
そしてありふれた会話。
しかし今にして思えばあの瞬間から——謎めいた暗殺者との死闘とその奇妙な結末へ至る運命のレールが敷かれてしまったのではないか——そんな風にも思うのだ。あの死闘の結末を思い出さずにはいられない。
あの暗殺者は——。
——本当に消滅してしまったのだろうか。
枯れて泥水の下に沈んだ蓮の花が残した種が誰にも気づかれる事なく泥中で発芽し、ある日思いがけず白い花として水面上に現れて驚かすように、あの暗殺者が突然この世界に蘇る——そんな事がいつか起こるのではないかと思えてならない。

この物語は——泥のように平凡な未来の日常の中、復讐のために忽然と——蓮の花のように——現れるかもしれない暗殺者の物語だ。また、これは僕が Haskell という言語に出会った物語でもある。

1.2 ラーン・ユー・ア・ハスケル・フォー・グレート・グッド

1.2.1 図書館にて

僕が Haskell に最初に出会ったのは、その数日前の土曜日のことだった。
高校三年生の五月に入ってから僕は遅まきながら大学受験に向けて本格的な勉強を始めていた。事情があって予備校に行かせてもらえなかった僕は夏休みのあいだ同級生に勉強の面倒を見てもらうことにしていたのだった。面倒を見てくれるのは学年トップクラスの成績を有する戦場ヶ原ひたぎと学年トップの成績を有する羽川翼だ。
その日は羽川の担当だった。帰る間際になって彼女から本を手渡された。ポップな青い象の表紙の本と、黄色い洋書——こちらは少し古い本らしい——だった。
「あのさ、阿良々木くんは数学って得意だよね？ これはね、Haskell って言うプログラミング言語の本なんだけど——」
そう言って羽川は象の表紙の本を僕に差し出した。
象の本の表紙には『すごい Haskell たのしく学ぼう！』と書いてあった。
「……よくわからないけどこれを読めということか？」
「——あのね阿良々木くん、大学進学を真面目に考えるなら大学に入るまでだけの勉強だけじゃなくて、卒業したら何をするかも考えないとね——だから世の中に出て武器になりそうな技能を身に着けることも今から意識した方がいいと思うんだ。それで考えてみたんだけど阿良々木くんは数学を核にできるような技能を身につけるといいと思う」
羽川は非の打ち所がない人間だ。恩人でもある。そして彼女はとても面倒見が良い。ついでに言うと思い込みも激しい。
「……つまりこの Haskell という言語は数学が得意なら簡単に身につけられて、しかも世の中に出て武器になるということか？」
「私はそんな事言ってないよ」
「……でしたっけ……？」
「私はね、『武器になりそうな』って言っただけ。実際に武器にできるかどうかは阿良々木くん次第だよ——それに、『数学が得意なら簡単』とも言ってないよ——私は、阿良々木くんが得意の数

学を生かして武器にできそうなものを探して提案しただけだよ」
文字にすると少しキツい事を言っている印象になるが、羽川はこんな時でも——なんというか穏やかな雰囲気なんだよな。
「つまり、武器にできるかどうかは僕の努力次第——という理解で OK?」
「うん。今度の月曜日はどうしても外せない用事があって『一回休み』だから日曜日と月曜日でその象の本を読んでおいてね。」
「羽川先生、なんかすごい無理ゲーな気がするんですが！　イージーモードでお願いします！」
そんな僕の抗議をあっさりスルーして羽川は——。
「それからね、その黄色い本は副読本のつもり。Haskell の本で分かりにくいところがあったらその本を読んでみるといいと思って」
黄色い本には『Categories for the Working Mathematician』と書いてあった。
洋書だ。
英語だ。
「……は、羽川先生、これ英語で書いてあるんですが！」
「数学の英語だから難しくないよ。それに大学に行くなら英語の専門書にも慣れておいたほうがいいと思って」
「……」
斜め上すぎる天才的正論に言挙げする気力を失った僕に追い打ちをかけるように羽川は——。
「んん、——あとね、水曜日に簡単な理解度のテストをするからね」
と宣告した。こうして僕は、本二冊分増し加わっただけのはずの荷物をひどく重く感じながら家路についたのだった。

1.2.2 羽川の本

夕食の後、羽川が貸してくれた本を読み始めた。そのときになって気付いたのだが両方とも図書館から借りてきたのではなく、羽川の私物だった。——図書館で貸してくれたからつい勘違いしていたのだが、よく考えれば羽川が本のまた貸しのようなマナー違反をするわけがないのだった。
洋書の方は古書店で買い求めたものらしく、見返しに鉛筆で数字が走り書きしてあった。いずれにせよ、羽川の複雑な家庭事情から推察する小遣い事情から考えて——安くない買い物だったはずだ。
身が引き締まる思いだった。情けない話だが、本当はこのときまで僕はあれこれ理由をつけて読まずに本を返却するというオプションを頭に入れていたのだった。だが——恩人である羽川がここまでしてくれたのだ。やれるところまで頑張ろう——柄にもなくそんな事を考えながら本を読み進めたのだった。

1.2.3 Haskell

もちろん羽川の心配はありがたかったが、僕もまったく将来の事を考えていなかったわけじゃない。自分なりに数学が得意であることを生かした技能について考え、IT 関係の技能を何か習得しようと考え、C 言語はそれなりに使えるようになっていた——初歩的なレベルではあったが。
一方 Haskell という言語については——まったくの初聞だった。しかし青い象の本を読み始めて少し経つと、数学を得意とするこの僕になぜ羽川がこの言語を薦めてくれたのか、その理由が徐々にわかってきた。この言語を使うと、数学的なアイデアに近い形で計算ができてしまうようだ。例えば、リストを扱ったこんな計算例を見てみよう——。

```
ghci> [x*x+1 | x <- [1..20]]
[2,5,10,17,26,37,50,65,82,101,122,145,170,197,226,257,290,325,362,401]
```

これは見かけも内容も

$$\{x^2+1 \,|\, x \in \{1,\ldots,20\}\}$$

という集合を作る様子と似ている。もし素数かどうかを判定する関数

$$\mathrm{isPrime}\colon \mathbb{Z} \to \{\mathrm{true}, \mathrm{false}\}$$

が与えられていたとしたら、`filter` という関数と組み合わせて、先程のリストからこんな風にして素数だけを濾し取る (filter) こともできる[*1]——。

```
ghci> filter isPrime [x*x+1 | x <- [1..20]]
[2,5,17,37,101,197,257,401]
```

なるほど、こうやって関数を組み合わせて計算してゆくのが Haskell という言語のスタイルなのだろうか……？

1.2.4　数学：関数の復習

数学：関数の定義の復習

関数の話が出たついでに、「関数」という概念が集合論に基づいた数学ではどんな形で定義されていたか、一応復習しておこう。何を今更という感じがしないでもないが、記号や言葉遣いを確認したいという思惑もあるので、少しお付き合い頂きたい。

集合論的な意味での関数 $\varphi\colon A \to B$ とは、積集合 $A \times B$ の部分集合であって次の性質 **(function)** を満たすものを指すのだった——。

(function)　$\forall x \in A \;\; \exists! y \in B \;\; \langle x, y \rangle \in \varphi.$

もちろん、上の条件に現れる「一意的存在」を表す記号 $\exists!$ を解いて

(function')　$\forall x \in A \;\; \exists y \in B \;\; \langle x, y \rangle \in \varphi \;\; \wedge \left(\forall y' \in B \;\; \langle x, y' \rangle \in \varphi \Rightarrow y = y' \right).$

と書きなおしてもよい。$\varphi \subseteq A \times B$ が関数条件 **(function)**（あるいは同じ事だが **(function')**）を満たすとき、$^\forall x \in A$ に対して、$\langle x, y \rangle \in \varphi$ となる $y \in B$ がただ一つ存在するが、このとき **φ の x における値は y であると**言い、

$$y = \varphi(x)$$

と書く。引数 x を括弧で囲まずに

$$y = \varphi x$$

と書く場合があるのもご存知の通り——要するに意味が通じれば良いわけだ。一般に、関数 $\varphi\colon A \to B$ が与えられたとき、A を φ の**定義域**、B を φ の**値域**と呼ぶ。

以上の定義は数学における標準的なものだが、分野によっては関数についてもう少し違った立場の定義を与える場合もある。すなわち、「関数 $\varphi\colon A \to B$」であっても、任意の $x \in A$ に対して、$\varphi(x)$

[*1] `Data.Numbers.Primes` というパッケージを使えば実際にこのコードを実行できる。

が定義されているとは限らないようなものを許容するように関数という概念を定義する立場である。その立場で議論する場合には、$\varphi(x)$ が定義されていないような $x \in A$ が存在するとき φ を**部分関数**と呼び、任意の $x \in A$ に対して $\varphi(x)$ が定義されているとき F を**全域関数**と呼ぶ。以下では数学における標準的な関数の定義を採用する。したがって、特に断らない限り本稿における関数はすべて——後者の立場で謂うところの——全域関数だということになる。

数学：関数の合成

関数あるいは写像の合成というのは自然なアイデアだ。関数

$$f\colon X \to Y$$

と関数

$$g\colon Y \to Z$$

があったとき、$^{\forall}x \in X$ を関数 f で写して得られる $f(x)$ は——f の値域が Y であることがわかっていることから——集合 Y の元となり、従ってこれに対して関数 g を適用することができる。こんな風にして続けて関数を適用すると

$$g(f(x))$$

という元が得られるが、このようにして $^{\forall}x \in X$ に対して $g(f(x))$ を対応させる関数を**関数 f と g の合成関数**と呼び、

$$g \circ f$$

という記号で表すのだった。合成関数 $g \circ f$ の定義域と値域はどうなるだろうか。——これは簡単だ。今の簡単な考察から明らかに

$$g \circ f\colon X \to Z$$

となる。余談だが、こういう話をするとき、

$$\mathrm{dom}(g \circ f) = X,\ \ \mathrm{cod}(g \circ f) = Z$$

などと書くようにすると後々便利だ。つまり、$\varphi\colon A \to B$ のとき

$$\begin{aligned} \mathrm{dom}(\varphi) &:= A, \\ \mathrm{cod}(\varphi) &:= B \end{aligned}$$

としておこう。dom は英語の domain から来ている。cod は、ええと——なんだったっけ[*2] ……。

さて、集合 X, Y に対して

$$\mathrm{Map}(X,Y) := \left\{ \varphi \subseteq X \times Y \;\middle|\; \varphi \text{ は関数} \right\}$$

と定義しよう。つまり、$\mathrm{Map}(X,Y)$ は関数を要素とする集合である。
こうすると、関数合成の記号 $\circ$ を $\mathrm{Map}(Y,Z) \times \mathrm{Map}(X,Y)$ 上の演算とみなすこともできる——。

$$\begin{array}{ccc} \mathrm{Map}(Y,Z) \times \mathrm{Map}(X,Y) & \xrightarrow{\circ} & \mathrm{Map}(X,Z) \\ \langle g, f \rangle & \longmapsto & g \circ f \end{array}$$

関数合成の演算 $\circ$ が結合法則 (associative law)

$$h \circ (g \circ f) = (h \circ g) \circ f$$

[*2] codomain 。普通の英語の辞書には載ってない。

を満たすことは簡単に確かめられる。実際、任意の $x \in \mathrm{dom}(f)$ に対して、関数合成の定義を繰り返し適用すれば

$$\begin{aligned}\big(h\circ(g\circ f)\big)(x) &= h\big(g\circ f(x)\big)\\ &= h\big(g\big(f(x)\big)\big)\\ &= h\circ g\big(f(x)\big)\\ &= \big((h\circ g)\circ f\big)(x)\end{aligned}$$

となる。——当たり前の話だが、定義域と値域がうまく合わなければ関数の合成はできない。関数の合成を考えるときにはいつも定義域と値域を意識して扱う必要がある。

数学：関数記号の位置、特に二項演算子

先程、関数 φ の x における値を

$$\varphi(x) \quad \text{または} \quad \varphi x$$

と書くと説明したが、実際にはどんな関数でも必ずそう書くわけではない。例えば、二つの実数 x_1, x_2 の和 $x_1 + x_2$ を返す関数

$$\begin{array}{rcl} +\colon \mathbb{R}\times\mathbb{R} & \to & \mathbb{R} \\ \langle x_1, x_2\rangle & \longmapsto & x_1 + x_2 \end{array}$$

の場合には関数の記号「$+$」は二つの引数の間に置かれている。こんな風に、二つの引数の間に関数記号を置くタイプの関数を、**中置関数**とか**二項演算子**などと呼ぶのだった。中置関数との比較の意味では、φx のように関数記号を先に、その後に引数を置くタイプの関数は**前置関数**と呼べるだろう。論理的な立場を徹底するならば、関数記号は常に先に書くことに統一したほうがスッキリするのかもしれないが、その立場では結合法則も

$$+\big(+(x_1, x_2), x_3\big) = +\big(x_1, +(x_2, x_3)\big)$$

と書かねばならないし、この書き方は

$$(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$$

と比べて随分と分かりにくいと思うのだ。

ついでの思いつきだが、前置関数と中置関数があるのだから**後置関数**というものがあってもおかしくないな——。

$$x\,\varphi$$

——理屈としてはあってもおかしくないが実例はなさそうだ——いや、そうでもないか。階乗関数

$$n \longmapsto n!$$

の関数記号は『!』だから、これは引数を 1 つ持つ後置関数だ。

前置関数や中置関数や後置関数は、関数の本質についての分類ではなく、単に関数記号をどこに置くかという面から見た分類だ。

関数の種類	引数の数	形式
前置関数	≥ 1	$\varphi(x_1,\cdots)$
二項演算子 (中置関数)	$=2$	$x\ \varphi\ y$
後置関数	≥ 1	$(\cdots,x_1)\varphi$

数学：二項演算子の結合性

これもまた本質というよりは書き方の習慣の話なのだが、二項演算子 (中置関数) には**結合性**がある。二項演算子 $\varphi\colon X\times X\to X$ があったとき、

$$A\ \varphi\ B\ \varphi\ C$$

と書いたとき、真ん中の B：

$$A\ \varphi\ \boxed{B}\ \varphi\ C$$

が「右にくっつく」つまり

$$A\ \varphi\ \left(\boxed{B\ \varphi\ C}\right)$$

と解釈するという約束になっている場合、φ を**右結合演算子**と呼ぶ。同様に、真ん中の B が「左にくっつく」つまり

$$\left(\boxed{A\ \varphi\ B}\right)\ \varphi\ C$$

と解釈するという約束になっている場合、φ を**左結合演算子**と呼ぶ。「結合性は左である」とか「結合性は右である」などという言葉遣いをする場合もある。

1.2.5 数学 ∩ プログラミング：型とその階層

プログラミング言語で扱うデータには色々な種類がある——。それは整数だったり浮動小数点数だったり文字列だったりするわけだが、普通の感覚に照らして考えれば数と文字列のデータは異なる**型 (type)** のデータだ。つまり、文字列データ `s` は文字列型 `String` と呼ばれるデータの集合に属し、整数データ `z` は整数型と呼ばれるデータの集合 `Int` に属し、更に

$$\texttt{String}\cap\texttt{Int}=\varnothing$$

となっている事が了解されているわけだ。型は集合の一種だが、集合ならどんなものでも良いのではなく——`String` や `Int` のように——同質な要素からなるような、特定の集合だけを型として扱う。

何らかの基礎的な集合があるとき、段階的に複雑な集合を作ってゆくというアイデアは自然だ。——例えば、実数列

$$(r_n)_{n\in\mathbb{N}}$$

を考えると、要するに自然数 n に対して実数 r_n が一つだけ決まれば良いのだからこれを関数

$$r\colon\mathbb{N}\to\mathbb{R}$$

と同一視できることがわかる。だから、実数列全体の集合は関数の集合

$$\mathrm{Map}(\mathbb{N},\mathbb{R})$$

と同一視できる。こうして——少し大げさな物言いかもしれないが——集合 $\mathbb{N}$ と $\mathbb{R}$ というビルディングブロックを Map というボンドで貼りあわせることによって、より複雑な集合——$\mathrm{Map}(\mathbb{N},\mathbb{R})$——を作った事になる。

全く同じようにして、**基本型 (basic types)** と呼ばれる基本的な型——データの集合——たち $S_1,...,S_N$ が与えられたとき、これらを Map で貼り合わせることによって、より複雑な**型**を考えることができる。——こうして得られた**型**は、Map で貼り合わせるための新しいビルディングブロックとみなせる——こうして更に複雑な**型**を考えることができる。このような操作を続けた結果得られる**型**全体の集まり $\mathscr{T}$ は、次のようにして——帰納的に——構成することができる。

$$\begin{aligned}
&\mathscr{T}_0 := \{S_1,...,S_N\}.\\
&\mathscr{T}_{n+1} := \mathscr{T}_n \cup \left\{ \mathrm{Map}(T_1,T_2) \,\middle|\, T_1,T_2 \in \mathscr{T}_n \right\} \quad (n \geq 0).\\
&\mathscr{T} := \bigcup_{n\in\mathbb{N}} \mathscr{T}_n.
\end{aligned}$$

結果として得られる型全体の集まり $\mathscr{T}$ は次の性質 **(maptypes1)(maptypes2)** を満たす事になる——。

(maptypes1) $S_1,...,S_N \in \mathscr{T}$,

(maptypes2) $^\forall T_1,T_2 \in \mathscr{T} \quad \mathrm{Map}(T_1,T_2) \in \mathscr{T}$.

与えられた型から別の型を作る方法は Map だけとは限らない。プログラミング言語において、型 X のデータを並べたもの——ここでは $\mathrm{List}(X)$ と呼ぼう——を考えることが多い。集合論的に模倣するならばこんな感じだろうか——。

$$\mathrm{List}(X) := \bigcup_{n\in\mathbb{N}} X^n,$$

ただし $X^0 := \{\varnothing\}, X^1 := X$ とする。この場合、List は一つの型 X に対して別の型 $\mathrm{List}(X)$ を返してくれる対応だと考えられる。いま見てきた Map や List のように、いくつかの「型パラメータ」から新しい型を作る写像を**型コンストラクタ (type constructor)** と[*3]呼ぶ。

Map は 2 つの型パラメータを取る型コンストラクタであり、List は 1 つ型パラメータを取る型コンストラクタであった。一般に、任意の $n>0$ に対して、n 個の型パラメータを持つ型コンストラクタが考えられる。このような型コンストラクタが存在する場合に**型**全体の集まり $\mathscr{T}$ がどうなるか考えてみよう。

$M(>0)$ 個の型コンストラクタからなる集合

$$K = \{k_1,\ldots,k_M\}$$

に対して、それぞれの型コンストラクタが持つ型パラメータの個数——この個数のことを英語では arity と呼ぶ——を対応させる関数

$$\mathrm{ar}\colon K \to \{1,2,3,\ldots\}$$

が与えられているものとする。つまり任意の型コンストラクタ $k \in K$ に対して $\mathrm{ar}(k)$ は k の型パラメータの個数なのだから、型コンストラクタ k は

$$k\colon \overbrace{\mathscr{T}\times\cdots\times\mathscr{T}}^{\mathrm{ar}(k)\text{ 個}} \longrightarrow \mathscr{T}$$

[*3] **型構成子**という訳語を充てる場合もある。

という写像になっている。基本型 $S_1,...,S_N$ が与えられたとき、型コンストラクタ $k_1,\ldots,k_M$ に基づいた型全体の集まり $\mathscr{T}$ の帰納的な構成は、次のようになる――。

$$\mathscr{T}_0 := \{S_1,...,S_N\}.$$
$$\mathscr{T}_{n+1} := \mathscr{T}_n \cup \bigcup_{i=1}^{M} \left\{ k_i(T_1,...,T_{\mathrm{ar}(k_i)}) \;\middle|\; T_1,...,T_{\mathrm{ar}(k_i)} \in \mathscr{T}_n \right\} \quad (n \geq 0).$$
$$\mathscr{T} := \bigcup_{n\in\mathbb{N}} \mathscr{T}_n.$$

この場合、結果として得られる型全体の集まり $\mathscr{T}$ は次の性質 **(gentypes1)(gentypes2)** を満たす事になる――。

(gentypes1) $S_1,...,S_N \in \mathscr{T}$,
(gentypes2) $^{\forall}k \in K\ ^{\forall}T_1,\ldots,T_{\mathrm{ar}(k)} \in \mathscr{T} \quad k(T_1,\ldots,T_{\mathrm{ar}(k)}) \in \mathscr{T}$.

1.2.6 Haskell と型

さっきから型についての取り留めのない話を書いていて、気がついたら 2 ページぐらい使ってしまった気がするが――これは青い象の本を途中まで読んだ時点での感想を述べるための前フリというやつである。

Haskell：関数の型

例えば f が集合 X から Y への関数であるとき、数学では――先ほど導入した型コンストラクタの記号 $\mathrm{Map}(X,Y)$ を用いれば

$$f \in \mathrm{Map}(X,Y)$$

と書ける。Haskell で同等の関数宣言を書くならば次のようになる――。

```
f :: (X -> Y)
```

一般的な書式は

```
変数 :: 変数が属する型
```

――となる。

もし C 言語でこれに相当する関数を書くならばそのプロトタイプ宣言は

```
Y  f(X);
```

となるはずだ――これぐらいなら `f` の型――つまり何を受け取って何を返す関数なのか――を読み取るのは簡単だ。

だが、「関数を受け取って値を返す関数」――例えば

$$g\colon \mathrm{Map}(X,Y) \to Z$$

を考えたりするとき、これに相当する C 言語の関数プロトタイプ宣言[*4] はこんなふうになる。

[*4] 関数ポインタを使っている。説明は省略する。サツバツ！

```
Z  g(Y(*)(X));
```

——もちろん typedef を使えば今のプロトタイプ宣言と等価なものをもう少し読みやすく書けるかも知れない——こんな風に——。

```
typedef Y (*MapXY)(X);
Z  g(MapXY);
```

もちろん「わかりやすさ」というのは人それぞれだ。しかし、どちらの書き方をされたとしても僕はこの g が Map(Map(X,Y),Z) に属する関数を表している事を簡単には読み取れない。一方、g: Map(X,Y) $\to Z$ に相当する Haskell の関数宣言はこうなる。

```
g :: (X -> Y) -> Z
```

さらに複雑なこんな例を考えると——。

```
g :: ((X -> Y) -> Z) -> W    -- Haskell
W  g(Z (*)(Y(*)(X)));  /* C */
```

すげえ。
高階の関数——関数を返す関数——を簡単に書けるんだな。

たった今出てきた (X -> Y) は、数学的には Map(X,Y) に相当し、1.2.5 の用語で言えば 2 つの型パラメータを持つ**型コンストラクタ**である。ただしこの型コンストラクタの記号->は**右結合の二項演算子**のように振る舞う。つまり、

```
X -> Y -> Z
```

——と書いたとき、真ん中の Y は「右にくっつく」ので——

```
X -> (Y -> Z)
```

——と書いたのと同じ意味になる。

Haskell：複数の引数

複数の引数を扱うやりかたは Haskell と C 言語では少し——いや——かなり違う。特別な理由がない限り、n 引数関数に対応する Haskell の関数宣言はこんな風に書くのが良さそうだ——。

```
f :: X1 -> X2 -> X3-> ... -> Xn  -> Y
```

これは

$$f \in \mathrm{Map}(X_1 \times X_2 \times X_3 \times \cdots \times X_n, Y)$$

とは全然違う型の宣言になっている。実際、->の結合性を利用せずに書きなおせば

```
f :: X1 -> (X2 -> (X3 -> (... -> (Xn -> Y) ...)))
```

となり、これに該当するものを Map を使って書くならば

$$f \in \mathrm{Map}(X_1, \mathrm{Map}(X_2, \mathrm{Map}(X_3, \ldots \mathrm{Map}(X_n, Y) \ldots)))$$

となる。——記号が込み入ってきたみたいだ。Map という記号で書くと入れ子が複雑になるので、

必要に応じて

$$\mathrm{Map}(X,Y)$$

という書き方の代わりに

$$X \triangleright Y$$

と書くことにしてみよう。Haskell の真似をして $(X \to Y)$ にしなかったのは

$$X \to Y \to Z$$

と書いたときデータ型 X,Y,Z が

$$X \xrightarrow{f} Y \xrightarrow{g} Z$$

のように関数で結ばれているという状況を——数学での習慣通りに——意味して欲しいからだ。
念のために書けば

$$X \triangleright Y = \{\varphi : X \to Y\}$$

——となる。
演算子 $\triangleright$ の結合性は Haskell に合わせて右にしておく。つまり、

$$X \triangleright Y \triangleright Z$$

と書いたらそれは

$$X \triangleright (Y \triangleright Z) \quad \left(= \ \mathrm{Map}\big(X, \mathrm{Map}(Y,Z)\big) \right)$$

を意味するということだ。少し見慣れない記号を入れたのでついでに練習しておくと、

$$x \in X,\ \ f \in X \triangleright Y \ \ \Rightarrow \ \ f(x) \in Y$$

となる。
さて、新しい記法のウォーミングアップも終わったので

```
g :: A -> B -> C -> D
```

という型を持った Haskell の関数宣言を考えることにする。g を数学の関数 g と同一視して書いてみれば

$$g \in A \triangleright B \triangleright C \triangleright D = A \triangleright (B \triangleright C \triangleright D) = \mathrm{Map}(A, B \triangleright C \triangleright D)$$

——となる。すると、$^\forall a \in A$ に対して

$$g(a) \in B \triangleright C \triangleright D = \mathrm{Map}(B, C \triangleright D)$$

となる。つまり $g(a)$ は関数だ。だから、$^\forall b \in B$ を持ってくれば

$$g(a)(b) \in C \triangleright D = \mathrm{Map}(C, D)$$

となる。全く同じようにして $^\forall c \in C$ を持ってくれば

$$g(a)(b)(c) \in D$$

となる。Haskell で複数の引数を持つ関数を書くとき、こんな風にして「関数を返す一引数関数」として実現することをカリー化[*5] というらしい。

[*5] 大雑把で不正確な説明かも知れないが、以下の議論のためにはこの程度の理解で十分である。

Haskell：多相型

型についての話を書いたとき (1.2.5)、**型コンストラクタ**についても触れたのだった。Haskell では、例えばこんな風に型コンストラクタを宣言できる——。

```
data Maybe a = Nothing | Just a
```

今の例では、Maybe は一つの型パラメータを持つ型コンストラクタになっている。一つ以上の型パラメータを持つ型を**多相型**と呼ぶ。先程も触れたことだが、->は二つの型パラメータを持つ型コンストラクタだ。型コンストラクタは**型を受け取って型を返す**——例えば Maybe に型パラメータ Int を与えると Maybe Int という具体型を返す——写像なのであって、関数より上位の概念だが、やはり関数に似た型を考える事ができる。このような型コンストラクタの『型』に相当するものを型 (type) と区別して kind [*6] と呼ぶ。これは GHCi 上では:kind コマンドで確認できる——。

```
ghci> :kind Maybe
Maybe :: * -> *
```

```
ghci> :kind (->)
(->) :: * -> * -> *
```

Haskell：型クラス

Haskell の型クラスは、型が実装するべき関数の宣言を記述する。自分で定義した型 MyType の要素に対して等号 (==) を使った等値性判定を行いたいならば、MyType を Eq 型クラスのインスタンスとして定義しておけばよい——。

```
instance Eq MyType where
   ......
```

Eq 型クラスの宣言はこんな形をしている——。

```
class  Eq a  where
   (==):: a -> a -> Bool
   (/=):: a -> a -> Bool
```

上に出てくる a は具体的な型が収まるべきプレースホルダーになっている。

Haskell：ファンクター

青い象の本を 7 章まで読み進めると、Functor 型クラスというのが出てきた。それはこんな風に宣言されている——。

```
class  Functor f where
   fmap :: (a -> b) -> f a -> f b
```

f a とか f b という塊が見えるが、これは型でないと意味をなさない。とすると f は型コンストラクタが収まるべきプレースホルダーなのだろうか？ ——どうやらそうらしい。実際にこんな例が掲げられていた：

[*6] 『すごい Haskell 楽しく学ぼう！』では「種類」という訳語があてられている。

```
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f  Nothing = Nothing
```

型クラス Eq の場合、そのインスタンスは具体的な型だが、型クラス Functor の場合、そのインスタンスは多相型でなければならない——ということになる。

更に読み進めようとしたとき、本にメモ——羽川の字だ——が挟んであるのに気付いた。内容はこうだった——。

[羽川のメモ]
functor : 関手、category theory、functor の object-part と arrow-part

羽川が貸してくれた黄色い本『Categories for the Working Mathematician』と関係していそうだ。——そう言えば、Haskell の本で分かりにくいところがあったらその本を読めと言われたのだった。object-part とか arrow-part はなんだろう？ 黄色い本を読んだらわかるのだろうか——。しかし夜も随分更けてしまった。現役の吸血鬼ではない僕にとって、夜は睡眠のための時間だ。

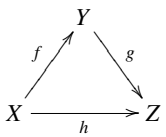
1.2.7 圏論（1）

日曜日は『Categories for the Working Mathematician』を読んだ。Category theory ——日本語では**圏論（けんろん）**と呼ばれる数学の一分野の本だった。英語が得意でない自分が英語で書かれた数学の本を読むなど無理だと思い込んでいたが、使われている文法はむしろ高校の教科書よりもレベルが低いことに気付いた。確かに単語は見慣れないものが多かったが、ネットのお陰で大体の単語は調べがついた。

数学：圏論の意味

今まで紹介する機会がなかったが僕には少し変わった習慣がある。それは、本の序文をちゃんと読むということである。——こう書くと別に普通の事のようだが、序文や目次に丁寧に目を通す人は実は少数派なのだ。そんなわけでいつもどおり僕は黄色い本の序文 (Introduction) を読み始めた。序文[*7] の内容は実際には難しく、最初の段落以外はよくわからなかった。それでも理解できたと思える範囲を序文を参考に説明するとこういうことになる——。

> 圏論は、数学的な諸理論の多くの性質が矢印の図によって統合でき、本質の理解を容易にするという観察から誕生した。例えば集合 X,Y,Z とそれらの間の関数からなる図



> を考えよう。——ここでは、意図的に集合の元 x とその関数値 $f(x)$ の関係を図から省いてい

[*7] Categories for the Working Mathematician(以下では CWM と略記）, p.1

る。このとき、X から Z には h という関数で到達することができるが、「遠回り」して Y を経由して到達することもできる。——この「遠回り」の場合実際には f と g を使っているから結果として合成関数 $g \circ f$ を使って Z に到達したことになる。そんなわけだから、もし $h = g \circ f$ であるときには、X から Z に到達するためには、どの道を通っても同じという事になる。こんなふうに「どの道を通っても同じものが得られる」とき、この図式 (diagram) は**可換 (かかん、commutative)** であると言う。今の例は集合の数がたった三つだったが、もっと集合の数やそれらを結ぶ関数の数が増えてくると、こういった図式でもっと複雑な状況を絵に書いて把握することが出来るというのが利点だ。

ところで、こんな図にまで抽象化してしまうと、この図式を異なる文脈で正しく解釈できることに気づく。例えば、集合 X,Y,Z が位相空間[*8]であり、f,g,h がそれらを結ぶ連続写像であると思っても今の議論は成立する。

序文で理解できたのはこの辺までで、正直に言うが僕は圏論の何が偉大なのか理解出来なかった。僕は序文をきちんと読む少数派の人間だと大見得を切った直後のこの体たらく、面目次第もない。少し面白いと思ったのは、X が集合であることを意識しないように $x \in X$ や $x \mapsto f(x)$ という元と元の関係を意図的に図式から省いてあったことだ。もしかしたら圏論というのは写像で結ばれた集合の系を代数的に抽象化して捉える理論なのかもしれない。

数学：圏の定義

難しくなってきた序文を放り出して読み進めてみるとどうやら僕の予想は大体当たっていたようだった。本に従って定義[*9]を書いてみることにする。

圏の定義

集合族 O,A と写像

$$\begin{array}{lll} A \xrightarrow{\mathrm{dom}} O, & A \xrightarrow{\mathrm{cod}} O, & O \xrightarrow{\mathrm{id}} A \\ f \longmapsto \mathrm{dom}(f), & f \longmapsto \mathrm{cod}(f), & X \longmapsto \mathrm{id}_X \end{array}$$

が与えられているとする。更に、

$$A \times_{\mathrm{O}} A := \left\{ \langle g, f \rangle \in A \times A \,\middle|\, \mathrm{dom}(g) = \mathrm{cod}(f) \right\}$$

上定義された二項演算

$$\begin{array}{ccc} A \times_{\mathrm{O}} A & \xrightarrow{\circ} & A \\ \langle g, f \rangle & \longmapsto & g \circ f \end{array}$$

が与えられたとする。このとき、

$\mathbf{C} := \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ \rangle$ が**圏** (**category**) であるとは (cat1) – (cat5) が成立することである：

[*8] 「連続性」にまつわる事柄を研究するための基礎として作られた空間概念。意外にもプログラミング言語の基礎づけにも役立つ事が後に知られるようになった。

[*9] CWM, p.10, ただし説明の都合や好みなどから記号は一部変えてある。

(cat1) $^{\forall}X \in O$ $\quad \mathrm{cod}(\mathrm{id}_X) = \mathrm{dom}(\mathrm{id}_X) = X.$

(cat2) $^{\forall}\langle g, f\rangle \in A \times_{\mathrm{O}} A$ $\quad \mathrm{dom}(g \circ f) = \mathrm{dom}(f).$

(cat3) $^{\forall}\langle g, f\rangle \in A \times_{\mathrm{O}} A$ $\quad \mathrm{cod}(g \circ f) = \mathrm{cod}(g).$

(cat4) $^{\forall}f, g, h \in A$ $\quad \langle h, g\rangle, \langle g, f\rangle \in A \times_{\mathrm{O}} A \implies (h \circ g) \circ f = h \circ (g \circ f).$

(cat5) $^{\forall}f \in A$ $\quad f \circ \mathrm{id}_X = \mathrm{id}_Y \circ f = f,$ ただし $X := \mathrm{dom}(f), Y := \mathrm{cod}(f).$

圏 $\mathbf{C}$ が与えられたとき、

$$\mathrm{Ob}(\mathbf{C}) := O,$$
$$\mathrm{Arr}(\mathbf{C}) := A$$

とそれぞれ呼び、$\mathrm{Ob}(\mathbf{C})$ の要素を**対象 (object)** と呼び、$\mathrm{Arr}(\mathbf{C})$ の要素を**射 (arrow)** と呼ぶ。

何やら随分抽象的だが、具体例——集合と関数からなる系——を念頭に置きながら読んでみることにしよう。例えば、集合 $S_1, \ldots, S_N$ に対して

$$O := \{S_1, \ldots, S_N\},$$
$$A := \bigcup_{1 \le i, j \le N} \mathrm{Map}(S_i, S_j)$$

としてやり、$^{\forall}X \in O$ に対して

$$\begin{aligned} \mathrm{id}_X : X &\longrightarrow X \\ x &\longmapsto x \end{aligned}$$

として定義してやれば、$\mathbf{C} := \langle O, A, \mathrm{dom}, \mathrm{cod}, \mathrm{id}, \circ\rangle$ は確かに圏になる事がわかる——。

集合と関数を扱っていたときには

$$\mathrm{Map}(X, Y)$$

というレベルで見ると関数合成などの理解がしやすくなったのだった。圏論でそれに該当するものは Hom と書かれることになっている——。

$$\mathrm{Hom}(X, Y) := \left\{ f \in \mathrm{Arr}(\mathbf{C}) \,\middle|\, \mathrm{dom}(f) = X, \mathrm{cod}(f) = Y \right\}.$$

複数の圏を扱っているときは**どの圏で考えているのか**を明記するために添字を付けて

$$\mathrm{Hom}_{\mathbf{C}}(X, Y)$$

と書いたりもする。

圏の定義から、任意の対象 $X, Y, Z \in \mathrm{Ob}(\mathbf{C})$ に対して

$$\mathrm{Hom}(Y, Z) \times \mathrm{Hom}(X, Y) \subseteq \mathrm{Arr}(\mathbf{C}) \times_{\mathrm{O}} \mathrm{Arr}(\mathbf{C})$$

となることがわかるし、更には

$$^{\forall}\langle g, f\rangle \in \mathrm{Hom}(Y, Z) \times \mathrm{Hom}(X, Y) \quad g \circ f \in \mathrm{Hom}(X, Z)$$

となることは簡単に確かめられる。

数学：図式の定義

前に進もうとしたが、図式の可換性の定義がサラリと文章で[*10]述べられていて、なんだかすこし曖昧な気がした。それでも基本的なアイデアは述べられているので、これを元に厳密な定義を考えてみることにした。

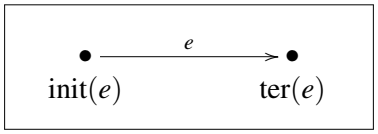
まずは図式を定義してみよう——。これは**有向グラフ (digraph)**——大雑把に言うと、**頂点**と呼ばれる点の集合 V の点を矢印で結んで得られる図形のこと——として定式化できそうだ：

有向グラフの定義

$G = \langle V, E, \mathrm{init}, \mathrm{ter} \rangle$ が**有向グラフ (digraph)** であるとは、次の条件 **(digraph)** が満たされることである：

(digraph)　　$\mathrm{init}, \mathrm{ter} \colon E \to V.$

有向グラフの定義における関数 init は辺——というか絵に書くときは矢印だが——$e \in E$ の**始点 (initial vertex)** を与え、関数 ter は辺 $e \in E$ の**終点 (terminal vertex)** を与える。そんなわけで、各 $e \in E$ に対して次のような矢印が対応していることになる：



これを真似して、圏における**図式**を次のように定義してみる：

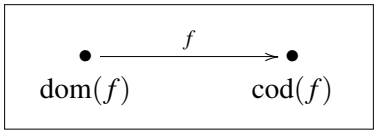
図式の定義

$\mathscr{C}$ を圏とし、$D := \langle V, E \rangle$ とする。ただし、$V \subseteq \mathrm{Ob}(\mathscr{C})$, $E \subseteq \mathrm{Arr}(\mathscr{C})$ とする。このとき、D **が** $\mathscr{C}$ **における図式である**とは $\langle V, E, \mathrm{dom}, \mathrm{cod} \rangle$ が有向グラフであることである。

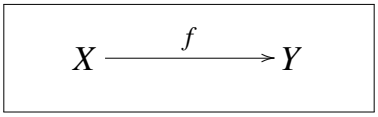
圏 $\mathscr{C}$ における図式 $D = \langle V, E \rangle$ に対して、常に次が成立する事に注意しておこう：

$$\forall f \in E \quad \mathrm{cod}(f), \mathrm{dom}(f) \in V.$$

始点を返す関数として dom を、終点を返す関数として cod を持つような有向グラフを図式と呼ぶように定義したのだから、圏における射 $f \in \mathrm{Arr}(\mathscr{C})$ と $\mathrm{dom}(f), \mathrm{cod}(f) \in \mathrm{Ob}(\mathscr{C})$ の関係をこんな風に



図示しても良いが、これだと頂点の名前が見づらい。こんな場合、$X := \mathrm{dom}(f), Y := \mathrm{cod}(f)$ としてやり、さらに • をやめて



[*10] CWM, p.8

と描くようにしたほうがよほど分かりやすい。そんなわけで、集合と写像を抽象化し、さらに一巡りしてようやく見慣れた図に戻ってきたが、圏の図式という立場からはこれはもう「集合」と「関数」を意味しているとは限らない――。これらは「対象」と「射」からなる図式なのだ。

数学：図式の可換性

図式の可換性を定義するためにはどうしたらよいだろうか。どうやら可換性を定義するためには、「ひとつながりの矢印」を表す言葉があると便利そうだ。グラフ理論の本を探してみると**歩道 (walk)** という概念が見つかった[*11]。

図式の歩道の定義

$\mathscr{C}$ を圏とし、$D := \langle V, E \rangle$ を $\mathscr{C}$ における図式とする。このとき、$f_1, \ldots, f_n \in E$ からなる n 重対 $\langle f_1, \ldots, f_n \rangle$ が D **の歩道である**とは、次の条件 **(walk)** が満たされることである：

$$\textbf{(walk)} \qquad {}^{\forall} i \in \{1, \ldots, n-1\} \quad \mathrm{cod}(f_i) = \mathrm{dom}(f_{i+1}).$$

歩道に含まれる射の個数 n を $\langle f_1, \ldots, f_n \rangle$ の**長さ (length)** と呼ぶ。

$$X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} X_2 \xrightarrow{f_3} \cdots \xrightarrow{f_n} X_n$$

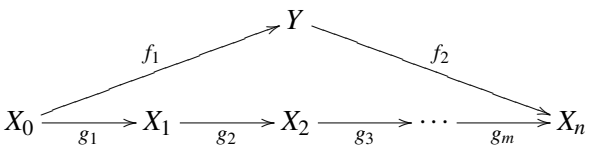
図式 D の歩道全体の集合を $\mathrm{Walk}(D)$ で表す。

三角形の図式の可換性は、長さ 1 の歩道 $\langle h \rangle$ と長さ 2 の歩道 $\langle f, g \rangle$ を合成して得られる $g \circ f$ が一致する事として述べられる。この例をよく検討して一般化すれば図式の可換性は次のように定式化できる――。

図式の可換性の定義

$\mathscr{C}$ を圏とし、$D := \langle V, E \rangle$ を $\mathscr{C}$ における図式とする。このとき、D **が可換である**とは、次の条件 **(com)** が満たされることである：

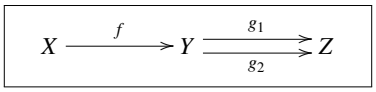
$$\begin{aligned} \textbf{(com)} \quad & {}^{\forall} n \geq 2 \ {}^{\forall} \langle f_1, \ldots, f_n \rangle, \langle g_1, \ldots, g_m \rangle \in \mathrm{Walk}(D) \\ & \mathrm{dom}(f_1) = \mathrm{dom}(g_1) \quad \text{かつ} \quad \mathrm{cod}(f_n) = \mathrm{cod}(g_m) \\ & \qquad \Longrightarrow \quad f_n \circ f_{n-1} \circ \cdots \circ f_1 = g_m \circ g_{m-1} \circ \cdots \circ g_1. \end{aligned}$$



ただし、歩道 $\langle f_1, \ldots, f_n \rangle$ の長さ n が 1 のときは $f_n \circ f_{n-1} \circ \cdots \circ f_1$ は f_1 自身を意味するものと約束しておく。

地味だが条件 **(com)** において $n \geq 2$ という制限を置いていることは重要だ。この制限をおかないと $m = n = 1$ の場合も含まれてしまい、図式 $D = \langle V, E \rangle$ に含まれる任意の対象 $X, Y \in V$ に対して、$\mathrm{dom}(f) = X, \mathrm{cod}(f) = Y$ となる射 $f \in E$ はただ 1 つしか存在しないことになってしまい、可換な図式として許されるものが減ってしまうことになる。

[*11] 良く似た概念に**道 (path)** というものがある。グラフ理論の標準的な言葉遣いでは、**道**は辺や頂点の順序を含まない概念である。合成を考慮する都合上、ここでは**歩道**を利用することにした。



例えば、今の可換性の定義の下では上の図式が可換であることは $g_1 \circ f = g_2 \circ f$ であることを意味する。しかし、**もし**可換性の定義において $n \geq 2$ という制限が**なかったとしたら**、上の図式が可換であることは $g_1 = g_2$ であることを意味するようになってしまう。

数学：関手 (Functor)

一応圏については理解したことにして、関手 (functor) の定義[*12] に進むことにした。名前から判断するに、きっと Haskell のファンクター型クラス `Functor` と関係しているのだろう。

関手の定義

圏 $\mathscr{C}, \mathscr{D}$ に対して、対象を対象に移す関数 F

$$F : \mathrm{Ob}(\mathscr{C}) \longrightarrow \mathrm{Ob}(\mathscr{D})$$

と射を射に移す関数（やはり F と書く）

$$F : \mathrm{Arr}(\mathscr{C}) \longrightarrow \mathrm{Arr}(\mathscr{D})$$

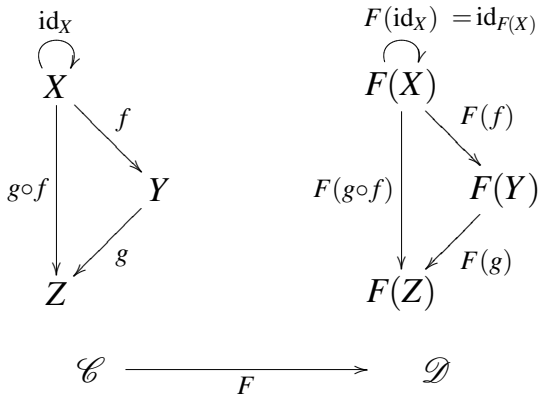
が与えられているとする。このとき、F が $\mathscr{C}$ から $\mathscr{D}$ への**関手 (functor)** であるとは次の **(func1)-(func3)** が成立することである：

(func1) $\forall X \in \mathrm{Ob}(\mathscr{C}) \qquad F(\mathrm{id}_X) = \mathrm{id}_{F(X)}.$

(func2) $\forall X, Y \in \mathrm{Ob}(\mathscr{C})\ \forall f \in \mathrm{Hom}(X,Y) \qquad F(f) \in \mathrm{Hom}\big(F(X), F(Y)\big).$

(func3) $\forall X, Y, Z \in \mathrm{Ob}(\mathscr{C})\ \forall \langle g, f \rangle \in \mathrm{Hom}(Y,Z) \times \mathrm{Hom}(X,Y) \qquad F(g \circ f) = F(g) \circ F(f).$

関手というものがどう役立つかは解らなかったが、とりあえず可換図式を可換図式に移してくれるものだと考えておけばよいのだろうか：



[*12] CWM, p.13, ただし説明の都合や好みなどから (ry

Haskell：関手?

次に、この関手というものがHaskellのFunctor型クラスとどんな関係があるのか考えて見ることにした。

```
class  Functor f where
   fmap :: (a -> b) -> f a -> f b
```

という定義をみると、型コンストラクタ->は右結合性なのだからこんな風に書いても意味は同じのはずだ――。

```
class  Functor f where
   fmap :: (a -> b)  ->  (f a -> f b)
```

こう書いてみるとだんだん関手との関係がわかってきたような気がする。そう思って青い象の本を調べると少し先の方、11.2に『ファンクター則』なんてものが載っていた。引用すると

(ファンクターの第一法則) `fmap id = id`
(ファンクターの第二法則) `fmap (f . g) = (fmap f) . (fmap g)`

つまり、Functor型クラスのインスタンスがfmapを実装する際には上記の二法則を**守るようにしなければならない**ということだ。「ファンクターの第一法則」はちょうど **(func1)** に相当し、「ファンクターの第二法則」は **(func3)** に相当しているようだ。そして **(func2)** はfmapの型宣言と似ている――だから

> Haskell の Functor 型クラスは、型を**対象**とし、関数を**射**とする圏 $\mathscr{C}$ から $\mathscr{C}$ への関手と関係している。

ということになりそうだ。……うーん、……「対象を対象に移す関数」と「射を射に移す関数」を同じ名前 F で呼んでいるせいか、対応が見えづらいような気がする……。
それにしても――何かしっくり来ない。新しく学んだ概念だから落ち着きが悪い――それだけのことなのかも知れない――。
でも――。何か見落としているような気がする。

少し雨が降り出していた。明日は晴れるだろうか。

1.3 海の家「れもん」にて

月曜日。いつもなら今日は羽川に勉強を見てもらう日なのだが、事前の連絡通り今日は一回休み。そんなわけで僕は海にいた。
もしかしたら読者の皆さんは「そんなわけで」が全然説明になってないと感じたかもしれない。
――確かに今日は本当は勉強をしないといけない日だし、本当は海に遊びに来るつもりはなかったのだが。もちろん家を出たときはちゃんと図書館で勉強するつもりでノートも羽川に借りた本も持ってきた。しかし、図書館に向かって自転車を漕いでいるときふと、こんなにも天気が良いし海に行ったらさぞかし爽快だろうなどと水着の用意もしていないくせに思いつき、そんな思いつきのままに駅に自転車を止めて電車に乗ってしまったのだ。
とは言うものの、泳ぐ用意もなかったため仕方なく僕は海水浴に興じる観光客を後目に海の家でノートと本を広げてぼんやりと考えていたのだった――。
こうして冒頭に書いたように僕はイカ娘に出遭った――という次第だ。

1.3.1　圏論（2）

Haskell：アプリカティブファンクター

昼飯時を過ぎているせいか、この時間になると客もまばらだった。最初に話しかけて少し言葉を交わしたあとも、イカ娘は店の手伝いをしながら僕のテーブルの側で立ち止まっては興味津々といった様子で僕のノートを覗きこんだりしていた。少し——いや、かなり気が散るのだが。それにしても、イカ娘は Haskell がそんなにも気になるのだろうか——。

ビーチではしゃぐ観光客の楽しげな声が耳に入るせいなのか——それとも店内を歩き回るイカ娘の華麗な触手さばきに時折目を奪われるせいなのか——あまり集中できないながら、僕は青い象の本の 11.3 を読んでいた。アプリカティブファンクター（applicative functor）の箇所だ。型クラス `Applicative` の主要な部分はこうだ——。

```
class Functor f => Applicative f where
   pure :: a -> f a
   (<*>) :: f (a -> b) -> f a -> f b
```

『`Functor f =>`』という箇所は『`f` が `Functor` 型クラスのインスタンスであるときだけ `f` は `Applicative` 型クラスのインスタンスになれる』という制約条件を表している。——というわけで、`Applicative` のインスタンスは `Functor` のインスタンスでもあるということになる。
そこまではわかる。
だが——なぜこんなものが必要なのだろうか？　それがよく分からなかった——。

分からないという状態にも色々なレベルがある——分からないポイントが自分で解っているレベルの「分からない」から、分からないのポイントが一体どれなのかがすでに判らないというレベルの「分からない」まで——。
自分が今直面してる「分からない」は多分その真ん中あたりだ。青い象の本にはアプリカティブファンクターはファンクターの強化版だと説明してある——まずここがわからない。——なぜ強化する必要があるのだろう？　もし `Functor` 型クラスのインスタンスが型を対象とし、関数を射とする圏 $\mathscr{C}$ から自分自身への関手だというのなら、可換図式はそのまま可換図式に写るのだから、改めて強化する必要はどこにもないはずだ。

考えていてもよくわからず、集中力が——ますます——低下しているのが自分でもよくわかった。
——気分転換しよう。僕以外の客がいなくなってしまったタイミングで、あいかわらず僕のノートが気になる様子のイカ娘に話しかけてみた——。
「なぁ、Haskell に興味があるのか？」
まるで猫缶を見せられた人馴れした野良猫のように目を輝かせてイカ娘がいそいそとやってきた。
「Haskell になってからよく知らないから興味あるでゲソ！」
『Haskell になる』ってどういう意味だろう——。
まあいいや。
少し話をしてみると、イカ娘が卓越した数学の才に恵まれている事に否応なく気付かされた。彼女はどうやらあまりきちんとした教育は受けていない様子だったが、にも関わらず僕のノートや羽川から借りた本をパラパラと数分間めくっただけで僕の知識と理解に追いつき——それどころか僕が

教えを請わねばならない立場になっていた。自分には数学の才能があるなどという自負心を——少しばかり——持ち合わせていたのだが、そんなプライドはこうして完膚なきまでに叩きのめされたのだった。——だが独りで勉強するのも少し飽きてきたところだったということもあり——格好の教師役を得た思いで色々教わることにした。

「どこがわからないか説明してみるでゲソ」

「いやさ、正直言ってどこがわからないかうまく説明できないというか……頭がぼんやりして理解できなくなっちゃうというか……」

「——じゃあその理解できなくなるのがどこか教えてみるでゲソ」

「うーん、それが難しいんだが……そうだな——じゃあ例えばだけど、なんで `Functor` 型クラスに強化版が必要なんだ？」

イカ娘がその質問を聞いて——意味がわからないとばかりに——首を傾げた。そうして僕のノートをもう一度パラパラとめくって何事かを確認してから——こう言った。

「多分——暦は、『型を対象として、関数を射とするHaskell の圏』をちゃんと理解していないでゲソ」

僕は当惑した——その圏のことは十分に理解しているつもりだったからだ——。

「ええと——それはどういう事？」

「じゃあ——例えば Haskell の基本型が仮に $\mathbb{R}$ だけだったらどうなるか考えてみるでゲソ」

「その圏を構成してみせればいいのか？」

「そうでゲソ」

そんなの簡単だ——というより簡単すぎやしないか？　少し馬鹿にされてる気がした。

「——そうだな、じゃあ『型を対象として、関数を射とするHaskell の圏』を $\mathscr{H}$ と呼ぶことにすると、$\mathrm{Ob}(\mathscr{H}) = \{\mathbb{R}\}$ となって——」

「——それはおかしいんじゃなイカ？」

「えっ……？」

「たとえばこんな関数宣言を考えてみるでゲソ——」

```
f :: (Real -> Real) -> Real
```

「——この関数 f の宣言は暦のノートにある数学記法で書けばこんな風になるはずでゲソ」

$$f\colon (\mathbb{R} \triangleright \mathbb{R}) \to \mathbb{R}$$

「……あれ？」

「——どこが変かわかったんじゃなイカ？」

「ええと——$\mathbb{R} \triangleright \mathbb{R}$ も $\mathrm{Ob}(\mathscr{H})$ に属してないとおかしい——？」

「その通りでゲソ。関数の矢印 $\to$ の両側は**型**のはずでゲソ。そして、すべての**型**は圏 $\mathscr{H}$ の対象——つまり $\mathrm{Ob}(\mathscr{H})$ に属する——から $\mathbb{R} \triangleright \mathbb{R} \in \mathrm{Ob}(\mathscr{H})$ でないとおかしいでゲソ」

言われてみれば——いや言ったのは僕か——その通りだった。なんでこんなことに気づかなかったんだろう。……ちょっとノートに書きながら考えてみよう。昨日まとめた箇所 1.2.5 を真似すれば基本型が $\mathbb{R}$ しかない場合の $\mathrm{Ob}(\mathscr{H})$ は——こんな風に構成することができるな……。

$$\begin{aligned} O_0 &:= \{\mathbb{R}\}. \\ O_{n+1} &:= O_n \cup \left\{ T_1 \triangleright T_2 \,\middle|\, T_1, T_2 \in O_n \right\} \quad (n \geq 0). \\ \mathrm{Ob}(\mathscr{H}) &:= \bigcup_{n \in \mathbb{N}} O_n. \end{aligned}$$

「……まあ大体いいでゲソ」

——あれ——この言い方引っかかるぞ……

「もしかしてなんかまだ見落としがあったかな……？」

「$\mathrm{Ob}(\mathscr{H})$ の構成に使われてる型コンストラクタが $\triangleright$ だけなのは寂しいでゲソ」

「……ああ、なるほど——きちんと書くならば型コンストラクタの集合 K——ただし $\triangleright \in K$ を仮定しておく——を使ってこんな風に構成するべきだったか——」

$$
\begin{aligned}
&O_0 := \{\mathbb{R}\}.\\
&O_{n+1} := O_n \cup \bigcup_{k\in K} \left\{ k(T_1,\ldots,T_{\mathrm{ar}(k)}) \;\middle|\; T_1,\ldots,T_{\mathrm{ar}(k)} \in O_n \right\} \quad (n \geq 0).\\
&\mathrm{Ob}(\mathscr{H}) := \bigcup_{n\in\mathbb{N}} O_n.
\end{aligned}
$$

「それでいいでゲソ！」

「——あれ、でも待てよ……今の構成方法だと $\mathrm{Ob}(\mathscr{H})$ は型コンストラクタの集合 K のとり方に依存してしまわないか？」

「いいところに気づいたでゲソ。Haskell の型を対象とし、関数を射とする圏 $\mathscr{H}$ というのは実は確定したものではないでゲソ」

「じゃあ『僕が扱ってる $\mathscr{H}$』と『他の誰かが扱ってる $\mathscr{H}$』が違ってるかもしれないということか……」

「そういうことになるでゲソ」

「でもそれは少し気持ち悪いな……」

「暦は数学を気にし過ぎなんじゃないカ？　確かに圏論は Haskell の挙動を合理的に設計するため使われているでゲソ。でもそれは——Haskell がすべての面で数学の習慣に従わなければいけないという事を意味しているわけではないでゲソ。それに、物事は厳密に論じれば理解しやすくなる——とも言い切れないでゲソ。」

「なるほど、物事を論じるには適切なレベルがあるということか……数学的に完璧に定式化しようと拘る事が却って物事の理解が妨げになっているようでは本末転倒だな。なんとなく痛いところを突かれた気がするよ」

「そんな凡庸な感想は無視して、先へ進むでゲソ」

「堂々とスルーされた！」

「まずは関手の定義をもう一度丁寧に見るでゲソ。暦も少し気になってるようでゲソが、Haskell への応用を考える場合、関手 F の『対象を対象に写す関数』と『射を射に写す関数』は名前を分けた方が結局は分かりやすいと思うでゲソ」

そう言ってイカ娘は僕のノートの新しいページに次のように書いた：

関手の定義 (object-part と arrow-part を分けて記述したものでゲソ)

圏 $\mathscr{C}, \mathscr{D}$ と関数

$$F_O : \mathrm{Ob}(\mathscr{C}) \longrightarrow \mathrm{Ob}(\mathscr{D}),\ \ F_A : \mathrm{Arr}(\mathscr{C}) \longrightarrow \mathrm{Arr}(\mathscr{D})$$

が与えられているとするでゲソ。このとき、$F := \langle F_O, F_A \rangle$ が $\mathscr{C}$ から $\mathscr{D}$ への関手であるとは次の **(func1')-(func3')** が成立することでゲソ：

(func1') $\forall X \in \mathrm{Ob}(\mathscr{C}) \qquad F_A(\mathrm{id}_X) = \mathrm{id}_{F_O(X)}$.

(func2') $\forall X,Y \in \mathrm{Ob}(\mathscr{C})\ \forall f \in \mathrm{Hom}(X,Y) \qquad F_A(f) \in \mathrm{Hom}\big(F_O(X), F_O(Y)\big)$.

(func3') $\forall X,Y,Z \in \mathrm{Ob}(\mathscr{C})\ \forall \langle g,f\rangle \in \mathrm{Hom}(Y,Z)\times\mathrm{Hom}(X,Y) \qquad F_A(g\circ f) = F_A(g)\circ F_A(f)$.

「……少し気になったんだけどさ、お前っていつもこのレベルで言文一致なの？」

「何かおかしいでゲソ？」

「……いやさ、この語尾まで含めて言文一致って相当変わってると思うけど——特に手書きのときは。ごめん、気にしないで続けてくれ……」

「——とにかく、こんな風に分けて書いてみるとHaskellの`Functor`型クラスとの関係がはっきりするんじゃなイカ？」

少し考えてみた。**(func2')**を書きなおすと

$$F_A\colon \mathrm{Hom}(X,Y) \to \mathrm{Hom}\big(F_O(X),F_O(Y)\big)$$

となる。そして、今考えている圏$\mathscr{H}$では

$$\mathrm{Hom}(X,Y) = \mathrm{Map}(X,Y) = X \triangleright Y$$
$$\mathrm{Hom}\big(F_O(X),F_O(Y)\big) = \mathrm{Map}\big(F_O(X),F_O(Y)\big) = F_O(X) \triangleright F_O(Y)$$

なのだから、結局

$$F_A\colon X \triangleright Y \to F_O(X) \triangleright F_O(Y)$$

となる。ここまで来れば

```
fmap :: (a -> b)  ->  (F a -> F b)
```

との関係は明らかだった——。

「——なるほど、`fmap`はF_Aに対応しているのか」

「そして関手Fのobject-part F_Oは型パラメータを1つ取る型コンストラクタになっているでゲソ」

圏論の関手とHaskellの`Functor`

F_O：関手のobject-part	`Functor`のインスタンスの型コンストラクタ
F_A：関手のarrow-part	`Functor`のインスタンスが実装する`fmap`

「この辺のことは薄ぼんやりは理解していたけど今はっきりと理解できたみたいだ」

「『わからない』にも色々な段階があるように——『わかる』にも色々な段階があるでゲソ」

「なるほど……いや勉強になります……」

「本題のアプリカティブに進む前に、なるべく記号を簡単にしたいから、関手Fのobject-partは同じ記号Fで、arrow-partは——できるだけ——区別してF_Aと書くようにしなイカ？」

「うん、そうしよう。Fと書いたときにそれが関手自身なのか関手のobject-partなのかは文脈で判断できそうだからな。——混乱しそうなときは必要に応じてF_Oという記号を使うようにすればいい」

「——早速でゲソが——こんな関数があったとするでゲソ——」

```
f :: X -> Y
```

「`X`と`Y`は何らかの具体型ってことだよな——」

「そうでゲソ。そして、`F`がHaskellのファンクターだとして、この`f`を元にこんな関数`f'`が欲しいとするでゲソ——」

```
f' :: F X -> F Y
```

「——それだったら fmap で作れる——」
「その通りでゲソ。じゃあ今度は f と f' がこんな形だったらどうなるか考えてみるでゲソ」

```
f :: X -> Y -> Z
f' :: F X -> F Y -> F Z
```

それも fmap で——と即答しそうになったが、この質問は罠の臭いがする。よく考えてみよう。

$$f \in X \triangleright Y \triangleright Z = X \triangleright (Y \triangleright Z)$$

なので

$$F_A(f) \in F(X) \triangleright F(Y \triangleright Z)$$

となるが、$F(Y \triangleright Z) = F(Y) \triangleright F(Z)$ **であるとは一般には言えない**。
このことは具体例を考えれば分かる。`Maybe(X -> Y)` は、集合としては

$$\{\texttt{Nothing}\} \cup \left\{ \texttt{Just f} \;\middle|\; \texttt{f :: (X -> Y)} \right\}$$

だが、`(Maybe X) -> (Maybe Y)` には、ある `y0 :: Y` に対して

```
g :: (Maybe X) -> (Maybe Y)
g _ = Just y0
```

と定義できる関数、つまりどんな引数に対しても `Just y0` を返すような関数も含まれている。明らかに、これは `Nothing` とも違い、しかもどの `f :: X -> Y` に対する `Just f` とも違う。
——こう考えてみると——。
「fmap だけじゃダメみたいだな」
「その通りでゲソ。——まずは、アプリカティブの`<*>`の型を見るでゲソ——。

```
(<*>) :: f(a -> b) -> f a -> f b
```

「これを今の状況に合わせて数式で書きなおせばこうなるでゲソ。

$$\text{（二項演算子）}\quad \langle * \rangle \quad : F(Y \triangleright Z) \times F(Y) \to F(Z).$$

「だから、$^\forall x \in F(Y \triangleright Z)$ と $^\forall y \in F(Y)$ に対して——

$$x \langle * \rangle y \in F(Z)$$

となるでゲソ。でも——これをこんな風に解釈することもできるんじゃなイカ？」

$$x \langle * \rangle \in \mathrm{Map}(F(Y), F(Z)) = F(Y) \triangleright F(Z).$$

「ええと——その $x \langle * \rangle$ というのはラムダ記法で書けば

$$x \langle * \rangle = \lambda y \mapsto (x \langle * \rangle y)$$

——ということ？」
「そういうことだけど——要するにただの部分適用じゃなイカ？」
「いや——言われてみればそうなんだけど、何故か二項演算子は部分適用できないように思い込んでたみたいだ」

「確かに、初めて見るとぎょっとするかもしれないでゲソ。でも、理屈は普通の関数と同じだし、要するに慣れの問題なんじゃなイカ？」
うーん……どうだろう。実際に慣れてみないとその感覚はわからないな。——あれ……？
「——今気づいたんだけど、これって x の後に関数記号 $\langle * \rangle$ が来ているから、つまり**後置関数**になってる！

$$\text{（後置関数）}\quad \langle * \rangle \quad : F(Y \triangleright Z) \to F(Y) \triangleright F(Z).$$

「その通りでゲソ」
「いやしかし、土曜の夜に思いつきで書いた後置関数の話題がここでつながるのか。こういうのを予定調和っていうんだろうな」
「……伏線を回収できたのがそんなに嬉しかったでゲソか……」

続きを考えてみよう。$x' \in F(X), y' \in F(Y)$ とすると——先程確認したように——$F_A(f) \in F(X) \triangleright F(Y \triangleright Z)$ だから

$$F_A(f)x' \in F(Y \triangleright Z)$$

なので、$\langle * \rangle$ を使えば

$$F_A(f)x'\langle * \rangle \in F(Y) \triangleright F(Z)$$

となる。従って、

$$F_A(f)x'\langle * \rangle y' \in F(Z)$$

であることがわかる。最後の式の左辺を Haskell 記法で書けば

```
(fmap f) x' <*> y'
```

となる。更に、`Applicative` 型クラスにおける F_A つまり `fmap` の後置関数バージョン`<$>`を使えば、

```
f <$> x' <*> y'
```

とも書ける。そんなわけで——。

「さっきの問題の答えは

```
f' :: F X -> F Y -> F Z
f' t u = f <$> t <*> u
```

だな」
「もっと引数が多い場合はどうなるでゲソ？」
「ここまでくればもう簡単だ。

```
f :: X1 -> X2 -> X3 -> ⋯ Xn -> Y
```

の引数の型を全部『`F` 付き』にしたものはこんな風に定義できる：

```
f' :: F X1 -> F X2 -> F X3 -> ⋯ F Xn -> F Y
f' x1 x2 x3 ⋯ xn = f <$> x1 <*> x2 <*> x3 <*> ⋯ <*> xn
```

」
「どうやらちゃんとわかったみたいじゃなイカ」

「いや、本当に助かったよ」

この後も話は弾み、モナドに関連して自然変換の話まで教わったのだが、機会があればこの話も書いてみたい[*13]。

1.4 死闘

1.4.1 商店街

夏の長い午後も終わりかけていた。少し涼しくなり、いかにも海水浴といった風情の客は減り始めた。気の早い一部の客が花火と消火用のバケツを持ちだしていたが外はまだ明るかった。もう少し待てば素敵な夕日を眺められそうだった。

夕食を作って待っているであろう妹たちの事を考えると——それでも少し普段より遅くなりそうだったが——そろそろ帰らねばならない時刻だった。駅まで送ってくれると申し出てくれたイカ娘と僕は小ぶりな商業用ビルやシャッターの降りた店が並ぶ静かな商店街をゆっくりと歩いていた。少ないながらも人の出入りが絶えなかった海の家から離れ、僕らは二人きりになっていた。何故かそんな事を急に意識してしまい、横で歩いているイカ娘は僕のことをどう思っているのだろう——などと埒もない事に思いを巡らせ始めたとき——イカ娘が突然足を止めた。

イカ娘の表情が殺気立っていた。待て、イカ娘はこんなにも——剣呑な雰囲気をまとった奴だったろうか？　僕は身の危険を感じた。怪異が人の姿をしており人と同じような感情を持っていたとしてもそれが怪異である以上何がしかの人知を超えた力を持つ存在なのであり、下手に感情を刺激することは命取りになりかねない。怪異に遭遇する人間は稀だとはいえ、怪異との接し方で生死が左右される事は珍しいことではないのだ。幾つかの怪異に接しているうちに、僕はいつしか怪異慣れ——悪い意味で——してしまっていたのかも知れない。イカ娘がどんな力を持っているのかはわからないが、しばらく忍に血を吸わせていない僕は吸血鬼としての最大の力を発揮することはできない。

もっとも、吸血鬼としての力が弱っているからこそ真夏の日差しの中を歩いても平気なのだが。

僕の影の中から忍が様子を伺っている気配が感じられたがこの危機的な状況に応じて何かしてくれる様子はなかった。もちろん彼女には僕を助ける義理もないのだ。必死で打開策について思いを巡らしている探す僕を無視して、イカ娘は静かな声でこう言った——。

「すごいニンジャ・アトモスフィアを感じる……でゲソ」

その頃になって僕も周囲を囲繞するただならぬ殺意に気付いた。まとわりつくような殺意。その恐るべき必殺の悪意はイカ娘が睨みつける方向から放たれていた。

「何者でゲソ？」

イカ娘がまだ姿を見せていない相手に呼びかけた。数十メートル先——路上にわだかまっていた不定形の黒っぽい霧のようなものが凝集して黒装束の男の形を取った。男が出現するのに合わせたかのように周囲が薄暗くなった。同時に夏の暑気は急速に薄れてゆき——それどころか肌寒さすら感じるようになっていた。

忍者——だろうか。男の姿は時代劇などで見かける忍者のような衣装をまとっていたが、コスプレ会場や撮影現場から抜けだしてきたという印象は全く受けなかった。——男がまとう漆黒の衣装から漂う間違え様のない血臭は、男が正真正銘本物の殺人者であることを物語っていた。

[*13] 残念。時間切れである。

男の素顔の半分を黒い面頬で隠しており、その両頬には「蓮」「殺」という赤い文字が妖しく光っていた。男はこちらに進み出ると軽く一礼し

「ドーモ、ハスケルスレイヤーです」

と挨拶した。男は軽い一瞥を僕にくれたがどうやら取るに足らない雑魚だとみなしたのか、僕に対しては名乗らなかった。次に男はこう叫んだ――。

「Haskeller 死すべし！　貴様をケジメする！」

これに応じてイカ娘は

「いきなり現れて――」

その台詞を最後まで聞き終える事はできなかった――なんの前触れもなく、唐突に全ての感覚を失ったからだ。

1.4.2 商店街（闇）

無聲の闇に放り出された僕は自分でも意外なほど落ち着いており、状況についてゆっくりと考えていた。五感はまだ回復していなかったが――なぜか忍の気配を感じた。――ふと、ここは忍のいる影の世界なのではないかと思った。やがて、ゆっくりと五感が戻ってきた。場所は今さっきまでいた商店街のはずだったが依然として非常に暗く、誰の姿も見えなかった。物の見え方が少しおかしかった。近くの建物に目を凝らすと急速に建物が遠ざかるような感じがした。周囲の物体は月光を受けてか仄かな銀光を帯びており、その微細な銀の光の粒たちによって物体の輪郭を認識できるのだった。

――ふと気になって月を見上げると、それは見慣れた月ではなく、夜空に高く輝く黄金の立方体だった。その立方体からは蜘蛛の糸のような銀の細かい繊維が数限りなく伸びており、それらは立方体から遠ざかるにつれて急速に夜空に溶け込んでいた。

「起きたか」

突然話しかけられて僕は振り向いた――が誰もいなかった。

「無駄じゃ。この空間では儂は形を纏っていない」

久しぶりに聞く忍の声だった。

「ここは影の世界じゃないのか？」

僕は先程から思っていたことを口に出した。

「うぬとは口を聞かん」

「話しかけてきたのはそっちじゃないか！　ひょっとしてツンデレキャラに路線変更でもしたのか？」

「バーバヤガからの伝言じゃ。『観察し、書きとめよ』とのことじゃ」

「バーバヤガって誰だそれ？」

「知らん」

「『観察』って何をだ？」

「知らん」

「一体どうなってるんだ？」

――最後のは質問じゃなく独り言だ。

「始まるぞ」

との声に促されて商店街に再び目をやると、商店街には二つの人影があった。――最初にこの闇の商店街を見たときには誰もいなかったはずだが。

僕の傍らにはいつの間にかイカ娘が立っていた。やや距離をおいた向こうには忍者装束の――先

程「ハスケルスレイヤー」と名乗った——男が立っていた。二人とも彫像のように静止していた。二人の姿は基本的には先程普通の世界で見た姿と似ていた——だが対比をなすような違いもあった。この闇の世界の中では男は二回りほど体が大きく、禍々しく陰惨な闘気をその身に滾らせていた。イカ娘は——美しかった。現実世界より頭身が高く、体全体を循環する微細な銀色の粒子によって体全体が輝いていた。

——突然、まるで一時停止が解除されたかのように二人の体に動きが戻った。イカ娘が叫んだ

「——なに理不尽な事を言ってるでゲソ！」

対峙する二人と僕はどうやら少しだけずれた時間軸に置かれていたらしい。そしてたった今僕は戦いの舞台を観察し記録すべく同期する時間軸に置かれた——そういうことらしかった。

1.4.3 ◆◆◆◆イカ娘 vs ハスケルスレイヤー◆◆◆◆

漆黒の闇を纏った禍々しいニンジャ、その名はハスケルスレイヤー。そのメンポの左右に刻まれた、鮮血よりなお赤い朱文字は「蓮」「殺」。溢れんばかりの殺意を宿すその視線の先にはイカ娘。このコトダマ空間におけるイカ娘は長身で女神めいた白銀の輝きを帯びており、その胸も豊満であった。

「狙いはなんでゲソ！」

「貴様をケジメする……。Haskeller は……死なねばならん！」

「それが理不尽だと言ってるでゲソ！」

ハスケルスレイヤーがメンポの奥で顔を嘲弄に歪めた。そして驚愕の真意を語り出した。

「……理不尽ではない。Haskell の高度な表現能力を獲得したコトダマ空間の情報寄生体が際限なく自己増殖を繰り返した結果、このコトダマ空間は情報密度臨界に近づいている。臨界点を超えればこのコトダマ空間が相転移を起こし一気に爆発する。そうなれば、お前たちが現実と呼んでいる世界も破滅するのだ」

「何言ってるかわからないし、この変な世界が終わると私の世界も破滅するなんておかしいじゃなイカ？」

「……だから貴様達は何もわかっていないと言うのだ。貴様たちが現実と呼んでいる世界はこのコトダマ空間の中に実装されている箱庭空間に過ぎない。したがってこのコトダマ空間の終焉とともに貴様達の宇宙も終わる」

ゴウランガ！　ハスケルスレイヤーの口から語られる壮大な宇宙観！

「お前の言ってる事は妄言でゲソ！　なにも証拠がないじゃなイカ！」

「証拠ならある……私が証拠だ。未来世界においてすでにコトダマ空間の相転移は起こったのだ。その影響で貴様達にとっての現実である箱庭宇宙の計算力学系の不動点が移動した。その結果、全ての物理定数が変動し、宇宙はビッグ・クランチによる終焉を迎えた。私は保存されていた継続を利用してスタックフレームを巻き戻し、歴史を修正するために過去の宇宙にに戻ってきたのだ」

「さっきから何を言っているかわからないでゲソ！　それに、もしそうだとしてもそれはこの宇宙とは違う並行世界での出来事でゲソ！　こっちの宇宙も破滅するとは限らないじゃなイカ！」

「貴様が納得するかどうかは問題ではない。この宇宙を破滅から守るためにすべての Haskeller をケジメする。貴様も例外ではない。名乗れ。そして名誉ある死を迎えるがよい！」

これ以上話しても無駄だとようやく悟ったのか、イカ娘がこの求めに応えて決断的にアイサツした。

「ドーモ、イカ娘です——。お前をぶちのめす！」

先に仕掛けたのはハスケルスレイヤーであった。イカ娘がオジギを終えた瞬間に二本のクナイ・ダートを投擲！　だがどうしたことだろう？　クナイはイカ娘に向かうどころか垂直に近い仰角で投げられている！　ハスケルスレイヤーとて男、コトダマ空間におけるイカ娘のわがままなボディ

に心ならずも気をそらされ手元が狂ったのだろうか？　続けざまに地を這うように前転しつつクナイを投擲する。だがまたしてもクナイは大きめの仰角で投げられている！　そして再び前転しつつクナイを投擲！　そしてようやくコントロールを取り戻したのか、最後に投げられた二本のクナイはまっすぐにイカ娘を目指して飛翔する！　だが見よ、最後に投げられた二本のクナイがイカ娘に近づくにつれて大仰角で投擲されたクナイたちが描く放物線が次々と最後のクナイに合流しつつある！　前転しながらの不安定な姿勢での連続投擲にも関わらずのこの驚くべき精密制御である――なんというワザマエ！
合計八本のクナイがイカ娘に命中するかと思われた――その刹那！　すべてのクナイはイカ娘の把握力を持つ触手に捉えられていた！　タツジン！　ハスケルスレイヤーのワザマエも見事だがイカ娘の神速かつ精密な触手さばきもまたスゴイ級である。息を付く間も与えずイカ娘がその触手に捉えられたクナイを全て同時に猛スピードで投げ放った。自らの凶器により絶体絶命の境地に落ちるハスケルスレイヤー。インガオホー。

速やかに迫る死の脅威に対してハスケルスレイヤーは瞬時に身を構え――体を振り絞るように放ったのはただ一発のツキ・パンチ――。
全身の筋肉が引き絞られる――加速度ベクトルが合成されるその極みには恐るべき豪速の拳！
――いまやハスケルスレイヤーの拳は超音速に達している！　右の拳を頂点に形成された衝撃波コーンと衝突した八本のクナイ・ダートは甲高い金属音を立てて崩壊した！　クナイ・ダートが持っていた運動量自体はさほど減殺されなかったとはいえ、分散して飛来する欠片にはもはや当初の殺傷力はない。ハスケルスレイヤーが軽く顔をガードしただけで飛来するクナイ・ダートの欠片はたちまち無力化されてしまう。そしてハスケルスレイヤーのツキ・パンチが生み出した衝撃波はそのままイカ娘に襲いかかった！　まさに攻防一体、ミヤモト・マサシの思想を具現化したタイチー・カラテの極意である！　コワイ！

イカ娘は咄嗟に体の正面に触手を束ねて壁を作る。続いて轟音とともに襲来する衝撃波――だが衝撃波はテンタクル・ウォールに吸収されてイカ娘には効かない！　この無力化は彼女が軟体動物だからこそ出来るジツである。もし彼女の体に一箇所でも硬い部分があったなら先ほどのクナイ・ダートのように衝撃波の破壊力に巻き込まれていたことだろう。軽業師めいた精密動作をメインとしたジツをしかけてくると思わせつつ豪快な超音速ツキ・パンチを放ったハスケルスレイヤーのコン・フーはもはやヤバイ級の域に達している――だが、瞬時に相手の攻撃を見極めて最小限の防御で防ぎ切ったイカ娘の冷静な判断もまた、彼女が見かけによらない熟練の戦士であることを物語っていた。

思いがけず先に油断したのはハスケルスレイヤーであった。イカ娘がテンタクル・ウォールを解除する際の動作が必要以上に鋭く、触手が微かに空気を切り裂いていたのを見逃したのである。高速運動する触手先端で生み出された真空の渦が音もなくハスケルスレイヤーに忍び寄り、次の瞬間全身を切り裂いた！　ナムサン！
「グアーッ！」
悲鳴を上げるハスケルスレイヤー！　全身の切創はいずれも深く、噴水めいた勢いで血が噴き出している。――だが、ハスケルスレイヤーは予想外にも手負いのスモトリめいた推進力を発揮し、目にも留まらぬ速さで連続ツキ・パンチを繰り出しながら襲いかかる！　これを己の拳と触手で迎え撃つイカ娘！
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」

「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
二本の腕しか持たぬハスケルスレイヤーと、両腕に加えて十本の触手を持つイカ娘。これまでの戦闘から伺える実力を考慮すると、手数で上回るイカ娘が優位なのは明らかである——。だが奇妙にも両者の応酬は拮抗していた。ナンデ？　注意深く観察するとその理由は明らかであった——イカ娘は全ての触手を戦闘に回していない！　十本の触手のうち四本は背中側にまわっており、戦闘とは無関係の奇妙で複雑な動作をしていた。勝利を確信したイカ娘が手を抜いて応戦することで相手を侮辱しているのだろうか——いやそうではない。見よ！　背中側の触手が動きまわるのにあわせてこぼれ出す銀の粒は複雑な軌跡を空間に残し、今や複雑な魔法陣を形成しつつあった。ハスケルスレイヤーはイカ娘がなにか仕掛けようとするのを察知したのかトドメを刺そうと一層鋭い攻撃を仕掛ける！
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」「イヤーッ！」
だがついにイカ娘が編んでいた魔法陣は完成し、周囲のコトダマ・エネルギーを吸収して起動！
突如虚空に緑色の光点の群れが湧き出し、中空の一平面上に整列して文字を形成した——。

```
GHCi, special version for KOTODAMA-IRC space :? for help
Loading package haskell-karate ... linking ... done.
ghci>
```

イカ娘が出現した光点スクリーンの上で触手を滑らせ文字を入力する——。

```
ghci> haskell_slayer <- return $ Just haskell_slayer
\noindent
```

入力された文字が爆発的な光輝を帯びた。それに合わせるようにスクリーンが巨大化し、周囲の空間を巻き込みながら湾曲してゆく。湾曲の勢いは止まらず、いまや円軌道の残像を残す光点のパイルからなるトルネードに成長している。トルネードの鋭い先端が象の鼻のように持ち上がり——ハスケルスレイヤーの体に突き刺さった！

物理的な衝撃を予期して身構えるハスケルスレイヤーの目に当惑の表情が浮かんだ。トルネードが突き刺さると同時に緑色の光点は肉体の厚みなどまったく無視してハスケルスレイヤーの体に素早く浸透し、微生物めいた動きで泳ぎまわっていた。トルネードの光点がすべてハスケルスレイヤーの体に注ぎ込まれると、光点の群れはハスケルスレイヤーの体の中でますます輝き、やがて急速に消滅していった。同時にハスケルスレイヤーの全身の輪郭が徐々に曖昧になり、急速に不透明度を失ってゆく。ここまで読んできた読者には何が起こったのかよく理解できることであろう。——そう、ハスケルスレイヤーは今や `Maybe` モナドに閉じ込められて $\mathrm{Ob}(\mathscr{H}^{\mathrm{Maybe}})$ に属する存在と化しており、$\mathrm{Ob}(\mathscr{H})$ に属する我々とは文脈がずれてしまったのである。もはや向こうからは何も手が出せない。
これが、あっけない死闘の幕切れであった。

1.4.4 別れ

一瞬めまいを感じた。周囲が明るくなった。微かな潮香を感じる——元の空間に戻ってきたのだ。濃厚だったニンジャ・アトモスフィアも薄れていた。元の空間に戻っての僕の最初の感想は——

「あっ、一人称に戻ってる！」

——だった。

イカ娘は——見たところ無傷だった。やや戦闘の興奮が残っているのか微かに頬を上気させている。ハスケルスレイヤーが先程立っていた場所には誰もいなくなっていた。だが、目をついと左右にずらすと微かな陽炎のようなゆらめきがあり、男の姿らしき輪郭をなしているのがわかった。僕は不安を感じてイカ娘に訊いた——。

「あれはもう危険じゃないのか？」

「大丈夫、あそこから出てくることは決して出来ないでゲソ」

「本当に——？」

「Maybe モナドによって文脈が変化して『いるかもしれないし、いないかもしれない』存在になってしまったから、もうこっちに干渉することはできないでゲソ」

「なるほど——」

とは言ったものの、僕はまだ不安を感じていた。——だがそれはそれとして。

「それにしてもさっきの技は凄かったな」

「あれくらいの相手、私にとってはベイビー・サブミッションでゲソ！」

「アイエエエエ！　ニンジャ・アトモスフィア消えてない！」

——僕は悲鳴を上げてしまった。

帰りの電車に揺られながら、僕はつらつらとハスケルスレイヤーの事を考えていた。イカ娘はハスケルスレイヤーが出てこれないと自信を持っていたようだが、Maybe はモナドだから——いや、やめておこう。考えても仕方がない。世の中の問題は、大抵は完全には解決できない。Maybe に閉じ込められたハスケルスレイヤーは、いつの日かこちらに『干渉してくるかもしれない』と考えるべきだ。

1.5 エピローグ

後日談というか、今回のオチ。

水曜日。僕は羽川の出したテストに合格し、面目を保つことができた。——イカ娘のおかげだ。羽川は興味深げに僕のノートをチェックして

「んん？ ——へぇ、Cartesian Closed Category [*14] まで行かなくてもこういう話が出来るんだね。——ちょっと意外だったかな」

——という感想をくれた。意味は解らなかったがとりあえず書いておく。

戦場ヶ原にも今回の顛末を話した。彼女の感想は——

「阿良々木くん。——どうもイカ臭いと思ってたらそんな子と会ってたのね」

——だった。

なんというか。

相変わらずだよなあ。

[*14] CWM, p.95

参考文献

この記事の執筆にあたり、以下の諸作品からキャラクターや設定を拝借いたしました。素晴らしい作品を発表されている原作者様各位に敬意を表明いたします。勝手なキャラクターや設定の拝借および改変については何卒ご寛恕下さいますよう願い申し上げます。

1.5.1 ネタ元

- 安部真弘「侵略！ イカ娘」

- 西尾維新〈物語〉シリーズ
  特に「化物語」（上・下）、「傷物語」、「偽物語」（上・下）

- ブラッドレー・ボンド、フィリップ・ニンジャ・モーゼズ「ニンジャスレイヤー」
  `http://d.hatena.ne.jp/NinjaHeads/`

また、執筆にあたって以下の文献を参考にいたしました。

1.5.2 参考にした教科書および専門書

[1] 結城 浩「数学ガール」シリーズ ソフトバンク・クリエイティブ
本記事の直接のネタ元というわけではありませんが、主人公「僕」が周囲の人間を巻き込みあるいは巻き込まれながら、対話を交えた試行錯誤のなかで数学を学んでいくというスタイルには多大な影響を受けました。

[2] 松村 英之「集合論入門」朝倉書店 (1966)
本記事が前提としている数学独特の言い回しや記号遣いに不慣れの場合は、「日常語としての集合論」についての教科書に目を通しておくことをおすすめします。そのような教科書は数限りなくありますが、個人的な好みからこの松村先生の本を推薦いたします。数学の諸分野で日常語として使われる集合論が、圏論への誘導を意識して説明されています。

[3] Miran Lipovača「すごい Haskell 楽しく学ぼう！」, 田中英行・村主崇行 共訳, オーム社 (2012/5)
プログラミング経験者向けの Haskell 入門書として評判の高い
"Learn You a Haskell for Great Good!: A Beginner's Guide", No Starch Press (2011/4)
の日本語訳です。とても読みやすく訳されていると思います。なお、英語原文は Web で全文を読むこともできます：`http://learnyouahaskell.com/`

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman "Compilers: Principles, Techniques, and Tools", Addison-Wesley (1986/1)
表紙の絵から「ドラゴンブック」と呼ばれて親しまれたコンパイラの教科書です。型や型コンストラクタについての一般論を書く際にこの本の 6.1 を参考にしました。

[5] S. Mac Lane, "Categories for the Working Mathematician", Springer-Verlag (1971)
よく読まれている圏論の教科書です。何の説明もなく「CWM」と言う略称で呼ばれることもあります。すでに何らかの分野における数学の専門知識を持っている読者を想定読者としているため難しいところも多いのですが、圏論について真剣な興味があるなら持っていて損はありません。日本語訳も『圏論の基礎』というタイトルで出版されています。

[6] J. L. Bell, "Toposes and Local Set Theories", Oxford University Press (1988)
図式の定義や可換性の定式化はこの本を参考にしました。現在は Dover から安価なレプリントが入手可能です。

[7] R. Diestel,「グラフ理論」 根上生也・太田克弘 共訳, シュプリンガー・フェアラーク東京 (2000)
"Graph Theory" 2nd ed., Springer-Verlag (2000)
の日本語訳です。グラフ理論に関係する事項を書く際に参考にしました。

1.5.3 追記

- 初版原稿の作成時に、`@xhl_kogitsune` 氏にはとりわけ丁寧な査読をして頂き、幾つもの有用なコメントを賜りました。おかげで、説明の方法や話の流れをかなり改善できました。この場を借りて御礼申し上げます。
- 第二版の印刷に際し、幾つかの誤植を訂正し、数式の微調整を行いました。また、気のついた箇所の文言を再度整えました。大幅な内容の追加はしていませんが、p.15 において、初版における明らかな説明の欠落と思われた id_X についての簡単な記述を補いました。

第2章

RTS海溝二万マイル

— @master_q

みんな毎日 GHC で元気にはすはすしてるでゲソ？ Hackage には色々なパッケージが登録されていて、それらの使い方を調べると新しい知見が得られ、読んでるだけで楽しいでゲソ。

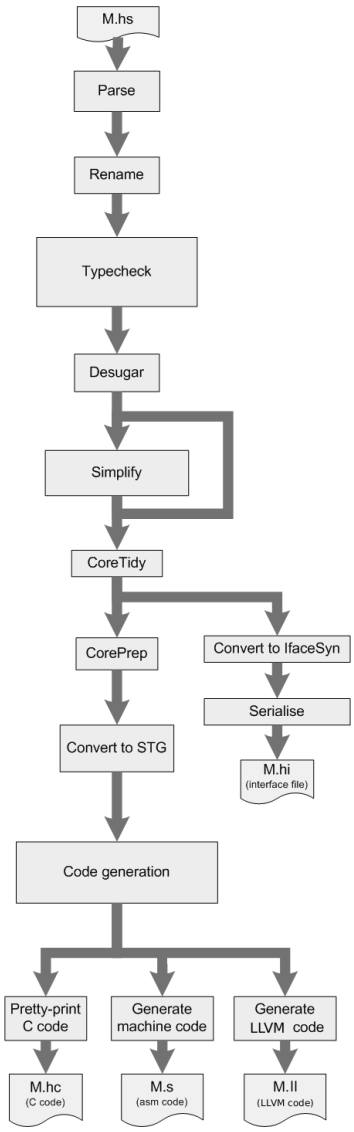
ところで、こんな楽しい世界 (GHC) がどんなしくみで動いているのか知りたくなイカ？

2.1 RTS は Haskell コードを実行する VM でゲソ!

一言に GHC と言っても中は広いんでゲソ。今日はその中でも RTS(Runtime System) というモジュールの中を探検してみようと思うでゲソ。

ところで RTS ってなんでゲソ？ ちょっと Haskell から離れて他の言語の実行モデルを思い出してみるでゲソ。C 言語の場合、ソースコードをコンパイラにかけるとアセンブリ言語になり、アセンブラを通すことで機械語になるでゲソ。この機械語同士をリンクすれば実行バイナリができるでゲソ。Java 言語の場合、ソースコードをコンパイラにかけると機械語ではなくバイトコードになるでゲソ。このバイトコードは Java VM という仮想マシンの上で実行できるでゲソ。その Java VM 自体は C 言語や C++ 言語など Java 以外の低レベルな言語で設計されることが多いでゲソ。Perl 言語の場合、ソースコードはコンパイラにかけず、Perl インタープリタ上で実行するでゲソ。そのインタープリタはやはり C 言語のような低レベル言語で設計されているでゲソ。

さて、Haskell に戻ってみるでゲソ。GHC は Haskell のソースコードを右の図のようなコンパイルパイプライン[*1]を通して機械語に変換するでゲソ。他の言語と比較すると C 言語のケースと似ているでゲソね。この機械語化された Haskell の関数をつなげれば、Haskell で作ったプログラムの意図通りに動作をする実行バイナリができるはずでゲソ。



[*1] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/HscMain

じゃあ、Haskell で書かれたソースコードはこのパイプラインを通してオブジェクトファイルを作り、それらだけをリンクすれば実行バイナリになるのカ、というとそんなことはないのでゲソ。以下のようなプリミティブは (少なくとも現時点では)Haskell で書くことができていないでゲソ。

- コンパイル済み Haskell 関数の実行
- forkIO などで作った Haskell スレッドのスジューリング
- 並列実行
- メモリ管理
- GC
- STM

既存の C 言語ライブラリでも Haskell ライブラリでも実現できないこのような機能を提供するのが RTS なんでゲソ。別の言い方をすると、GHC にとって RTS は Java にとっての VM のようなものだ、ということもできるでゲソ。この RTS の全てを解説することは残念ながら大変荷が重いでゲソ。今回は実際に Haskell ソースコードをコンパイルしてみた結果を調査することで、「機械語化された Haskell 関数を RTS がどのように実行するのか」を解明することに焦点をしぼってみようと思うでゲッソ。

今回対象とする実行環境は Debian GNU/Linux amd64 sid 2012/07/01 時点、GHC のバージョンは 7.4.1 で、-threaded オプションは使わないでゲソ。[*2]

2.2 [表層] Haskell の世界

まず簡単な Haskell コードを書いてみるでゲソ。

```
main :: IO ()
main = putChar 'H'
```

うん、とっても簡単でゲソ。このコードはどうやって動くかわかるでゲソ？

1. main 関数から実行開始
2. putChar に'H' を適用
3. hPutChar に stdout と'H' を適用
4. (その他いろいろな Haskell 関数が呼び出される)
5. なんだかんだあって stdout のバッファに'H' を書き込む

Haskell 言語の上ではこれで動作を理解できたことになるでゲソ。しかし実行バイナリとしての動作を考えてみると色々な疑問がわくじゃなイカ。

わからないことリスト
*** [疑問 1] そもそも `main` 関数がどこからどうやって呼び出されるのか？**
*** [疑問 2] `Haskell` の関数はどうやって他の関数を呼び出すのか？**
*** [疑問 3] `stdout` バッファの `flush` は誰がやるのか？**

[*2] RTS にはスレッド版と非スレッド版があるでゲソ。-threaded オプションを付けない場合には、コンパイラは非スレッド版 RTS を使うようになるでゲソ。

2.3 潜水艦に乗り込もうじゃなイカ!

まず “[疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？ ” を解明したいでゲソ。ひょっとしてさっきの main.hs ソースコードをコンパイルしてできたオブジェクトファイルの中に C 言語の main 関数が紛れているんじゃなイカ？ 調べてみるでゲッソ!

```
$ ghc -c Main.hs
$ nm Main.o
0000000000000020 D Main_main_closure
0000000000000018 T Main_main_info
0000000000000010 d Main_main_srt
0000000000000050 D ZCMain_main_closure
00000000000000b8 T ZCMain_main_info
0000000000000040 d ZCMain_main_srt
0000000000000000 D __stginit_Main
0000000000000000 D __stginit_ZCMain
                 U base_GHCziTopHandler_runMainIO_closure
                 U base_SystemziIO_putChar_closure
                 U ghczmprim_GHCziTypes_Czh_static_info
                 U newCAF
0000000000000000 d sfC_closure
                 U stg_CAF_BLACKHOLE_info
                 U stg_ap_p_fast
                 U stg_bh_upd_frame_info
```

Main.o の中には C 言語の main 関数はなかったでゲソ。どうやって実行されるのかさっぱりじゃなイカ……。それはそうといくつか見えるシンボルは何を表わすのでゲソ？ Haskell 関連のシンボルは z-encoding [*3] という規則でエンコーディングされているでゲソ。それを翻訳するとイカのような意味になるでゲソ。

- Main_main_closure := Main.main closure
- ZCMain_main_closure := :Main.main closure
- base_GHCziTopHandler_runMainIO_closure := base GHC.TopHandler.runMainIO closure
- base_SystemziIO_putChar_closure := base System.IO.putChar closure
- ghczmprim_GHCziTypes_Czh_static_info := ghc-prim GHC.Types.C# static info

GHC.TopHandler, System.IO, GHC.Types, Main はモジュール名じゃなイカ。でも:Main なんてモジュールは定義した覚えがないでゲソ。また closure と info というのはなんでゲソ？ さらに以下のシンボルは RTS 組み込みのもののようでゲソ。

- newCAF
- stg_CAF_BLACKHOLE_info
- stg_ap_p_fast

[*3] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/SymbolNames

- stg_bh_upd_frame_info

わからないことが増えたじゃなイカ……。

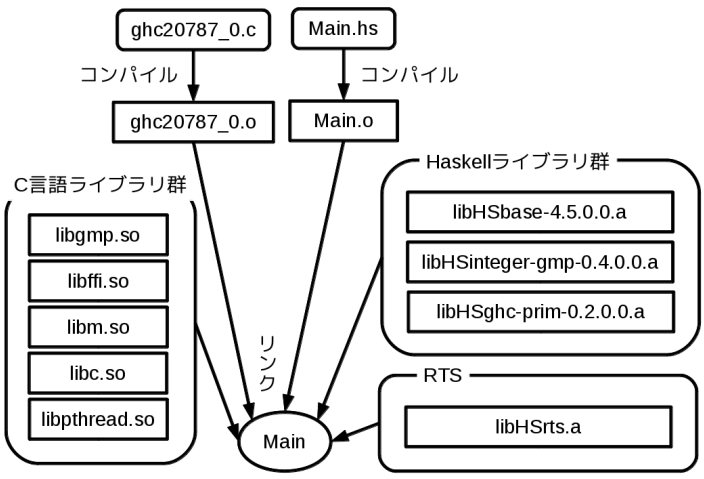
```
### わからないことリスト（更新）###
* [疑問 4] :Main モジュールとは何か？
* [疑問 5] closure, info とは何か？
* [疑問 6] newCAF とは何か？
* [疑問 7] stg_*とは何か？
```

2.4　[中深層] C 言語の世界

気分をかえて、ghc -v コマンドで Main.hs のコンパイル順序をよく見てみるでゲッソ。

```
$ ghc -v Main.hs
--snip--
*** C Compiler:
'/usr/bin/gcc' '-fno-stack-protector' '-Wl,--hash-size=31' '-Wl,--reduce-
memory-overheads' '-c' '/tmp/ghc20787_0/ghc20787_0.c' '-o' '/tmp/ghc20787
_0/ghc20787_0.o' '-DTABLES_NEXT_TO_CODE' '-I/usr/lib/ghc/include'
*** Linker:
'/usr/bin/gcc' '-fno-stack-protector' '-Wl,--hash-size=31' '-Wl,--reduce-
memory-overheads' '-o' 'Main' 'Main.o' '-L/usr/lib/ghc/base-4.5.0.0' '-L/
usr/lib/ghc/integer-gmp-0.4.0.0' '-L/usr/lib/ghc/ghc-prim-0.2.0.0' '-L/us
r/lib/ghc' '/tmp/ghc20787_0/ghc20787_0.o' '-lHSbase-4.5.0.0' '-lHSinteger
```

ふむ。このログからコンパイルの流れ図を書いてみるとイカのようになるでゲソ。



今回作った Main.hs を動かすために、Haskell ライブラリ群と C 言語ライブラリ群の力を借りる

んでゲソが、その寄せ集めでは実現できない部分を吸収してくれるのが RTS で、そのライブラリ名は libHSrts.a なんでゲソ! [*4]

ところで身に覚えのない/tmp/ghc20787_0/ghc20787_0.c などというソースコードがコンパイル対象になっているでゲソ。気になるので生成ファイルを残すように GHC にオプションを与えてみるでゲソ。

```
$ ghc -keep-tmp-files -tmpdir ./tmp Main.hs
```

すると tmp/ghc25150_0/ghc25150_0.c なんてファイルが残っていたでゲソ。

```
#include "Rts.h"
extern StgClosure ZCMain_main_closure;
int main(int argc, char *argv[])
{
    RtsConfig __conf = defaultRtsConfig;
    __conf.rts_opts_enabled = RtsOptsSafeOnly;
    return hs_main(argc, argv, &ZCMain_main_closure,__conf);
}
__asm__("\t.section .debug-ghc-link-info,\"\",@note\n\t.ascii ...
```

見つけたでゲソ! C 言語の main 関数じゃなイカ! ということは、GHC が作った実行バイナリは hs_main 関数から始まるということでゲソ。hs_main というのは何者なのでゲソ？ この hs_main から Haskell の世界を繋ぐのが RTS なのでゲソ。RTS の本体はこの hs_main 関数から呼び出される real_main 関数で、この関数を見れば Haskell プログラムの一生が一目瞭然でゲソ!

```
static void real_main(void)
{
    int exit_status;
    SchedulerStatus status;

    hs_init_ghc(&progargc, &progargv, rtsconfig);
    {
        Capability *cap = rts_lock();
        rts_evalLazyIO(&cap,progmain_closure, NULL);
        status = rts_getSchedStatus(cap);
        taskTimeStamp(myTask());
        rts_unlock(cap);
    }
    switch (status) {
/* --snip-- */
    }
```

[*4] スレッド版 RTS のライブラリ名は libHSrts_thr.a でゲソ

```
    shutdownHaskellAndExit(exit_status);
}
```

real_main 関数の中身を簡単に解説すると、

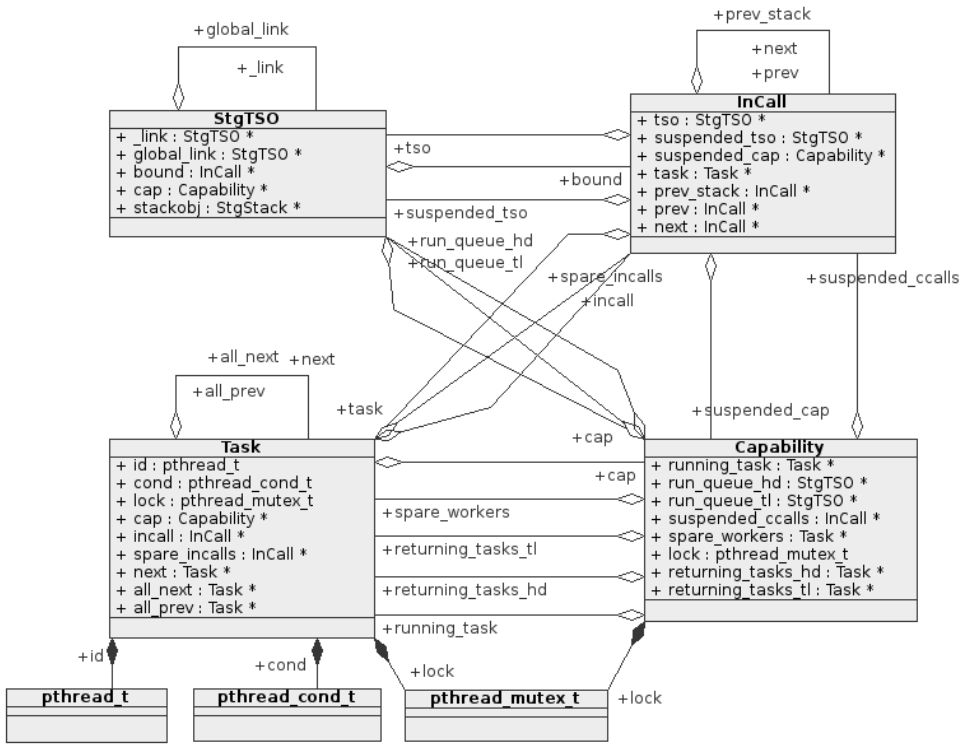
1. hs_init_ghc := RTS の初期化
2. rts_lock := Task 生成、Capability の取得
3. rts_evalLazyIO := StgTSO を生成して run queue に繋げ、スケジュール実行
4. rts_unlock := Task 解放
5. shutdownHaskellAndExit := 後処理

のようになっているでゲソ。キモは rts_evalLazyIO 関数あたりだということがわかると思うでゲソ!

ところで Capability とかなんのことでゲソ？

- Capability := RTS における仮想 CPU。非スレッド版 RTS では一つだけ作られるでゲソ
- Task := 実際の実行を司る何か。非スレッド版 RTS なので、最初は一つだけ作られるでゲソ
- StgTSO := Haskell のスレッドの実体。Haskell のスタックはこの構造体が保持しているでゲソ

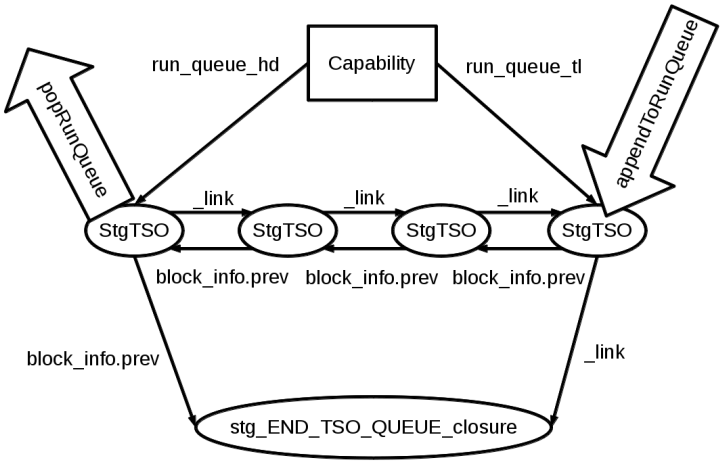
これだけだと StgTSO だけあれば十分じゃなイカと思うでゲソが、threaded な RTS の場合にはイカの図のように Task が pthread と紐付くようになるでゲソ。今回は pthread を使わない RTS なので、簡素な実装で済んでいるんでゲソ。



2.4.1 rts_evalLazyIO 関数でプログラム本体を実行

```
hs_main => real_main => rts_evalLazyIO => scheduleWaitThread => schedule
```

のような順序で呼び出される schedule という関数の中で Haskell のスレッドをスケジュール実行するでゲソ。当然、スケジュール実行するからには run queue があり、それは Capability に紐づいた StgTSO の queue でゲソ。



スケジュール実行される StgTSO は何をするかというと、あらかじめ StgTSO のスタックにやるべきことが積まれてから実行されるんでゲソ。

```
/* rts/RtsAPI.c */
INLINE_HEADER void pushClosure   (StgTSO *tso, StgWord c) {
  tso->stackobj->sp--;
  tso->stackobj->sp[0] = (W_) c;
}
/* --snip-- */
StgTSO *
createIOThread (Capability *cap, nat stack_size,  StgClosure *closure)
{
  StgTSO *t;
  t = createThread (cap, stack_size);
  pushClosure(t, (W_)&stg_ap_v_info);
  pushClosure(t, (W_)closure); /* = ZCMain_main_closure */
  pushClosure(t, (W_)&stg_enter_info);
  return t;
}
```

stg_*と ZCMain_main_closure というシンボルを StgTSO のスタックに積んでいるでゲソ。ZCMain_main_closure とはなんだったかというと Main.o の中に含まれていたことを覚えていると思うでゲソ。Main.o のシンボルが StgTSO のスケジュール実行によって呼び出されそうな気がしてきたでゲソ!

```
### わからないことリスト（更新）###
* [疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？
  => 手掛かり: StgTSO を schedule 実行する中で呼び出されそう
```

2.4.2 shutdownHaskellAndExit 関数で後始末

```
hs_main => real_main => shutdownHaskellAndExit => hs_exit_ => flushStdHandles
```

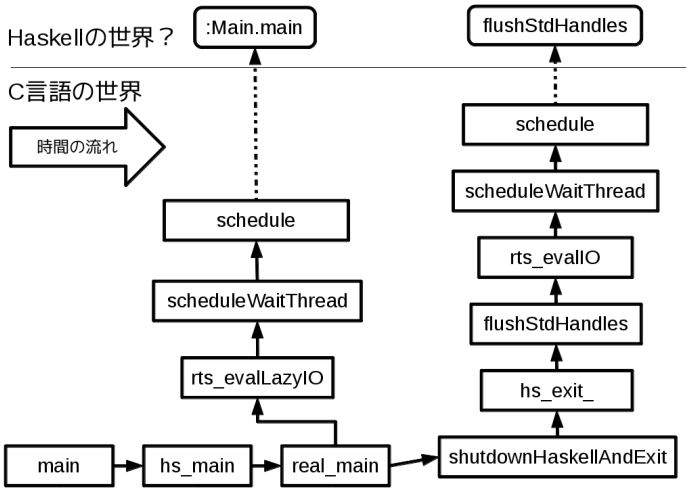
と呼び出される flushStdHandles 関数の中で再び Haskell コードがスケジュール実行されるんでゲソ。もう全部終わったかと思ったじゃないカ。今度はどんなことを StgTSO にさせるかというと……

```
/* rts/RtsAPI.c */
StgTSO *
createStrictIOThread(Capability *cap, nat stack_size,  StgClosure *closure)
{
  StgTSO *t;
  t = createThread(cap, stack_size);
  pushClosure(t, (W_)&stg_forceIO_info);
  pushClosure(t, (W_)&stg_ap_v_info);
  pushClosure(t, (W_)closure); /* = flushStdHandles_closure */
  pushClosure(t, (W_)&stg_enter_info);
  return t;
}
```

closure については後の章で解説するでゲソが、Haskell の flushStdHandles 関数が stdout と stderr のバッファを flush する関数でゲソ。Haskell の main 関数の中で stdout の flush をしなくても、RTS 終了直前に flush してくれるようでゲソ。気軽に putStr して怠惰に実行終了しても、ちゃんと RTS が flush してくれるので便利でゲッソ!

```
### わからないことリスト（更新）###
* [疑問 3] stdout バッファの flush は誰がやるのか？
  => 解決: RTS 終了直前に flushStdHandles 関数が呼び出される
```

C 言語で書かれた部分を調べることで判明したことを図にまとめてみたでゲッソ。



2.5 [漸深層] 幽霊船

……とちょっと待ってほしいでゲソ。さっきから ZCMain_main_closure なるものが出てきているんだが何者なんでゲソ？ z-encoding からひもとくと、:Main.main closure になるようでゲソ。でも:Main.main なんて関数を作った覚えがないでゲソ。その謎を解くために、GHC の-ddump-ds オプションで Main.hs をコンパイラパイプラインの Desugar まで通してみるでゲソ。

```
$ ghc -c Main.hs -ddump-ds
==================== Desugar (after optimization) ====================
Result size = 8

Main.main :: GHC.Types.IO ()
[LclIdX]
Main.main = System.IO.putChar (GHC.Types.C# 'H')

:Main.main :: GHC.Types.IO ()
[LclIdX]
:Main.main = GHC.TopHandler.runMainIO @ () Main.main
```

:Main.main 関数がコンパイル時に自動生成されているでゲソ!

```
### わからないことリスト（更新）###
* [疑問 4] :Main モジュールとは何か？
  => 解決：コンパイル時に Main.main 関数を runMainIO でラップするために自動生成される
```

2.5.1 main 実行前にシグナルハンドラ設定 (GHC.TopHandler.runMainIO)

じゃあ GHC.TopHandler.runMainIO というのは何者なんでゲソ？

```
runMainIO :: IO a -> IO a
runMainIO main =
    do
      main_thread_id <- myThreadId
      weak_tid <- mkWeakThreadId main_thread_id
      install_interrupt_handler $ do
           m <- deRefWeak weak_tid
           case m of
               Nothing  -> return ()
               Just tid -> throwTo tid (toException UserInterrupt)
      main -- hs_exit() will flush
    `catch`
      topHandler
--snip--
install_interrupt_handler :: IO () -> IO ()
install_interrupt_handler handler = do
   let sig = CONST_SIGINT :: CInt
   _ <- setHandler sig (Just (const handler, toDyn handler))
   _ <- stg_sig_install sig STG_SIG_RST nullPtr
   return ()
--snip--
topHandler :: SomeException -> IO a
topHandler err = catch (real_handler safeExit err) topHandler
```

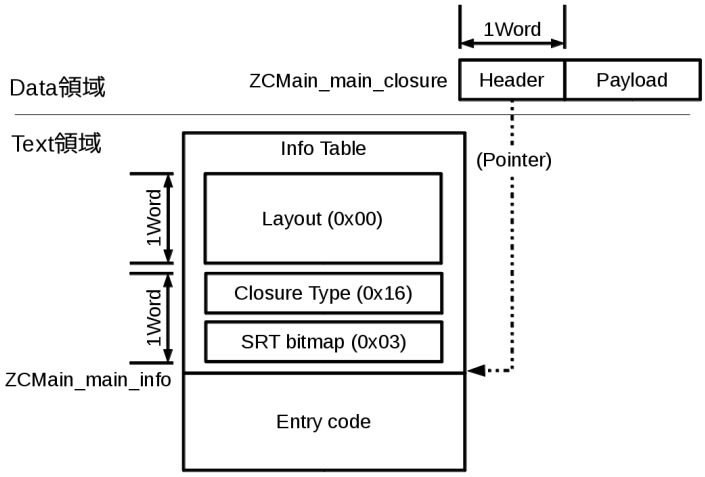
main 関数の実行前にシグナルハンドラの設定をしているでゲソ。すなわち setHandler で SIGINT が入ったら Haskell の main 関数を実行しているスレッドに UserInterrupt 例外が入るようにし、sigaction を SA_RESETHAND に、つまり一回 SIGINT が入ったらシグナルハンドラの設定をデフォルトに戻すように設定するでゲソ。そういえば GHC が吐いたバイナリは C-c を二回入力しないと終了できなかった気がするでゲソ。こんなところで日頃の疑問が解けたじゃなイカ。また、main 関数が例外終了したら real_handler を呼び出して後始末をするでゲソ。

わからないことリスト（更新）###
*** ［疑問 1］ そもそも `main` 関数がどこからどうやって呼び出されるのか？**
=> 解決: `StgTSO` が実行開始すると `:Main.main` から `runMainIO` 経由で呼び出される

2.6 Haskell の関数はどんなメモリ配置？

ちょっとコードの解析を休んで、これでまで説明を避けてきた末尾に closure や info が付いたシンボルについて解説するでゲソ。この図はコンパイル済み:Main.main 関数のメモリレイアウトでゲソ。[*5]

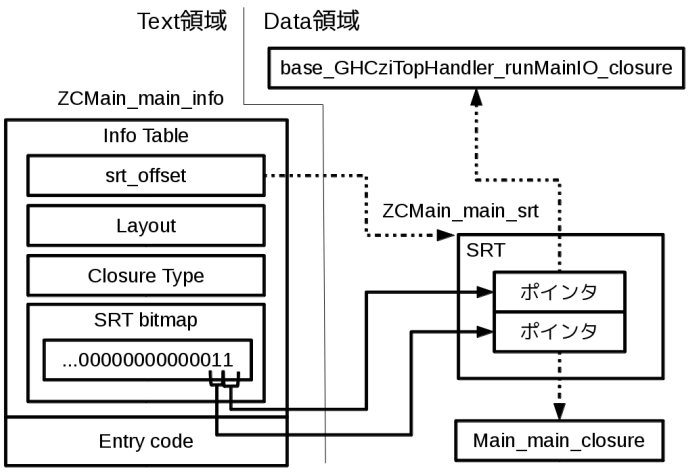
[*5] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/HeapObjects



Haskell の関数は HOGE_closure と HOGE_info という二つのシンボルにコンパイルされるでゲソ。HOGE_closure の方が親玉で、先頭 1 ワードには HOGE_info へのポインタが入っているでゲソ。HOGE_info の高位のアドレス方向には機械語のコードが格納されていて、そして低位のアドレス方向には Info Table という closure の形式を見わけるためのヒントが格納されているでゲソ。一つのポインタで二度おいしいとはすごいしくみじゃなイカ!

Closure Type には closure の形式が入っているでゲソ。:Main.main の場合 0x16 が入っているので、includes/rts/storage/ClosureTypes.h ヘッダの定義によると THUNK_STATIC ということになるでゲソ。THUNK というのはみんな知っている通り遅延されて未評価の式でゲソ。STATIC というのは動的に確保されない thunk という意味でゲソ。:Main.main はトップレベルで引数がないので thunk は 1 つしか作られないはずでゲソ。Layout は Payload がどのような形式になっているのか表わしているでゲソ。

今度は SRT bitmap に注目して図を描いてみたでゲソ。



SRT bitmap はその Info Table から辿れる SRT の中のどのワードまで有意なポインタが入っているのか表わしているでゲソ。GC する際に SRT 内のポインタを素早く辿るために使う典型的な手法でゲッソ。SRT(Static Reference Table) というのは Entry code 内で使用している closure へのポインタのリストのようでゲソ。

少し話がそれるでゲソが GHC のソースコードやドキュメントでは bitmap という単語は 2 つの意味で使われるでゲソ。1 つはさっき説明した SRT bitmap でゲソ。もう 1 つは Payload が Bitmap layout という形式であった場合に、その Payload のサイズとどの部分がポインタかを表わす bitmap でゲソ。この Payload の bitmap は SRT bitmap とは形式も異なり、上位のビット列に Payload のサイズが格納され、下位のビット列にポインタ/非ポインタかを表わすビットが格納されているでゲソ。混同しがちなので気をつけるでゲッソ!

```
### わからないことリスト（更新）###
* [疑問 5] closure, info とは何か？
  => 解決: Haskell の関数をコンパイルすると作られ、info に機械語コードが格納されている
```

2.7 [深海層] cmm 言語の世界

なんとなく解ったような気になったでゲソ。でも、C 言語の schedule 関数によるコンパイル済み Haskell 関数のスケジュール実行のイメージがさっぱりつかめないじゃなイカ？ 落ち着いて C 言語の schedule 関数から追い掛けてみるでゲッソ。

2.7.1 :Main.main 関数はどこでゲソ!?

```
/* schedule 関数 (rts/Schedule.c) */
static Capability *
schedule (Capability *initialCapability, Task *task)
{
/* --snip-- */
  while (1) {
/* --snip-- */
    switch (prev_what_next) {
/* --snip-- */
    case ThreadRunGHC:
    {
        StgRegTable *r;
        r = StgRun((StgFunPtr) stg_returnToStackTop, &cap->r);
        cap = regTableToCapability(r);
        ret = r->rRet;
        break;
    }
```

schedule 関数は StgTSO を実行可能であれば、StgRun 関数を呼び出すでゲソ。

```
/* StgRun 関数 (rts/StgCRun.c) */
#define STG_RUN "StgRun"
/* --snip-- */
static void GNUC3_ATTRIBUTE(used)
StgRunIsImplementedInAssembler(void)
{
    __asm__ volatile (
        /*
         * save callee-saves registers on behalf of the STG code.
         */
        ".globl " STG_RUN "\n"
        STG_RUN ":\n\t"
        "subq %0, %%rsp\n\t"
        "movq %%rsp, %%rax\n\t"
        "addq %0-48, %%rax\n\t"
        "movq %%rbx,0(%%rax)\n\t"
        "movq %%rbp,8(%%rax)\n\t"
        "movq %%r12,16(%%rax)\n\t"
        "movq %%r13,24(%%rax)\n\t"
        "movq %%r14,32(%%rax)\n\t"
        "movq %%r15,40(%%rax)\n\t"
        /*
         * Set BaseReg
         */
        "movq %%rsi,%%r13\n\t"
        /*
         * grab the function argument from the stack, and jump to it.
         */
        "movq %%rdi,%%rax\n\t"
        "jmp *%%rax\n\t"
```

StgRun 関数はアセンブラで書かれていて、しかも長いでゲソが、やってることは簡単でゲソ。x86-64 の ABI として保存すべきレジスタ[*6] を C 言語のスタックに保存し、Capability の r メンバー (StgRegTable 型) へのポインタを r13 レジスタに入れてから stg_returnToStackTop にジャンプするだけでゲソ。

この StgRegTable には仮想 CPU で用いるレジスタが入っているでゲソ。ただし全ての仮想 CPU レジスタがメモリに置かれている訳ではなくて、割り当て可能なものはなるべく物理 CPU のレジスタを使って速度をかせぐでゲソ。[*7] includes/stg/MachRegs.h ファイルの x86_64_REGS を見れば x86-64 での割り当てがわかるでゲッソ。cmm 言語をコンパイルしたバイナリを逆アセンブルする際には注意でゲソ。

[*6] http://www.x86-64.org/documentation/abi.pdf

[*7] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/HaskellExecution/Registers

ジャンプ先の stg_returnToStackTop のソースコードを見てみると……

```
/* stg_returnToStackTop 関数 (rts/StgStartup.cmm) */
#define CHECK_SENSIBLE_REGS() \
    ASSERT(Hp != 0);            \
    ASSERT(Sp != 0);            \
    ASSERT(SpLim != 0);         \
    ASSERT(SpLim - WDS(RESERVED_STACK_WORDS) <= Sp);
/* --snip-- */
stg_returnToStackTop
{
  LOAD_THREAD_STATE();
  CHECK_SENSIBLE_REGS();
  jump %ENTRY_CODE(Sp(0));
}
```

これが cmm 言語 [*8] でゲソ。 cmm 言語とは C 言語モドキの見た目を持っている高級アセンブラで、限定用途の switch 文は使えるでゲソが、ループ構文はないでゲソ。cmm 言語の中では実際の CPU レジスタではなく、さっきでてきた仮想 CPU レジスタを使ってプログラミングするでゲソ。スタックポインタも仮想 CPU のものなので、cmm 言語で使われるスタック領域は C 言語とは別の領域になるでゲソ。compiler/cmm/CmmParse.y が cmm 言語のパーサなんでゲソが、基本文法とは別に専用の命令も定義されていたりするでゲソ。さらに Haskell の関数は長いコンパイルパイプラインを通って一旦 cmm 言語に変換された後に機械語になるでゲソ。GHC をささえる縁の下の力持ちでゲソね!

ところで stg_returnToStackTop 関数の 3 行が全く読めないでゲソ……。LOAD_THREAD_STATE と ENTRY_CODE は compiler/cmm/CmmParse.y によって解釈される拡張文法らしいでゲソ。なんとか compiler/codeGen/CgForeignCall.hs を読みながらカンで読んでみるでゲソ。

0. BaseReg(x86-64 だと r13 レジスタ) から StgRegTable へのポインタを得る
1. StgRegTable の rCurrentTSO メンバーを tso 変数に
2. tso の先にある StgTSO の stackobj メンバーを stack 変数に
3. StgRegTable の Sp メンバーを stack->sp で初期化
4. StgRegTable の SpLim メンバーを stack->stack + RESERVED_STACK_WORDS で初期化
5. StgRegTable の rHpAlloc メンバーを 0 で初期化
6. StgRegTable の Hp メンバーを初期化
7. StgRegTable の HpLim メンバーを初期化
8. スタックとヒープが正常かチェック
9. StgRegTable の Sp メンバーの先頭にジャンプ

つまり BaseReg の中にあるスタックとヒープの情報を初期化して StgTSO のスタック先頭にジャンプするんでゲソ。スタック先頭はなんだったかというと stg_enter_info だったじゃなイカ？ createIOThread 関数で積んでいたハズでゲソ。

[*8] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/CmmType

```
/* includes/Cmm.h */
#define Sp(n)  W_[Sp + WDS(n)]
#define Sp_adj(n) Sp = Sp + WDS(n)
#define TAG_BITS                3
#define TAG_MASK ((1 << TAG_BITS) - 1)
#define GETTAG(p) (p & TAG_MASK)
#define LOAD_INFO                                   \
  if (GETTAG(P1) != 0) {                            \
      jump %ENTRY_CODE(Sp(0));                      \
  }                                                 \
  info = %INFO_PTR(P1);
#define ENTER()                                       \
 again:                                               \
  W_ info;                                            \
  LOAD_INFO                                           \
  switch [INVALID_OBJECT .. N_CLOSURE_TYPES]          \
         (TO_W_( %INFO_TYPE(%STD_INFO(info)) )) { \
/* snip */                                            \
  default:                                            \
   {                                                  \
      UNTAG_R1                                        \
      jump %ENTRY_CODE(info);                         \
/* rts/HeapStackCheck.cmm */
INFO_TABLE_RET( stg_enter, RET_SMALL, P_ unused)
{
    R1 = Sp(1);
    Sp_adj(2);
    ENTER();
}
```

stg_enter_info でやっていることを気合いで読むでゲソ!

0. StgTSO のスタック 2 ワード目を R1 レジスタに取り出す (= closure)
1. StgTSO のスタックポインタを 2 ワード戻す
2. R1 レジスタのタグビットが立っていたらスタックポインタ先頭の差すアドレスにジャンプ
3. R1 レジスタから Info Table のアドレスを得る (= info)
4. Info Table の Closure Type によって処理を切り換え
5. Closure Type が特殊なものでなかった場合 Entry code へジャンプ

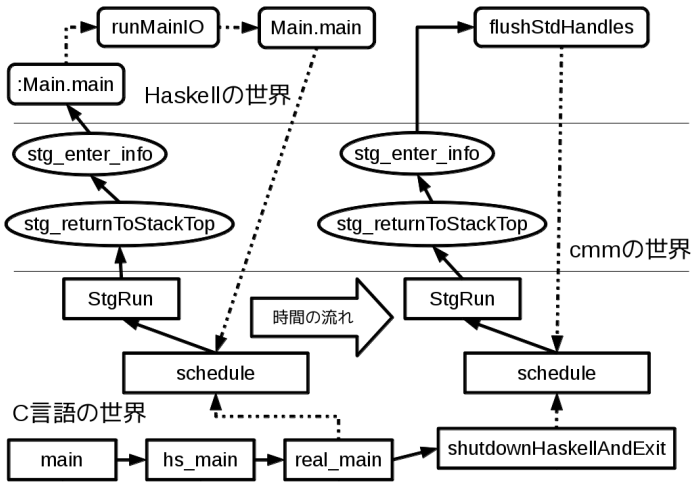
今回のケースではスタックの二番目に積まれていたのは ZCMain_main_closure なので、ZCMain_main_closure の先頭アドレス (Header) から ZCMain_main_info(Info Table) を辿り、 Closure Type が THUNK_STATIC なので ZCMain_main_info にそのままジャンプするんじゃなイカ？

結局 stg_enter_info は「StgTSO の次のスタックに積まれた Entry code をいい感じに実行する」という機能を持っているみたいでゲソ。

ということで、やったでゲソ! 遂に Haskell で書かれた:Main.main 関数に到達したでゲソ!

```
### わからないことリスト（更新）###
* [疑問 2] Haskell の関数はどうやって他の関数を呼び出すのか？
  => 手掛かり: 最初の関数は stg_enter_info から呼び出される
* [疑問 6] stg_*とは何か？
  => 手掛かり: stg_enter_info:=スタックに積まれた Entry code を実行する
```

ちょっと疲れたので、一旦休憩をかねてここまでの軌跡を図にまとめるでゲッソ。



こうして眺めてみると RTS というか GHC が吐き出す実行バイナリは大きく 2 つの領域に分断されていることがわかるでゲソ。それは C 言語と cmm 言語でゲソ。上の図には Haskell で書かれた領域も出てくるでゲソが、Haskell も結局は cmm 言語に変換されるので同じ領域と考えても良いでゲソ。GHC が吐き出すバイナリが実行開始すると、まず C 言語で記述されたコードが走り出すでゲソ。この領域では CPU が決めた ABI で動いているでゲソ。例えば x86/x64 の場合にはスタックも ABI の通りなので、スタックトレースが取れるでゲソ。ところが、StgRun 関数を経由して cmm 言語で記述されたコードが走りはじめると、そのコードが自発的に C 言語コードを呼び出したり schedule 関数に戻らない限り、CPU が決めた ABI ではなく GHC が自分自身のために作り出した仮想 CPU によって決めた ABI によって関数呼び出しを行なうでゲソ。もっとも小さなレベルでは、この関数呼び出しは stg_enter_info のような cmm 言語で書かれたアドレスへのジャンプの繰り返しに見えたでゲソ。このような ABI は x86/x64 の ABI ではないので、gdb などのデバッガを通してみてもスタックトレースは取れないでゲソ。参照透明な海の力はハードウェアが決めた法則さえ再定義するんでゲソ!

2.7.2 関数から関数へジャンプ! (:Main.main => runMainIO)

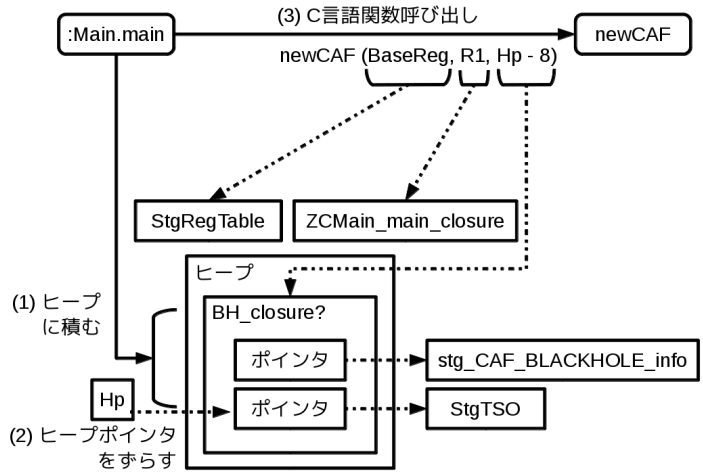
さっき辿り着いた:Main.main 関数の Entry code(ZCMain_main_info) からふたたび出発でゲソ! ZCMain_main_info の cmm ソースコードは ghc の-ddump-cmm オプションを使えば手に入るでゲソ。

```
ZCMain_main_info()
        { update_frame: <none>
          label: ZCMain_main_info
          rep:HeapRep static { Thunk }
        }
    cg2:
        if (Sp - 16 < SpLim) goto cg4;
        Hp = Hp + 16;
        if (Hp > HpLim) goto cg6;
        I64[Hp - 8] = stg_CAF_BLACKHOLE_info;
        I64[Hp + 0] = CurrentTSO;
        (_cg7::I64,) = foreign "ccall"
          newCAF((BaseReg, PtrHint), (R1, PtrHint), (Hp - 8, PtrHint));
        if (_cg7::I64 == 0) goto cg8;
        goto cg9;
    cg4: jump stg_gc_enter_1 ();
    cg6:
        HpAlloc = 16;
        goto cg4;
    cg8: jump I64[R1] ();
    cg9:
        I64[Sp - 16] = stg_bh_upd_frame_info;
        I64[Sp - 8] = Hp - 8;
        R1 = base_GHCziTopHandler_runMainIO_closure;
        R2 = Main_main_closure;
        Sp = Sp - 16;
        jump stg_ap_p_fast ();
}]
```

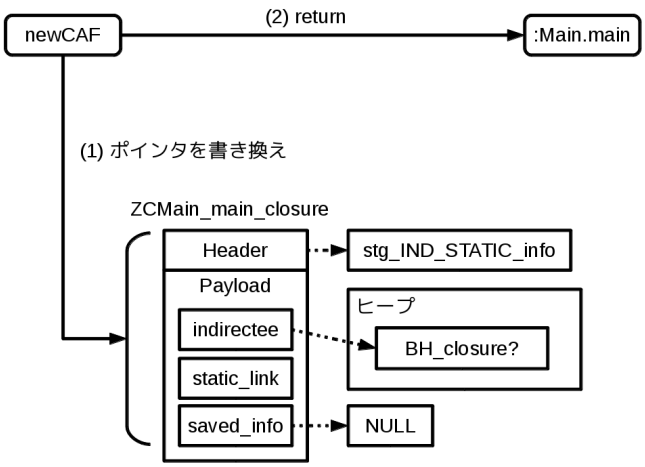
:Main.main 関数の Entry code 前半ではヒープにポインタを格納してから C 言語の newCAF 関数を呼び出すでゲソ。newCAF のコードを読むと判明するでゲソが、このヒープに置いたポインタは動的に作成した closure として使われるようでゲソ。ここではこの作成した closure は BH_closure と名前を付けて説明するでゲソ。

ヒープポインタ (Hp) はスタックポインタ (Sp) と違い、高位のアドレス方向に成長するんでゲソ。ここで、ヒープポインタ (Hp) とスタックポインタ (Sp) をずらす前にそれぞれ HpLim レジスタと SpLim レジスタと比較して、領域確保可能かどうか判別しているでゲソ。もしヒープやスタックあふれを引き起こすようであれば、stg_gc_enter_1 (本体は GC_GENERIC という define マクロ、rts/HeapStackCheck.cmm で定義) を経由して一旦 C 言語の schedule 関数に戻るでゲソ。schedule 関数は仮想 CPU の rRet レジスタから StgTSO 中断原因を調べて、領域を再割り当てしてくれたり GC を実行してくれたりするでゲソ。

もちろん今回の実行ではヒープもスタックもあふれないので、そのまま newCAF 関数を呼び出すでゲソ。



newCAF 関数 (rts/sm/Storage.c) に入ると、:Main.main 関数の closure を書き換え、:Main.main に return するでゲソ。



CAF(Constant Applicative Form) というのは「一度実行したら、その結果をメモ化して使いまわせるもの」のことでゲソ。例えば、無引数の関数はメモ化できるので、CAF になるでゲソ。CAF かどうかは単純に「無引数の関数としてコンパイルされる」かどうかで決まって、IO モナドかどうかには関係しないので、今回の:Main.main も CAF なんでゲソ。

「無引数の関数として定義する」ことではなく、「無引数の関数としてコンパイルされるかどうか」が重要になる点に注意するでゲソ。今回の話題の範囲外になるので深く説明はできないでゲソが、無引数の関数として定義してもコンパイル時にそうでなくなったり、:Main.main 以外にもコンパイル時に無引数の関数が生成されることなんかもあるでゲソ。[*9] なので、どれが CAF かを判断するには、コンパイラの生成するコード (Core 言語や STG 言語) などを見て確認する必要があるでゲソ。

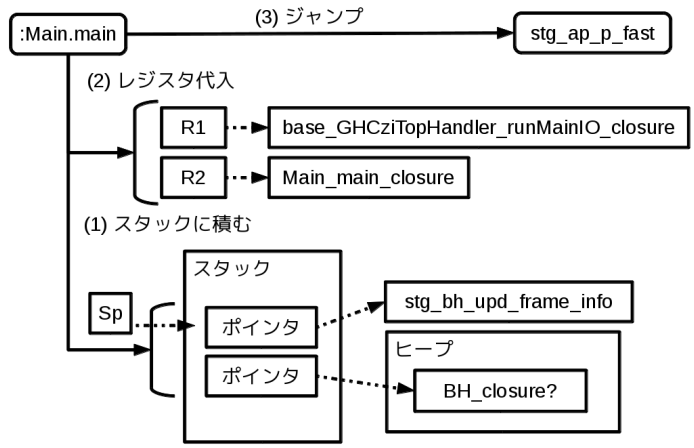
[*9] http://stackoverflow.com/questions/8330756/what-are-super-combinators-and-constant-applicative-forms

newCAF 関数はこのメモ化に必要な準備をするでゲソ。newCAF 関数を読んだ段階ではまだ thunk を潰した結果が得られていないので、closure に BLACKHOLE という印を付けておくんでゲソ。そして closure に対応する thunk の評価が完了した段階になったら、今まで BLACKHOLE だった closure を評価済みの値で上書きするんでゲソ。[*10] threaded な RTS を使う場合には同時に別々の Task から thunk を触るケースが考えられるのでこの BLACKHOLE のしくみは大変重要でゲソ。

仮に thunk を潰している最中 (BLACKHOLE 状態) に別の Task から closure が呼び出された場合、stg_IND_STATIC_info(rts/StgMiscClosures.cmm) は stg_CAF_BLACKHOLE_info を呼び出し、messageBlackHole 関数 (rts/Messages.c) を使って BLACKHOLE 状態にある closure の評価待ち queue を作るようでゲソ。そしてブロックする準備ができたら ThreadBlocked を要因として schedule に戻るようでゲソ。すると schedule 関数は BLACKHOLE によってブロックされた StgTSO を再び run queue には繋げずに run queue から別の StgTSO を取り出してスケジュール実行を再開するでゲソ。ということはこの BLACKHOLE の closure に触ったことを要因とするブロックは誰か別の StgTSO が thunk を評価完了し終わったら、ブロック解除されるはずだと推測できるでゲソ。ここらへんはちょっと理解不足でゲソ。

```
### わからないことリスト（更新）###
* [疑問 6] newCAF とは何か？
  => 解決: newCAF 関数はメモ化に必要な器を closure に確保する
```

:Main.main 関数の Entry code 後半ではスタックに stg_*関数へのポインタを積み、R1 レジスタに呼び出し関数を R2 レジスタに引数へのポインタを代入して、stg_ap_p_fast へジャンプするでゲソ。



stg_ap_p_fast のソースコードは、GHC をソースコードからコンパイルするとできる inplace/bin/genapply というコマンドを実行することでソースを見れるでゲソ。さらにこの stg_ap_p_fast で使っている StgFunInfoExtraRev_arity マクロは inplace/bin/mkDerivedConstants というコマンドによって自動生成されるでゲソ。

この genapply コマンドは多量の cmm 関数を自動生成するでゲソ。さらっと見るかぎりでは

[*10] http://blog.ezyang.com/2011/05/reified-laziness/

Haskell 関数をコンパイルしてできる closure を動作させるのに必要な語集が stg_*関数のように見えるでゲソ。それらの語集の意味を知るにはもっと調査が必要そうでゲソね。

と、ここまで潜ってきて cmm コードを一つ一つ解釈するのに息切れしてきたでゲソ。参照透明で新鮮な空気を吸いた……いでゲソ。なんか楽をする方法はなイカ？ 手っ取り早く gdb で追ってみるでゲソ。[*11] gdb で実行する際には +RTS -V0 オプションを付けるといいでゲソ。Haskell スレッドのタイムスライスを実現するためのタイマーシグナルを無効にできるでゲソ。gdb で break している最中にタイマーシグナルが入ってしまうと、本来の動作を観測できなくなるでゲソ。

```
$ gdb Main
GNU gdb (GDB) 7.4.1-debian
--snip--
Reading symbols from /home/kiwamu/src/DiveIntoRTS/Main...done.
(gdb) b ZCMain_main_info
Breakpoint 1 at 0x404278
(gdb) run +RTS -V0
Starting program: /home/kiwamu/src/DiveIntoRTS/Main +RTS -V0
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0000000000404278 in ZCMain_main_info ()
(gdb) si
0x000000000040427c in ZCMain_main_info ()
(gdb) si
0x000000000040427f in ZCMain_main_info ()
```

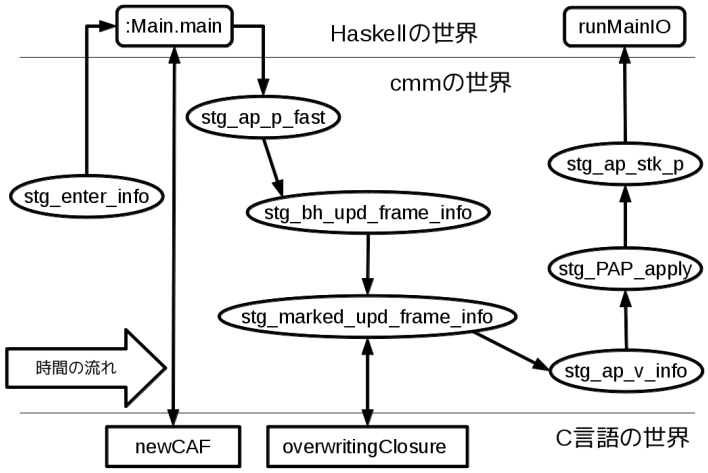
こんな感じにひたすらステップ実行していけばいつかは base_GHCziTopHandler_runMainIO_info に辿り着くはずでゲソ。頭使わなくて済んで楽勝じゃなイカ。

gdb で追いかけた関数の履歴

```
=> ZCMain_main_info
=> newCAF (C 言語関数)
=> stg_ap_p_fast
=> stg_bh_upd_frame_info
=> stg_marked_upd_frame_info
=> overwritingClosure (C 言語関数)
=> stg_ap_v_info
=> stg_PAP_apply
=> stg_ap_stk_p
=> base_GHCziTopHandler_runMainIO_info
```

やはり closure と closure の間は stg_*関数のジャンプによって繋いでくれるようでゲソ。この呼び出し関係を図にしてみるとイカのようなになるでゲソ。

[*11] http://hackage.haskell.org/trac/ghc/wiki/Debugging/CompiledCode




```
### わからないことリスト（更新）###
* [疑問 7] stg_*とは何か？
  => 解決: Haskell 関数をコンパイルしてできる closure を動作させるのに必要な語集
```

この後どのように実行が進むかというと、コンパイルされた Haskell 関数 (closure) を stg_*などの RTS 側で準備された Entry code が間を繋げながら実行が進むんでゲソ。各 Entry code 間はジャンプ命令によって呼び出しが行なわれるのでゲソ。そのため、本来のスタックは使わず、StgTSO のスタックのみを使用することになるでゲソ。

2.8 浮上でゲッソ

cmm 言語の世界まで潜ったところで、今回の潜水を振り返ってみるでゲソ。RTS は C 言語と cmm 言語で実装されていたでゲソ。GHC の吐くコンパイル済みバイナリは C 言語の main から RTS の schedule 関数を呼び出すでゲソ。schedule 関数は Haskell のスレッドである StgTSO をスケジュール実行していたでゲソ。StgTSO の実行に入ると C 言語の ABI から cmm 言語の ABI に切り替わり、コンパイル済みの Haskell 関数である closure を実行していたでゲソ。

さて、疑問は全部解決したんでゲソ？

```
### わからないことリスト（最終結果）###
* [疑問 1] そもそも main 関数がどこからどうやって呼び出されるのか？
  => 解決: StgTSO が実行開始すると:Main.main から runMainIO 経由で呼び出される
* [疑問 2] Haskell の関数はどうやって他の関数を呼び出すのか？
  => 未解決!
* [疑問 3] stdout バッファの flush は誰がやるのか？
  => 解決: RTS 終了直前に flushStdHandles 関数が呼び出される
* [疑問 4] :Main モジュールとは何か？
  => 解決: コンパイル時に Main.main 関数を runMainIO でラップするために自動生成される
* [疑問 5] closure, info とは何か？
```

=> 解決: Haskell の関数をコンパイルすると作られ、info に機械語コードが格納されている
* [疑問 6] newCAF とは何か？
=> 解決: newCAF 関数はメモ化に必要な器を closure に確保する
* [疑問 7] stg_*とは何か？
=> 解決: Haskell 関数をコンパイルしてできる closure を動作させるのに必要な語集

ぬ？ 疑問 2 が解決してないじゃなイカ! この疑問 2 を解決するためには大量にある stg_*関数をどう組み合わせればどのような機能が実現できるのか、その大ざっぱな仕様について調査する必要があるでゲソ。かなり古い論文でゲソが、 “Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine” [*12] を読んで、設計思想のようなものを理解してから cmm 言語を読むと理解が速いかもしれないでゲソね。次回はもっと準備をしてから潜ろうと思うでゲッソ!

この夏、みんなも「わからないこと」を探しに RTS の海に潜ってみなイカ？

2.9 謝辞

今回の記事では筆者の調査結果が正しいのか自信が持てなかったので、サークル外の方々にレビューをお願いしたでゲソ。

@tyasunao さんには cmm 言語と STG 言語の基本的な読み方を教えてもらったでゲソ。@shelarcy さんには CAF について教えてもらったでゲソ。実は CAF の説明の文章は @shelarcy さんにそのまま書いてもらったでゲソ。

紙面を通じて感謝でゲソ! 本当にありがとうでゲソー!

2.10 参考文献

- GHC 7.4.1 のソースコード [*13]
- suztomo の日記「Haskell のコンパイル中に現れる STG と, GDB で見る C-backend な C–」 [*14]
- INSIDE 233「Tracing the compilation of Hello Factorial!」 [*15]
- GHC Developer Wiki「I know kung fu: learning STG by example」[*16]
- GHC Developer Wiki「Object Structure」 [*17]
- github: master-q/DiveIntoRTS (本記事での調査結果詳細) [*18]
- GHC ソースコードリーディング勉強会 #readghc のまとめ [*19]

[*12] http://research.microsoft.com/apps/pubs/default.aspx?id=67083
[*13] https://github.com/ghc/ghc/tree/ghc-7.4.1-release
[*14] http://d.hatena.ne.jp/suztomo/20111224/1324718354
[*15] http://blog.ezyang.com/2011/04/tracing-the-compilation-of-hello-factorial/
[*16] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/GeneratedCode
[*17] http://hackage.haskell.org/trac/ghc/wiki/StgObjectTypes
[*18] https://github.com/master-q/DiveIntoRTS
[*19] http://wiki.haskell.jp/Workshop/ReadGHC/

第3章

評価

— @xhl_kogitsune

「お前たち! 今日も元気に簡約するでゲソ!!! タイトルを見て千反田とかいう奴が出てくることを期待していたのなら残念だったでゲソね! ここは私が侵略済みでゲソ!!

今日は式の**評価**の話をするでゲソ! 評価、あるいは**簡約**、というのはここでは計算の 1 ステップみたいなものでゲソ。たとえば式 `3 * (1 + 2)` を 1 回評価 (簡約) すると式 `3 * 3` になって、更に 1 回評価 (簡約) すると値 `9` が得られる、みたいな感じでゲソ。

一言に評価と言っても、やり方は色々あるでゲソ。式のどこから評価をしていくか、という**評価順序**も色々あるし、1 回の評価をどう実装するかも色々あるでゲソ。今日は、私オススメの評価順序である Normal Order 評価の話をしつつ、答えが出るまでにかかる評価回数を減らすにはどのように 1 回の評価を実装すればいいかという話をするでゲソ。

まず式 1 の評価を考えるでゲソ。ここでは Haskell 風に書いたでゲソが、`(\x -> x * x)` という「x を受け取って $x*x$ を返す関数」を $(1+2)$ に適用しているでゲソ。

式 1: Haskell

```
(\x -> x * x) (1 + 2)
```

OCaml だと

式 1: OCaml

```
(fun x -> x * x) (1 + 2)
```

C++ だと

式 1: C++

```
int f(int x){ return x * x; }
int main(){
...
int r = f(1 + 2);
...
}
```

こんな感じでゲソ (r の値を計算するでゲソ)。今日は一番最初の Haskell 風表記を使って説明するでゲソ。評価順序とかは実際の Haskell 実装とは違うことがあるので注意でゲソ! 式 1 に出てきた、関数、関数適用 、四則演算の書き方くらい把握しておけばだいたいサンプルコードは読めるんじゃなイカ?

さて、実は式の評価の順番 (評価順序) には色々あるでゲソ。この式を、どこから評価するでゲソか? C++ や OCaml なら、まず `1 + 2` を計算して `3` を出すところからはじめるんじゃなイカ? 然る後に関数 (`\x -> x * x`)[*1] を呼び出し、`x = 3` として `x * x` → `3 * 3` → `9` という風になるんじゃなイカ? これを式変形の過程として書くと図 3.1 のようになるでゲソ。

```
(\x -> x * x) (1 + 2)
→ (\x -> x * x) 3
→ 3 * 3
→ 9
```

図 3.1: 式 1: Applicative Order 評価、文字列ベース簡約

行頭の矢印 (→)1 個分が、評価 1 回分に対応するでゲソ。それぞれの評価ステップでは、下線を引いた部分を評価しているでゲソ:

1. まず最初に、`(1 + 2)` を評価して `3` に置き換えるでゲソ。
2. 次に、`(\x -> x * x)` の `3` への適用を評価するでゲソ。具体的には、関数の中身 `x * x` の中の仮引数 `x` を実引数 `3` に置き換えて、`3 * 3` を作るでゲソ。
3. 後は `3 * 3` を評価して完了じゃなイカ!

これはよく使われる評価順序の一つで、関数の引数を評価してから関数適用を評価する、**Applicative Order (作用的順序)** と呼ばれるものでゲソ。今回は 3 ステップで評価できたでゲソ。Call By Value の時に使われる評価順序はこんな感じでゲソ。手続き型言語の大半はこんな感じでゲソ。

しかし、世の中 Applicative Order ばかりじゃないでゲソ。これ以外に方法なんてあるでゲソ? と思う人もいるかもしれないでゲソ。でも、引数の評価を後回しにして関数適用を先に評価する、**Normal Order (正規順序)** という評価順序もあるでゲソ。Haskell なんかはそんな感じの順序でゲソ。式 1 を Normal Order で評価すると図 3.2 のようになるでゲソ。

```
(\x -> x * x) (1 + 2)
→ (1 + 2) * (1 + 2)
→ 3 * (1 + 2)
→ 3 * 3
→ 9
```

図 3.2: 式 1: Normal Order 評価、文字列ベース簡約

1. 最初に、`(\x -> x * x)` の `(1 + 2)` への適用を評価するでゲソ。`(1 + 2)` はそのままにして、関数の中身 `x * x` の中の仮引数 `x` を実引数 `(1 + 2)` に置き換えて、`(1 + 2) * (1 + 2)` を作るでゲソ。慣れていない人には、Applicative Order の時は値 (3) だったところに式 (`(1 + 2)`) が無理やり入っているように見えるかもしれないでゲソが、入れてしまえば何ともないでゲソ。

[*1] OCaml 版なら `(fun x -> x * x)`、C++ 版なら `f`

2. それから、(1 + 2) の評価をして (左側)、
3. それから、(1 + 2) の評価をして (右側)、
4. 最後に 3 * 3 の評価をして、最終的には同じ答え 9 を得るでゲソ。

これが私の好きな Normal Order でゲソ! **遅延評価 (Lazy Evaluation)** の時に使われる評価順序はこんな感じでゲソ。引数の評価を遅延させてとりあえず関数適用の評価をしてしまうでゲソ。Normal Order には、後で説明するでゲソが、「とりあえず Normal Order を使っておけば間違いは起こらない」という安心感があるでゲソ。今日の話の前半ではそんな Normal Order の宣伝をするでゲソ!

しかし、今回は 4 ステップかかったでゲソ。Applicative Order の時と比べて 1 ステップ増えてしまったでゲソ…。でも、大丈夫でゲソ! 後半では評価の回数を減らす方法を紹介するでゲソ!

3.1 準備しなイカ?

ここで、下準備として基本的な用語とかを説明しておくでゲソ。今日の話は、λ 計算における「λ 式から、正規形に至るまでの評価 (β 簡約)の回数」についての話に基づいているでゲソ。何のことか分からなくても心配ないでゲソ。おおざっぱには「上に出したようなHaskell 風の式から、結果の値 (整数)が出るまでの評価 (簡約)の回数」くらいに考えておけばいいでゲソ。この二つには違う面もあるでゲソが、今日はそこには踏み込まないので気にしなくていいでゲソ[*2]。

3.1.1 式

まず、評価をしたい**式**とは何かをはっきりさせようじゃなイカ! 今日使う Haskell 風の式では「変数」「関数 (λ 抽象)」「関数適用」「四則演算」「整数定数」の 5 種類のことが書けるでゲソ[*3]。λ 計算では、このうち「変数」「関数 (λ 抽象)」「関数適用」の 3 種類だけから構成される **λ 式**を扱うでゲソ。構文や意味とかをまとめるとこんな感じになるでゲソ。カッコは適宜省略するでゲソ。λ 式は Haskell 式と表記がちょっと違うだけで怖くないでゲソよ♪

表 3.1: 式

種類	λ 式	Haskell 風の式	意味
変数	x	`x`	変数 x
関数/λ 抽象	$(\lambda x.A)$	`(\x -> A)`	x を受け取って A を返す関数
関数適用	(AB)	`(A B)`	関数 A を引数 B に適用する
四則演算	なし	`+` など	
数値定数	なし	`1` など	

この A とか B の所には実際には λ 式や Haskell 風の式が入るでゲソ。たとえば、Haskell 風の式での `(\x -> x * x)` は、「引数 `x` を受け取ってその 2 乗 `(x * x)` を返す関数」でゲソ。この場合、`A` は `(x * x)` に相当するでゲソ。これを式 `(1 + 2)` に適用すると、`((\x -> x * x) (1 + 2))` になるでゲソ。この式の値は $(1+2)*(1+2)=9$ になるはずでゲソ。これで簡単な整数演算は書き放

[*2] たとえば、λ 計算では全て先行評価でも遅延評価でもできるでゲソが、Haskell とかの式の四則演算とかは普通、被演算数の値を先に評価する必要があるので先行評価が必要でゲソ。こういう話は [SPJ 1987] あたりに書いてあるでゲソ。

[*3] 純粋な λ 式だと 1+2 とかすら書けなくて読みにくくなるので、四則演算とかを足して読みやすい例を作るでゲソ

題じゃなイカ?[*4] λ 式でも同様でゲソ[*5]。

Haskell「風」と書いたのは、Haskell だと型エラーになる式が出てきたり、実際の評価順が Haskell と違う場合があるからでゲソ。注意するでゲソ。

3.1.2 評価の 1 ステップと redex

今日は後半で「評価の回数」を減らす話をするでゲソ。で、「評価 1 回」って何でゲソ?「計算の 1 ステップ」みたいなもの、と言ったでゲソが、ここでもう少しきちんと定義しておこうじゃなイカ!

λ 計算の場合は、評価は β **簡約**という、関数適用の計算 1 回分に相当する操作でゲソ。これは、

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

という形をした λ 式 (これを簡約可能式 (reducible expression)、略して **redex** と呼ぶでゲソ) を、「A の中にある x を B で置き換えたもの」(これを $A[B/x]$ と表記するでゲソ) に置き換える操作でゲソ。関数の中の仮引数を実引数に置換する感じじゃなイカ! たとえば $((\lambda x.x)y)$ の場合、$A = x, B = y$ で、これを β 簡約すると「x の中にある x を y で置き換えたもの」つまり「y」になるでゲソ。

Haskell 風の式の場合には、評価は β 簡約の他に「四則演算の計算」も含むでゲソ。

```
n + m
```

のような式 (n, m は整数) を「四則演算を計算した結果」に置き換えるステップを 1 回の評価とみなすでゲソ。今日はこのような式も redex と呼ぶことにするでゲソ。式 1 で、

```
(\x -> x * x) (1 + 2)
```

の (1 + 2) という redex を選んで評価すると、

```
(\x -> x * x) 3
```

になるでゲソ。一方、(\x -> x * x) (1 + 2) という redex の評価を先にすると、x * x の中の x を (1 + 2) で置き換えて

```
(1 + 2) * (1 + 2)
```

になるでゲソ。このように、一つの式の中に redex が複数ある場合があるでゲソ。そのどれを選んで評価するかで評価順序が変わってくるでゲソ。

3.1.3 計算の終了 - 正規形 (Normal Form)

評価ステップを繰り返していって、redex がなくなるまで (それ以上できない状態の式になるまで) 到達したら、そこで計算終了 (停止) でゲソ。この redex がない式のことを **正規形 (Normal Form)** というでゲソ。

ただし、正規形に辿り着かず、無限に評価が終わらない場合もあるでゲソ。

[*4] ループとか再帰は今回は扱わないでゲソ。Y コンビネータを λ 式で表現して再帰を書く分には今回の話同様に評価すればいいでゲソが、それだと評価回数が多くなるでゲソ。明示的に Y コンビネータをプリミティブとして定義する場合とかは特別に考慮が必要でゲソ。これらの話については詳しくは [SPJ 1987] あたりを読めばいいんじゃなイカ?

[*5] λ 計算では四則演算や数値定数が無いのにどうやって計算するかって? チャーチ数とかを使えば変数・λ 抽象・関数適用の 3 種類だけで整数 (のようなもの) と整数演算は表現できるでゲソ (しかしこのことと今日の話はあまり関係ないでゲソ)。詳しくはチャーチ数で検索するか「簡約! λ カ娘」(1 冊目) を参照でゲソ。

3.2 評価順序を変えなイカ?


さて、一通り下準備が終わったところで最初に話した評価順序の話をもう一度しておくでゲソ。


すいません、遅れました!


なっ……こんなに早くここにたどり着くとは…あぶらあげの罠はどうしたのでゲソ?


? 何のお話ですか?


…なんでもないでゲソ。しかし今更来てももはや侵略済みでゲソ! まぁいいでゲソ、話を続けるでゲソよ! お前も Normal Order で侵略してやるでゲソ!

3.2.1 Applicative Order (作用的順序)


最初にちょっと出てきた **Applicative Order (作用的順序)** は、関数の引数をまず評価してから、関数適用を評価するでゲソ。**最右最内戦略**、つまり「一番右側の、一番内側の redex を選んで評価する」という言い方もするでゲソ[*6]。最初の方で出した図 3.1 では、式 1: (`\x -> x * x`) (`1 + 2`) を Applicative Order で評価しているでゲソが、(`1 + 2`) を 3 に評価してから関数適用を評価しているでゲソ。下線を引いた部分が各ステップで評価した redex を表しているでゲソ。


関数型、手続き型問わず広く見られる評価順序ですね。普及度からすればこちらの評価順序に「正規」という名前を与えても良かったのではないでしょうか?


そうはいかないでゲソ。他の評価順序では停止するのに Applicative Order で評価すると停止しないことがある、という欠点があるでゲソ。そんな評価順序には「正規」という名前はやれないでゲソ。たとえば式 2 のような場合でゲソ[*7]。

式 2: 評価順序によって停止したり停止しなかったりする例

```
(\y -> 1) ((\x -> x x) (\x -> x x))
```

これを Applicative Order で評価すると、図 3.3 のように無限ループするでゲソ。

この式の中の (`\x -> x x`) (`\x -> x x`) (λ 式だと $(\lambda x.xx)(\lambda x.xx)$) は、計算が停止しない有名な式のひとつでゲソ。Applicative Order で評価するとこの部分の評価を繰り返して無限ループになるでゲソ。

3.2.2 Normal Order (正規順序)


確かに、停止しないのは困りますね。何かいい評価順序はないものでしょうか…


そこで **Normal Order (正規順序)** でゲソ。これは、**最左最外 (leftmost outermost) 戦略** と

[*6] 構文木上での preorder 順で一番最後の redex という言い方もできるんじゃなイカ?

[*7] これは Haskell だと型が付かないでゲソが、Haskell でも再帰を使えば Applicative Order で評価すると停止しない式が書けるでゲソ。今回は再帰は扱わないのでこういう例にしたでゲソ

```
(\y -> 1) ((\x -> x x) (\x -> x x))
→ (\y -> 1) ((\x -> x x) (\x -> x x))
→ (\y -> 1) ((\x -> x x) (\x -> x x))
→ ...
```

図 3.3: 式 2: Applicative Order では停止しない

も呼ばれるもので、その名の通り「最も外側にある redex のうち最も左側にあるものを評価する」評価戦略でゲソ[*8]。式 1 の Normal Order での評価は最初の方の図 3.2 で出した通りでゲソ。`(\x -> x * x) (1 + 2)` の方が `(1 + 2)` よりも外側にある (前者が後者を包含している) ので、前者を先に評価するでゲソ。

この Normal Order は、「正規」の名にふさわしく、「ある式 e を正規形まで評価して停止する評価順序が存在するなら、Normal Order で評価すれば必ず停止する」という性質があるでゲソ。つまり、どうやっても計算が停止しないような式でない限りは、Normal Order をとりあえず使っておけば必ず計算が停止するのでめでたいでゲソ!

さっきの式 2 も、Normal Order なら図 3.4 のように停止するでゲソ:

```
(\y -> 1) ((\x -> x x) (\x -> x x))
→ 1
```

図 3.4: 式 2: Normal Order なら停止する

3.2.3 チャーチ・ロッサー

これで停止するので安心ですね。でも、評価順序によって停止する場合と停止しない場合があるのなら、評価順序によって別の答えが出たりしないんでしょうか?

評価順序によって停止するか停止しないかが違ってくることはあるでゲソが、停止する場合は全ての停止する評価順序は必ず同じ答えを出す (同じ正規形になって停止する) でゲソ! これはチャーチ・ロッサー (Church-Rosser) の定理というものでゲソ。停止しない場合は答えが出ないんだから気にする必要はないんじゃなイカ?

[*8] 構文木上での preorder 順で一番最初の redex という言い方もできるんじゃなイカ?

3.2.4 どの評価順序が評価回数が少ないでゲソ?


よかった、答えが変に変わる心配もしなくていいですね! でも、式 1: `(\x -> x * x) (1 + 2)` を Applicative Order で評価した場合は評価は 3 回で、Normal Order で評価した場合は 4 回だから、1 回増えちゃってます。……ひょっとして、Applicative Order で停止する式であれば Applicative Order で評価すれば、Normal Order より評価の回数は減ったりしないでしょうか?


残念ながら世の中そんな甘くないでゲソ。確かに式 1 の場合だと Applicative Order の方が評価回数が少ないでゲソ。一方で、Normal Order の方が評価回数が少ない場合もあるでゲソ。

式 3: Normal Order の方が Applicative Order よりも評価回数が少ない例

```
(\y -> 1) (2 + 3)
```


式 3 は、「引数 `y` を受け取って `1` を返す関数」を `(2 + 3)` に適用しているでゲソ。この引数 `x` の値は結局使われないので評価するだけ無駄でゲソが、Applicative Order だと図 3.5 のように、律儀に評価してしまうでゲソ (式 2 も無限回 vs 1 回で多くかかっているとも言えるでゲソが、式 3 のように両方停止する場合でも Normal Order の方が評価回数が少ないことも実際あるでゲソ)。

```
(\y -> 1) (2 + 3)
→ (\y -> 1) 5
→ 1
```

図 3.5: 式 3: Applicative Order だと 2 回


Normal Order なら図 3.6 のように一発でゲソ! 関数 `(\y -> 1)` の中身 `1` を取ってきて、その中にある仮引数 `y` を実引数 `(1 + 2)` に置き換えるでゲソが…`y` なんて一度も出てこないので、実引数は完全消滅して、`1` が得られるでゲソ!

```
(\y -> 1) (2 + 3)
→ 1
```

図 3.6: 式 3: Normal Order だと 1 回


このように、「Applicative Order での評価が、最終的には要らなくなる引数を無駄に評価していた」場合は Normal Order の方が評価回数が少なく、一方で「Applicative Order では 1 回で済んでいた引数の評価が、Normal Order では複数個にコピーされて別々に評価された」場合は Applicative Order の方が評価回数が少なくなるでゲソ。更に、Applicative Order でも Normal Order でもない評価順序の方が評価回数が少ない場合もあるでゲソ。たとえば

式 4: Normal Order でも Applicative Order でもない評価順序が回数最小である例

```
((\g -> (g (g (\x -> x)))) (\h -> ((\f -> (f (f (\z -> z)))) (h (\y -> y)))))
```

のような場合でゲソ[*9]。ちょっと長いので、

```
X = (\x -> x)
Y = (\y -> y)
Z = (\z -> z)
```

のように略記すると

式 4: 略記版

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
```

になるでゲソ。この式を Normal Order で評価すると、図 3.7 のようにコピーがたくさん発生して正規形までに 19 回かかるでゲソ。

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\h -> ((\f -> (f (f Z))) (h Y))) X))
→ ((\f -> (f (f Z))) (((\h -> ((\f -> (f (f Z))) (h Y))) X) Y))
→ ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((\f -> (f (f Z))) (X Y)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((X Y) ((X Y) Z)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (((Y ((X Y) Z)) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((X Y) Z) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (((Y Z) Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((Z Y) ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ (Y ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z))
→ ((((\h -> ((\f -> (f (f Z))) (h Y))) X) Y) Z)
→ ((((\f -> (f (f Z))) (X Y)) Y) Z)
→ ((((X Y) ((X Y) Z)) Y) Z)
→ (((Y ((X Y) Z)) Y) Z)
→ ((((X Y) Z) Y) Z)
→ (((Y Z) Y) Z)
→ ((Z Y) Z)
→ (Y Z)
→ Z
```

図 3.7: 式 4: Normal Order だと 19 回

Applicative Order だと図 3.8 のように 12 回で済むんじゃなイカ?

[*9] 式 4 は [Lamping 1990] に出てきた例をちょっと変えたものでゲソ

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\g -> (g (g X))) (\h -> ((h Y) ((h Y) Z))))
→ ((\h -> ((h Y) ((h Y) Z))) ((\h -> ((h Y) ((h Y) Z))) X))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) ((X Y) Z)))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) (Y Z)))
→ ((\h -> ((h Y) ((h Y) Z))) ((X Y) Z))
→ ((\h -> ((h Y) ((h Y) Z))) (Y Z))
→ ((\h -> ((h Y) ((h Y) Z))) Z)
→ ((Z Y) ((Z Y) Z))
→ ((Z Y) (Y Z))
→ ((Z Y) Z)
→ (Y Z)
→ Z
```

図 3.8: 式 4: Applicative Order だと 12 回

更に、最小の評価順序は図 3.9 のようなものでゲソ。これは Normal Order、Applicative Order のどちらでもないでゲソ。`(X Y)` の評価を更に 1 回減らして 11 回で評価できるでゲソ。

```
((\g -> (g (g X))) (\h -> ((\f -> (f (f Z))) (h Y))))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\h -> ((\f -> (f (f Z))) (h Y))) X))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\f -> (f (f Z))) (X Y)))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) ((\f -> (f (f Z))) Y))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) (Y (Y Z)))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) (Y Z))
→ ((\h -> ((\f -> (f (f Z))) (h Y))) Z)
→ ((\f -> (f (f Z))) (Z Y))
→ ((\f -> (f (f Z))) Y)
→ (Y (Y Z))
→ (Y Z)
→ Z
```

図 3.9: 式 4: 最小回数の評価順序はそのどちらでもなく 11 回


じゃあ、最終的にいずれ必要になる引数の場合だけ Applicative Order で評価して、残りの要らないっぽい引数は Normal Order で遅延評価する、みたいなことをすれば良いのではないでしょうか?


そうでゲソね。ただ、「最終的にいずれ必要になる引数」の判定は**正格性解析**とも関連する話で、完全な解析は決定不能でゲソ。近似アルゴリズムがいくつかあるようでゲソ。興味のある人は [Sakai 2010] あたりを見るといいんじゃなイカ? この場合は、評価順序を決めるのに解析が必要でゲソが、コンパイラに実装するには良いんじゃなイカ?

あと、Normal Order だと評価回数が増えるだけでなく、必要なメモリ量が激しく増大する場合もあるでゲソ (たとえば [Sakai 2010] 冒頭にそういう例が載っているでゲソ) が、今回そういう話もしないでゲソ。


まぁ、今日はその話は置いといて、単純で安心な Normal Order で評価しつつ、評価回数を評価ステップの実装の工夫で減らす方法を紹介するでゲソ。Normal Order の場合問題はコピーの発生だったので、コピーを最小限に抑える方向性を追求するでゲソ。こちらの方向性も、評価回数の本当の最小化、まで目指すと β 簡約以外の処理時間が指数時間になるとかいうアルゴリズムがあるでゲソが、そこまでがんばらない方法は今も主に Haskell などの遅延評価をするプログラミング言語で広く使われているでゲソ。

まとめ

1. 一般に、評価順序は複数ある
2. 評価順序によって計算が停止するか停止しないかが変わる場合がある
3. が、計算が停止する場合は計算結果 (正規形) は一意に定まる
4. 正規形に到達する評価順序が存在すれば、Normal Order での評価も必ず正規形に到達して停止する
5. 評価順序によって評価の回数は異なることがあり、Normal Order の方が良い場合も Applicative Order の方が良い場合もそれ以外の順序が良い場合もある

3.3 グラフ簡約しなイカ?


そういえばイカ娘さん、質問があるのですが。式 1 の Normal Order での評価の時にこんな風に

```
→ (1 + 2) * (1 + 2)
→ 3 * (1 + 2)
→ 3 * 3
```

評価していましたが、2 箇所ある `(1 + 2)` を別々に評価する必要はあるのでしょうか?


いい指摘でゲソ。文字列ベースで考えると、`(1 + 2)` は 2 回別々に出てくるので、2 回評価するのが筋な気がするでゲソ。

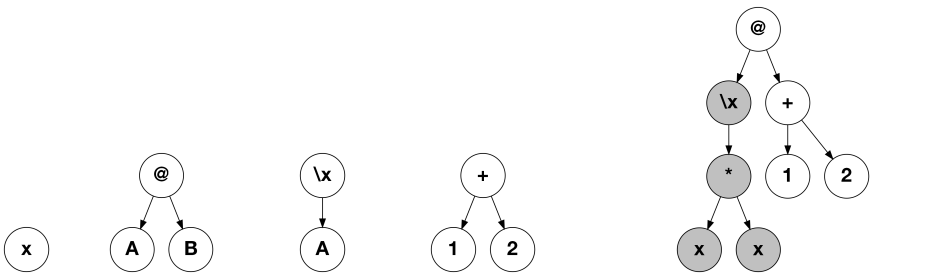
しかし、これだと Applicative Order の時よりも 1 回多く評価することになるでゲソ。この例だと評価回数の差は 1 回でゲソが、場合によってはもっとたくさん差がつくでゲソ。Normal Order だと評価回数が指数爆発することもあるでゲソ。

この 2 つの `(1 + 2)` は、もともと一つだったものが β 簡約の時に 2 つに増えたものでゲソ。こ

の「β 簡約の時に引数がコピーされる」というのが Normal Order の評価回数の増加原因でゲソ。どうにかしてコピーをせず、余計な評価を増やさずに評価できないものか…それができる方法の一つに、グラフ簡約という評価の実装方法があるでゲソ。引数をコピーせずに 1 つだけ持っておいて共有することで、コピーも余分な評価も防ぐでゲソ。これが編み出されたことで、引数のコピーを防いで評価回数を減らすことができたでゲソ。

3.3.1 式のグラフ表現

グラフ簡約を説明するために、少し準備するでゲソ。さっきまでは式を文字列で書いていたでゲソが、ここでは式をグラフで表現する方法を説明するでゲソ。グラフにすると文字列よりも色々操作しやすいでゲソ[*10]。表 3.1 の式はそれぞれ、図 3.10 (a)〜(d) のようにツリーで表現できるでゲソ:



(a) 変数 `x` (b) 関数適用 `(A B)` (c) 関数 `(\x -> A)` (d) 四則演算・数値定数 `1 + 2` (e) `((\x -> x * x) (1 + 2))`

図 3.10: 式のグラフ表現

変数 `x` は、図 3.10 (a) のように 1 つのノードに変数名を書いて表現するでゲソ。

関数適用 `(A B)` は、図 3.10 (b) のように、関数適用を表す `@` ノードを使って表すでゲソ。`@`ノードの左右の子が、A, B にそれぞれ対応しているでゲソ。

関数/λ 抽象 `(\x -> A)` は、図 3.10 (c) のように、λ 抽象を表す `\x` ノードを使って表すでゲソ (`x` の部分は変数名でゲソ)。`\x` ノードの子が関数本体 A に相当するでゲソ。

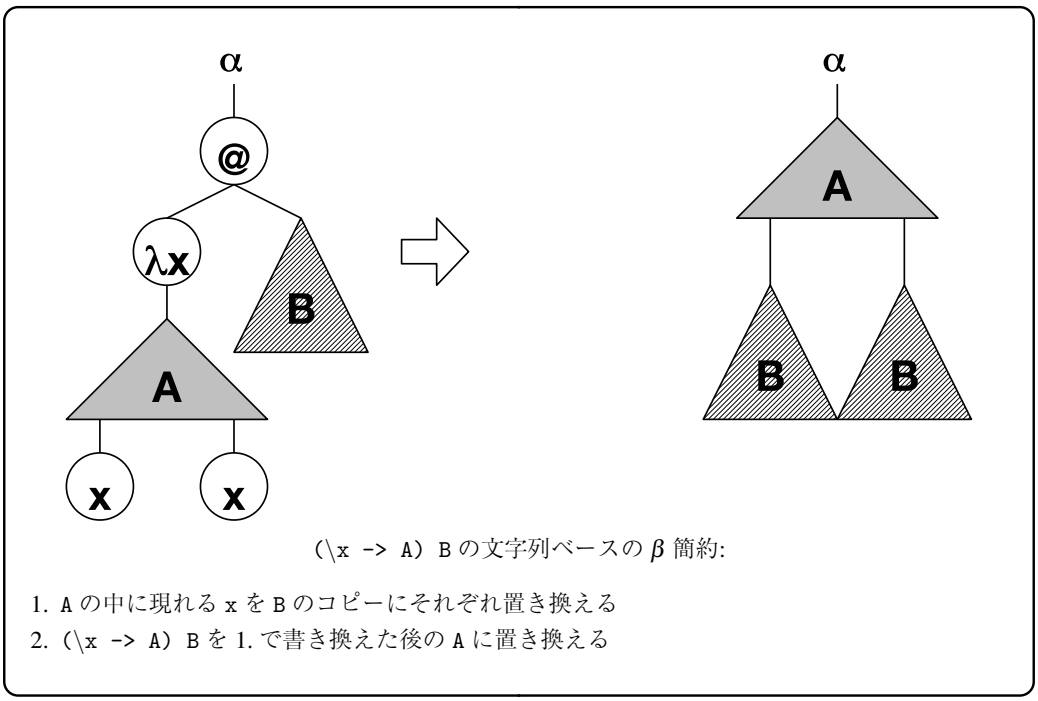
四則演算・数値定数 は、図 3.10 (d) のように、ノードに演算子 (この場合は `+`) や数値 (`1`, `2` など) を書いて表すでゲソ。演算子ノードの左右の子が被演算数でゲソ。この場合は `(1 + 2)` を表しているでゲソ。

図 3.10 (e) は、式 `((\x -> x * x) (1 + 2))` の抽象構文木表現でゲソ。`\x` とその子孫ノードからなるサブツリー (灰色の部分) が、部分式 `(\x -> x * x)` と対応しているでゲソ。

[*10] ここでは式は「抽象構文木」というツリーで表現できるでゲソが、後でツリーじゃなくて一般のグラフに拡張していくので、「グラフ表現」という言い方をしておくでゲソ。

3.3.2　文字列ベース簡約のグラフ表現

最初に説明した文字列ベースの β 簡約 (文字列ベース簡約) をグラフで表現すると図 3.11 みたいな感じになるでゲソ。これは $\alpha = (\lambda x.A)B$ の β 簡約を表しているでゲソ。図 3.11 の左側では、評価前の $\alpha = (\lambda x.A)B$ の状態を表しているでゲソ。左下の 2 つの x と書かれたノードが A の下にぶら下がっているのは、A の中に 2 回 x が出てくることを表現しているでゲソ。これを β 簡約すると右側のように、A の中の x があった場所に B をコピーしてつなげたものになるでゲソ。



`(\x -> A) B` の文字列ベースの β 簡約:

1. `A` の中に現れる `x` を `B` のコピーにそれぞれ置き換える
2. `(\x -> A) B` を 1. で書き換えた後の `A` に置き換える

図 3.11: 評価の実装方法その 1: 文字列ベース簡約

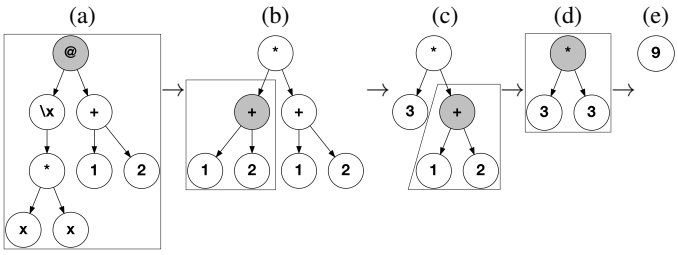


図 3.12: 式 1: Normal Order 評価、文字列ベース簡約

図 3.12 で、式 1: `(\x -> x * x) (1 + 2)` を文字列ベース簡約による Normal Order での評価をグラフで表現してみたでゲソ。これは 図 3.2 に対応しているでゲソ。図 3.12 の灰

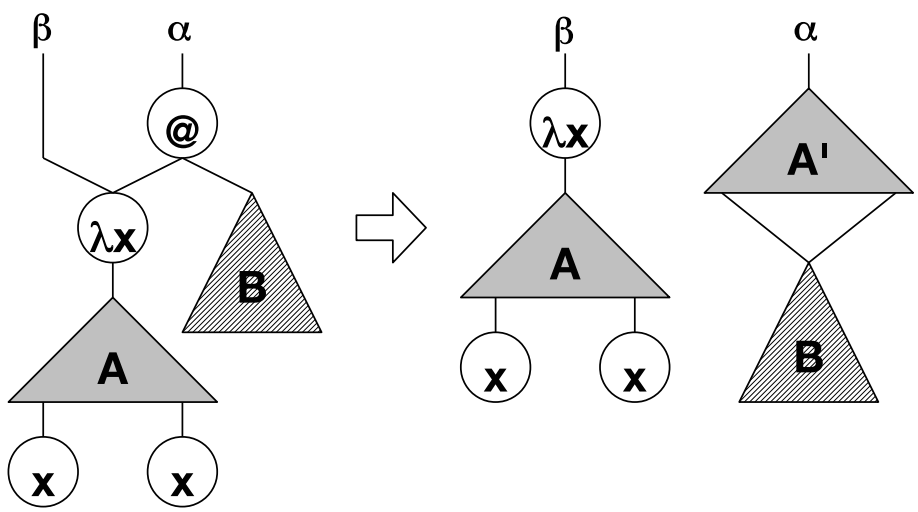
色のノードとその下にあるノード (枠で囲まれている部分) が、次に評価される redex を表していて、図 3.2 で下線が引かれた部分に対応しているでゲソ。

3.3.3 グラフ簡約

式 1 の Normal Order 評価で評価回数が増えたのは、1 回目の評価で (\x -> x * x) の関数 body の中に x が 2 回出てくるのにあわせて、(1 + 2) を 2 つにコピーしたからでゲソ。図 3.11 でも、「B」のサブツリーが 2 つにコピーされているでゲソ。コピーした以上は、後でコピーごとに別々に評価されると評価回数が無駄に増えるでゲソ。でも、(1 + 2) という式は参照透明なので何度評価しても変わらないし、コピーせずに 1 回だけ評価して済ませたいでゲソ。こんな感じで、1 回の評価で 2 回出てくる (1 + 2) を評価できればいいんじゃなイカ?

```
(\x -> x * x) (1 + 2)
→ (1 + 2) * (1 + 2)
→ 3 * 3
→ 9
```

それを実現する方法が **グラフ簡約 (Graph Reduction)** でゲソ。グラフ簡約では、関数適用の評価時に仮引数 x が関数 body の中に複数回出てきても、実引数をコピーせずに、実引数へのポインタをセットするなどして共有させることで、コピーを減らし、評価回数を減らしているでゲソ。元々の文字列ベースでの評価では式が構文木 (つまりツリー) になっていたでゲソが、グ



(\x -> A) B のグラフ簡約:

1. A をコピーして A' を作る
2. A' の中に現れる x を B へのポインタで置き換える
3. (\x -> A) B を 2. で書き換えた後の A' に置き換える

図 3.13: 評価の実装方法その 2: グラフ簡約

ラフ簡約では部分式が共有されるので式がグラフになるのでこの名前になっているでゲソ。グラフ表現で図にすると図 3.13 みたいな感じじゃなイカ?

ここで、$\alpha = (\lambda x.A)B$ を β 簡約する時に、B を 2 つコピーせずに、共有しているでゲソ。これで、B の評価は 1 回だけで済むでゲソ!

ちょっと待ってください。それはとても良いことなのですが、この図 3.13 を見ると今度は A がコピーされているように見えるのですが、これはなぜでしょうか?

それは、A が別の所 (図 3.13 の β) から共有されているかもしれないからでゲソ。なので、文字列ベース簡約の時のように直接 $(\lambda x.A)B$ の中の A を書き換えるとまずいでゲソ。そこで、A の部分をコピーして A' を作ってから、A' の中の x を B へのリンクに書き換えているでゲソ。文字列ベース簡約の場合はそもそも共有されるような場合がないからコピーせずに直接書き換えてよかったでゲソ。

図 3.14 に、式 1 のグラフ簡約での評価を示すでゲソ。ここでも灰色のノードとその下にあるノード (枠で囲まれている部分) が、次に評価される redex を表すでゲソ。以降の図でも灰色のノードが次に評価される redex を表すでゲソ (但し枠は別の意味で使っていたりするでゲソ)。図 3.12 (b) で 2 つにコピーされていた `(1 + 2)` の項が、図 3.14 (b) では共有されて 1 つになっているじゃなイカ! この共有された `(1 + 2)` が 図 3.14 (b) → (c) で 1 回で評価されて、全体として 3 回で評価が完了するでゲソ!

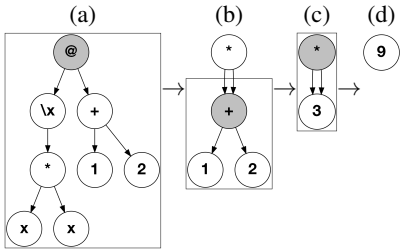


図 3.14: 式 1: Normal Order 評価、グラフ簡約

グラフ簡約は部分式が共有されるので文字で表現しにくいでゲソが、共有部分式に `let` で名前を付けて表現すると図 3.15 のようになるでゲソ。`let x = hoge in fuga` と書くと、`hoge` に `x` という名前を付けて `fuga` の中で使うでゲソ。グラフ簡約でのノードに名前を付けて共有関係を示すためにここでは使っているでゲソ。`let` に関する操作は β 簡約とかではないので「評価ステップ」の回数にはカウントしないことにするでゲソ。

```
(\x -> x * x) (1 + 2)
→ let x = (1 + 2) in x * x
→ let x = 3 in x * x
→ 9
```

図 3.15: 式 1: Normal Order 評価、グラフ簡約

グラフ簡約は、Normal Order だけじゃなく、他の評価順序の場合でも評価回数を減らすこともできるでゲソ。また、文字列ベース簡約では評価順序をどのようにしてもできなかった少ない評価回数を、グラフ簡約で実現することもできるでゲソ。たとえば、文字列ベース簡約では 11 回までしか評価回数を減らせなかった式 4 も、グラフ簡約を使えば図 3.16 のように 10 回まで減らせるでゲソ! 図 3.16 (d) で `(X Y)` を、図 3.16 (h) で `(Z Y)` を評価しているでゲソが、これらは 2 箇所から共有されているでゲソ。図 3.8 で文字列ベース簡約をした場合には、`(X Y)` と `(Z Y)` の簡約が 2 回ずつ必要だったでゲソが、グラフ簡約では共有されているので 1 回ずつで済むので、全体で評価回数が 2 回減っているでゲソ。

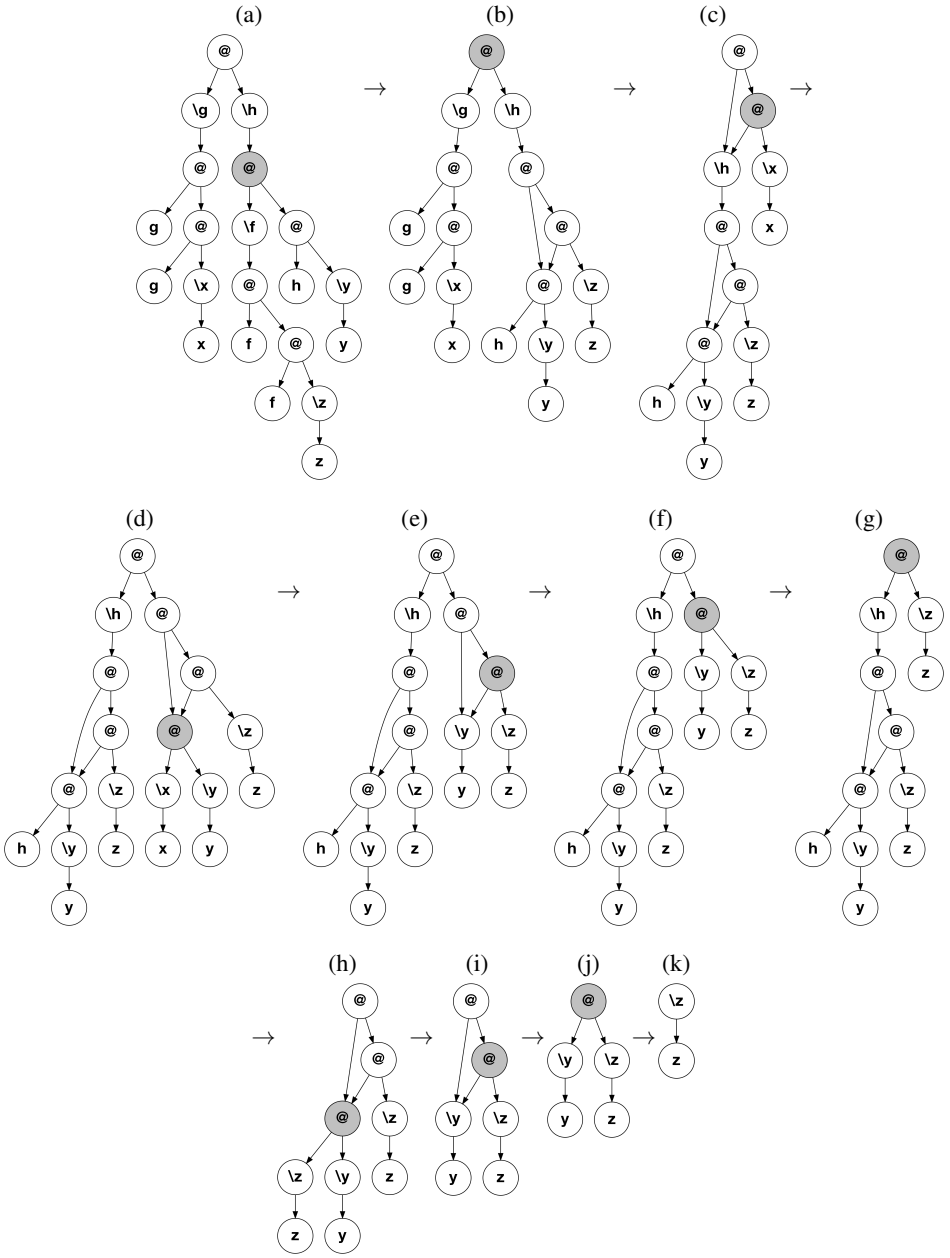


図 3.16: 式 4: Applicative Order 評価、グラフ簡約

3.4 Fully Lazy グラフ簡約

でもやっぱりこれだと A のコピーが増えてませんか?

……その通りでゲソ。A が丸ごとコピーされるので、その中にある関数適用は無駄に複数回評価されることがあるでゲソ。たとえば式 5 のような場合でゲソ。これをグラフ簡約を使って Normal Order で評価すると、図 3.17 (文字列表現)、図 3.18 (グラフ表現) のようになるでゲソ。

式 5: グラフ簡約 (Normal Order) でも無駄な評価が出る例

```
(\f -> (f 1) + (f 2)) (\x -> x + (\y -> y * 5) 6)
```

```
((\f -> ((f 1)+(f 2))) (\x -> (x+((\y -> (y*5)) 6))))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((f 1)+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+((\y -> (y*5)) 6))+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+(6*5))+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((1+30)+(f 2))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in (31+(f 2))
→ (31+(2+((\y -> (y*5)) 6)))
→ (31+(2+(6*5)))
→ (31+(2+30))
→ (31+32)
→ 63
```

図 3.17: 式 5: Normal Order 評価、グラフ簡約: 10 回

グラフ簡約でも 10 回かかるじゃなイカ! (\x -> (x+((\y -> (y*5)) 6))) の 1 と 2 への適用を評価する時に、(x+((\y -> (y*5)) 6))) がコピーされるでゲソ。図 3.18 (b) → (c) で、枠で囲まれた (x+((\y -> (y*5)) 6))) が 2 つに増えているのが分かるんじゃなイカ? それが後で x=1 の時 (図 3.18 (c) → (d) → (e)) と x=2 の時 (図 3.18 (g) → (h) → (i)) とで別々に評価されるでゲソ。このため、((\y -> (y*5)) 6) と (6*5) の評価がそれぞれ 2 回ずつ行われているでゲソ。

でも、((\y -> (y*5)) 6) の部分は x を含まないので、別々の x の値に対して適用しても結果は変わらないでゲソ。なので、この部分はコピーせずに共有して、全体で一回評価すればいいはずでゲソ。

一般的には、(\x -> A) B の評価をする時に、A を全部コピーするんじゃなくて、A のうち、「x に関係しない部分」はコピーせずに共有すればいいでゲソ。x と関係なければ、A が別の所で共有されていて、別の x の値に適用されても値は変わらないから問題ないんじゃなイカ! こういう方法を **Fully Lazy グラフ簡約 (Fully Lazy Graph Reduction)** と呼ぶでゲソ。

グラフで図示すると図 3.19 のようになるでゲソ: *A* の中の黒い部分 (*A* の中の「x に関係しない

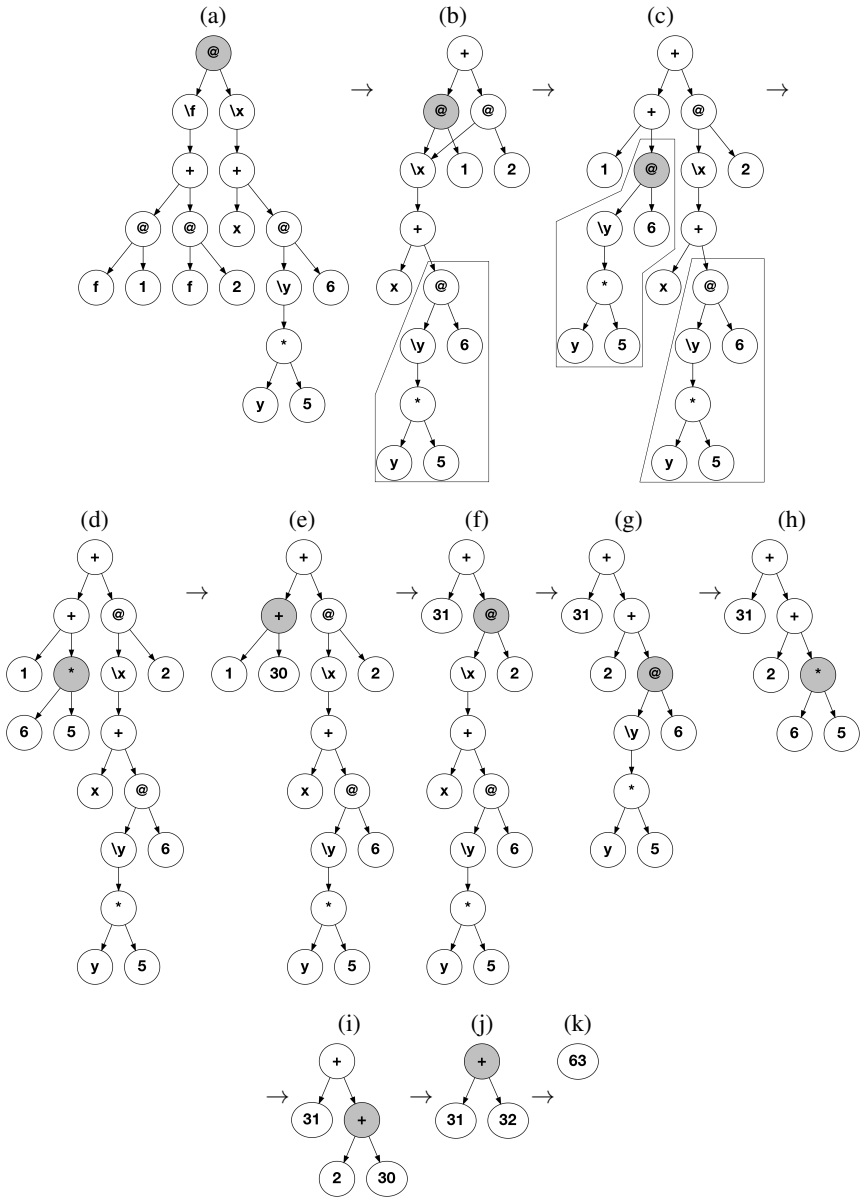
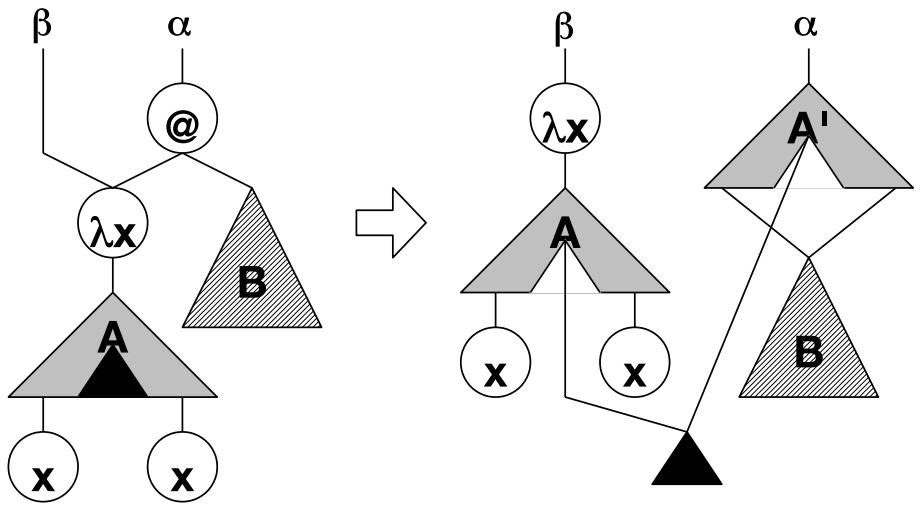


図 3.18: 式 5: Normal Order 評価、グラフ簡約: 10 回

部分」を表しているでゲソ) はコピーせずに共有して、*A* のそれ以外の部分だけコピーしているでゲソ。

具体的に「x に関係しない部分」が何かについてはいくつか手法あるでゲソが、一番簡単なのは Maximal Free Expression を使う方法でゲソ。**Free Expression** とは、自由変数を含まない式のことでゲソ。**Maximal Free Expression** とは、Free Expression のうち maximal なもの、つまり他の Free Expression の一部に含まれてはいないもののことでゲソ。例えば、`(x+((\y ->`



(\x -> A) B の Fully Lazy グラフ簡約:

1. A をコピーして A' を作る。但し、x に関係ない部分 (図中黒三角) はコピーせずに、A と A' とで共有する。
2. A' の中に現れる x を B へのポインタで置き換える
3. (\x -> A) B を 2. で書き換えた後の A' に置き換える

図 3.19: 評価の実装方法その 3: Fully Lazy グラフ簡約

```
((\f -> ((f 1)+(f 2))) (\x -> (x+((\y -> (y*5)) 6))))
→ let f = (\x -> (x+((\y -> (y*5)) 6))) in ((f 1)+(f 2))
= let z = ((\y -> (y*5)) 6) in let f = (\x -> (x+z)) in ((f 1)+(f 2))
→ let z = ((\y -> (y*5)) 6) in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = (6*5) in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = 30 in let f = (\x -> (x+z)) in ((1+z)+(f 2))
→ let z = 30 in let f = (\x -> (x+z)) in (31+(f 2))
→ let z = 30 in (31+(2+z))
→ (31+32)
→ 63
```

図 3.20: 式 5: Normal Order 評価、Fully Lazy グラフ簡約: 8 回

(y*5)) 6)) の場合、Free Expression は 5, 6, (\y -> (y*5)), ((\y -> (y*5)) 6) でゲソ。この中で Maximal Free Expression は ((\y -> (y*5)) 6) でゲソ。

(\x -> A) B の A の中にある Maximal Free Expression は自由変数を含まないので文脈とは独立に評価して値が決定できるのでゲソ。特に、x も含まないのでゲソ。なので、評価する時にもコ

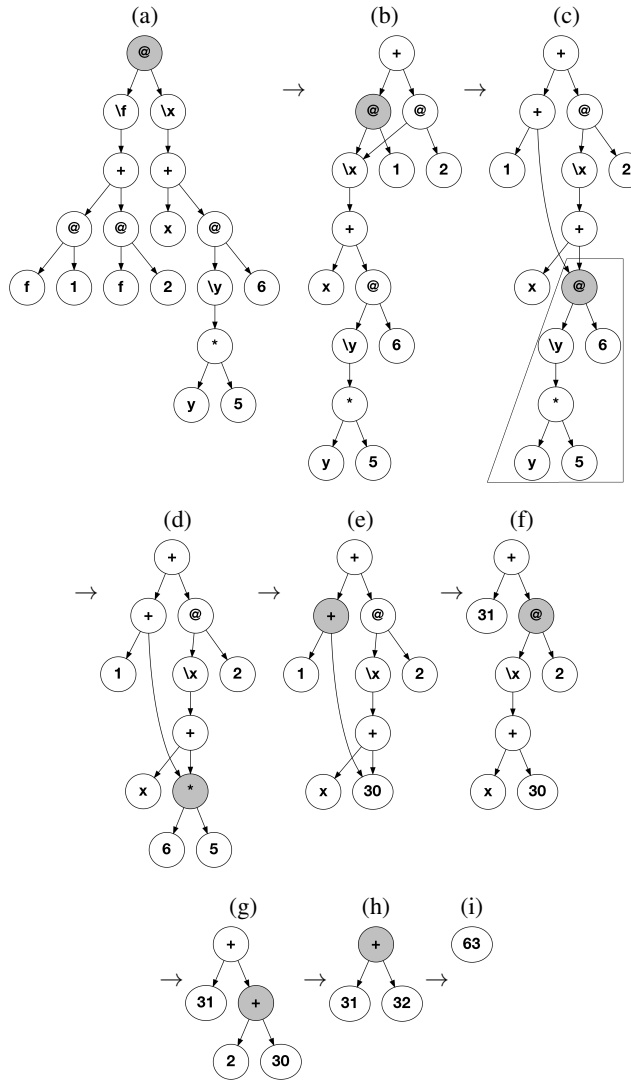


図 3.21: 式 5: Normal Order 評価、Fully Lazy グラフ簡約: 8 回

ピーする必要がないのでゲソ! Maximal とついているのは、単に効果を最大化するためでゲソ。

Maximum Free Expression を使って Fully Lazy グラフ簡約 (Normal Order) すると、式 5 の場合は図 3.20 (文字列表現)、図 3.21 (グラフ表現) のようになるでゲソ。図 3.20 の 3 行目で、`f` を `1` に適用する前に、Maximum Free Expression である `((\y -> y * 5) 6)` を `z` として括りだして、4 行目でそれ以外の部分 `(\x -> (x+z))` はコピーして関数適用を評価しているでゲソ (括りだされた `z` はコピーしないでゲソ)。グラフで見ると、図 3.18 (c) では 2 つにコピーされている `((\y -> y * 5) 6)` (枠で囲まれた部分) が、図 3.21 (c) では共有されているのが分かるんじゃなイカ?

3.5 Optimal Reduction

ここまでくれば完璧、でしょうか?

実はまだまだ最適じゃないでゲソ! 最適への道は長いでゲソ……。イカのような式 6 を考えてみるでゲソ。

式 6

```
((\f -> ((f 1) + (f 2))) (\x -> ((\y -> (x + (y * 5))) 6)))
```

前節で示した Fully Lazy グラフ簡約だと、Normal Order の場合図 3.22 のように 10 回評価が必要でゲソ。

```
((\f -> ((f 1)+(f 2))) (\x -> ((\y -> (x+(y*5))) 6)))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((f 1)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in (((\y -> (1+(y*5))) 6)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((1+(6*5))+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in ((1+30)+(f 2))
→ let f = (\x -> ((\y -> (x+(y*5))) 6)) in (31+(f 2))
→ (31+((\y -> (2+(y*5))) 6))
→ (31+(2+(6*5)))
→ (31+(2+30))
→ (31+32)
→ 63
```

図 3.22: 式 6: Normal Order 評価、Fully Lazy グラフ簡約: 10 回

これは `((\y -> (x+(y*5))) 6)` が 2 個にコピーされて、`6` への適用と `y * 5` の掛け算がそれぞれ 2 回ずつ行われたから、その分回数が増えているでゲソ。`y * 5` は結局 `6 * 5` に評価されるので、`x` の値に依存しないでゲソが、`y * 5` は Free Expression ではない (`y` が自由変数) ので共有できないでゲソ。`((\y -> (x+(y*5))) 6)` の中にある Free Expression は `5` と `6` だけでゲソ。

これを解決する手法もあるにはあるでゲソ。究極的には、`(\x -> A) B` を評価する時に、「自由変数 `x` を含まない部分式」はコピーを省ける、かもしれないでゲソ (だって `x` を含まないのでゲソからね)。前節の Fully Lazy グラフ簡約では、「自由変数を全く含まない部分式」という、それより強い条件を課してコピーを省いていたので、こういう無駄が出てきたでゲソ。ただ、「自由変数 `x` を含まない部分式」だけだと他の自由変数を含むことがあったり諸々の事情で、アルゴリズムは一気に複雑化するでゲソ…。

そのうちのひとつ、[Lamping 1990] の手法の雰囲気を紹介するでゲソ[*11]。これは、β 簡約の回数を最小化するという意味で **Optimal Reduction** と呼ばれる手法の一つでゲソ。雰囲気としては、図 3.23 のような感じでゲソ。A の上に▽の形をしたノードと、A の下に△の形をしたノードがあって、

[*11] 難しすぎて理解できなかったので雰囲気だけでゲソ

∗ と○の記号がついているでゲソ。これは、「上から辿った時に、▽の ∗ から入った場合 (つまり β から辿った場合) には、△から下に辿るときも ∗ の方 (x) に行く。逆に上から▽の○印の方から入った場合 (α からたどった場合) は、△から下に辿るときも○の方 (B) に行く。」という意味でゲソ。このようにすることによって、A の構造を共有したまま、x の部分だけが違う複数の文脈を表現できるでゲソ。

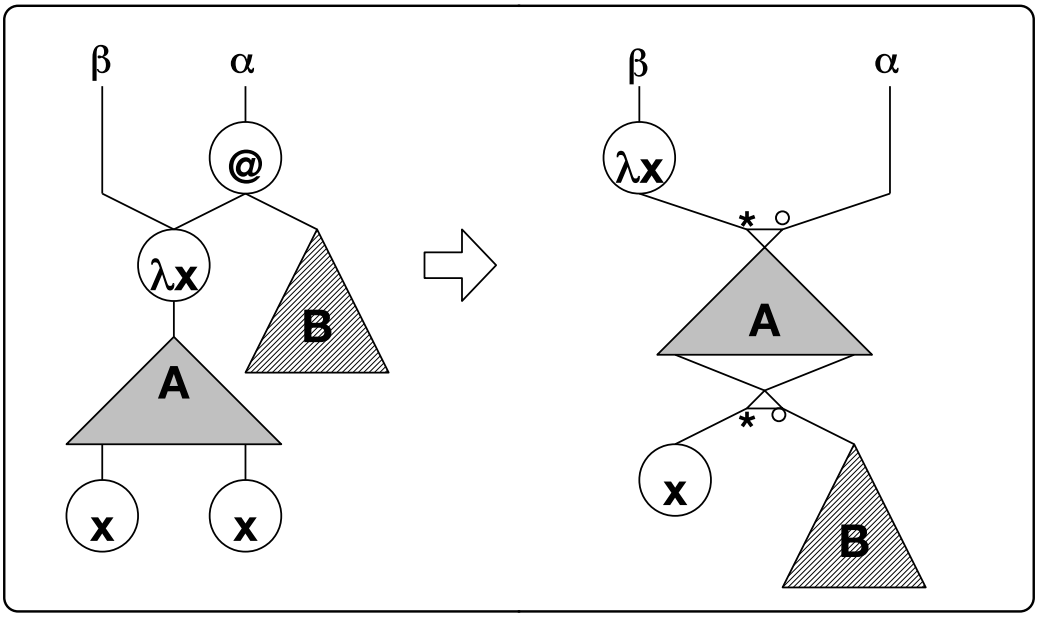


図 3.23: 評価の実装方法その 4: Optimal Reduction

図 3.24 に、[Lamping 1990] による式 6 の評価の雰囲気を示すでゲソ。Normal Order じゃなくなっているでゲソが、雰囲気だけ感じて欲しいでゲソ。

(a) から始まって、(b) までは普通でゲソ。

(b) で (\x -> (\y -> x + y * 5) 6) を 1 と 2 に適用したものを評価して (評価 2 回分) (c) のようにするでゲソ。ここで、(\y -> x + y * 5) 6) はほとんどがコピーされずに残っていて、(\y -> [q] + y * 5) 6 のようになっているでゲソ。[q] には 1 か 2 が入るでゲソが、そのどちらが入るかは、上からたどって [p] に入る時、[p] に向かう 2 本の枝 (∗と o のラベルが付いた枝で、それぞれ (\x -> (\y -> x + y * 5) 6) を 1 と 2 に適用した場合に対応するでゲソ) のどちらが使われるかに連動して決まるでゲソ。[p] に入る時に∗の枝を使えば [q] の値も∗の枝を、[p] に入る時に o の枝を使えば [q] の値も o の枝を使う感じでゲソ。このような苦肉の策によって、(\y -> x + y * 5) 6 の構造のほとんどを共有したまま、評価を進めることができるのでゲソ。

(c) → (d) → (e) で、y * 5 あたりを重複なしに評価した後、(e) → (f) では (\x -> (\y -> x + y * 5) 6) を 1 に適用した時相当の評価を∗の枝を使ってしているでゲソ。一番上の + ノードの左側の子ノードを評価しているでゲソが、[p] に∗の枝から入っているので、[q] の値はそれと連動して 1 になるでゲソ (太い矢印で示したツリーが使われるでゲソ)。同様に、(f) → (g) では (\x -> (\y -> x + y * 5) 6) を 2 に適用した時相当の評価を o の枝を使ってしているでゲソ。このようにして、8 回の評価で評価が完了したでゲソ!! ここでは説明の都合上 Normal Order じゃない評価を示したでゲソが、本式の Optimal Reduction では Normal Order 評価でも 8 回で済むはずでゲソ。

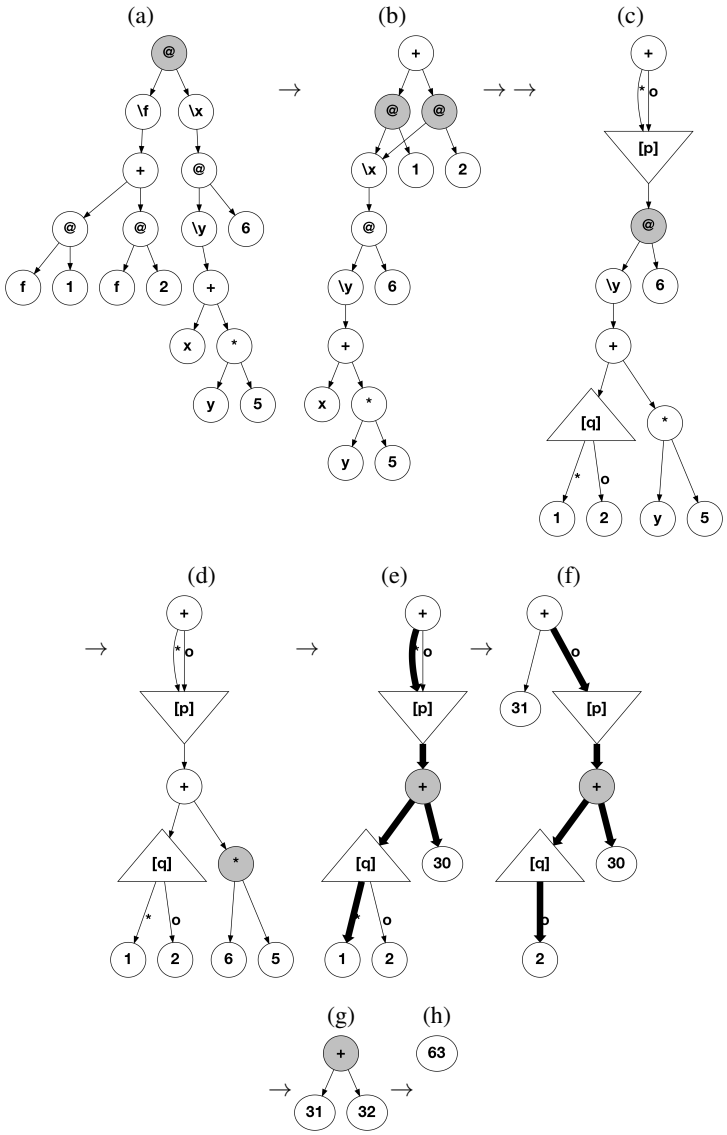


図 3.24: 式 6: Optimal Reduction

この [Lamping 1990] の手法は、元論文とか見るともっと多くの種類のノードとかエッジとかがあって、グラフ簡約ルールが 30 個以上もある複雑なものでゲソ。どうやら、これは β 簡約回数を最小化するらしいでゲソが、β 簡約以外にすることが増えて (▽ノードとかの管理でゲソね)、評価 1 回あたりの計算量が指数時間かかる、というヤバイ状況らしいでゲソ。後に他の手法も考えられたようでゲソが、どのみち私には難しすぎてよく分からなかったでゲソ。追求したい人はなんかそれだけで一冊本 [Asperti+ 1998] が出ているようなのでそれ読めばいいんじゃなイカ?[*12]

[*12] 私は読んでないでゲソ…

今日は、評価の回数を「評価順序の変更」と「評価方法の改良」の二つの点から少なくする話をしたでゲソ。割と「プログラムをそのまま」「Normal Order で」評価することにこだわっていたので、プログラム解析をしたりプログラムを変換してから実行するような話は扱わなかったでゲソ。
そういう、今回扱わなかった話の一つ目は、コンパイラ最適化でゲソ。たとえば、今日の話は `(1 + 2)` がコピーされてどんどん増えていくのを防いで評価回数を減らそう、という話でしたでゲソが、最初からイカのような式が書かれていた時にはどうにもならないでゲソ。

式 7

```
((1 + 2) * (1 + 2))
```

この場合、二つの `(1 + 2)` はコピーで増えたわけではないので、今日の話の範囲ではこのまま別々に評価することになるでゲソ。

でも、解析すれば同じ式であることを検知して一つにまとめることはできそうですよね?

もちろん、できるでゲソ。それは、評価方法の改良というよりは、コンパイラ最適化の一つである**共通部分式除去 (Common Subexpression Elimination (CSE))**でゲソ。他にもコンパイラの最適化手法はたくさんあるので、色々早くすることができるはずでゲソ。
扱わなかった二つ目は、(Fully Lazy) Lambda Lifting 関連の話でゲソ。今回は、Fully Lazy グラフ簡約のあたりで Free Expression を評価の時に括りだしていたでゲソが、そういうのを Lambda Lifting というプログラム変換を使って前もって括りだしておく方法もあるでゲソ。これもコンパイラ実装向きな話じゃなイカ?

3.6 まとめと参考文献

結局、今日の話をまとめるとどんな感じになるんでしょうか?

迷ったらとりあえず Normal Order でグラフ簡約すればいいんじゃなイカ? 評価順序も、評価の実装方法も色々工夫できるので面白いでゲソが、突き詰めすぎると底なし沼でゲソ!

参考文献

SPJ 1987 S. Peyton Jones. The Implementation of Functional Programming Languages. Prentice-Hall, Inc., 1987.
`http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/`
評価順序の話から基本的なグラフ簡約の話などだいたいカバーしているでゲソ! 今回は省いた実装や WHNF、再帰関連の話なども豊富でゲソ。

Sumii 2005 住井英二郎. ラムダ計算入門 (2005 年度「計算機ソフトウェア工学」授業資料)
`http://www.kb.ecei.tohoku.ac.jp/~sumii/class/keisanki-software-kougaku-2005/lambda.pdf`
λ計算・β簡約などの基礎の話がまとまっているでゲソ。

shelarcy 2007 本物のプログラマは Haskell を使う 第 8 回 遅延評価の仕組み
`http://itpro.nikkeibp.co.jp/article/COLUMN/20070305/263828/`.
遅延評価の基本から、Haskell における遅延評価の実際、弱頭部正規形 (Weak Head Normal Form (WHNF))、豊富な参考リンクなど。

Sakai 2010 酒井 政裕. Haskell Advent Calendar 2010 22 日目: 正格性解析 (Strictness Analysis)
`http://msakai.jp/d/?date=20101228`.
正格性解析を用いて評価順序を変える話でゲソ。

Lamping 1990 J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, pages 16–30, 1990.
Optimal Reduction のアルゴリズムの一つでゲソ。ファンシーな部品がたくさんあるグラフの図が載っているでゲソ。

Levy Jean-Jacques Le'vy, Reductions and Causality (講義スライド)
`http://pauillac.inria.fr/~levy/courses/tsinghua/reductions/`
Optimal Reduction 理解の参考にしたでゲソ。

Asperti 2009 Andrea Asperti, "The optimal reduction of lambda expressions" (スライド)
`http://www.cs.unibo.it/~asperti/SLIDES/optimal.pdf`
Optimal Reduction のヤバさを感じるスライドでゲソ。

Asperti+ 1998 A. Asperti, S. Guerrini, "The Optimal Implementation of Functional Programming Languages", Cambridge Tracts in Theoretical COmputer Science n.45, Cambridge University Press, 1998.
Optimal Reduction 関係で本一冊書けるらしいでゲソ。

第4章

オール・アバウト・ケーゾク・イン・スキーム

— @yryozo

お主たち！ 今日も元気に括弧をつけてるでゲソか?

括弧の乱れは心の乱れ、心の乱れは家庭の乱れ、(略)、そして国の乱れは宇宙の乱れ、とどこかの偉い人も言ってたような気がするでゲソ。これからも精進してカッコいいプログラマーを目指すでゲソよ。

さて、前回は Scheme にとって大事な「末尾呼び出しの最適化 (Tail Call Optimization, TCO)」の話をしたでゲソ [*1]。でも Scheme にとって大事なものはもう一つ [*2]あるでゲソ。そう「継続 (Continuation)」でゲソね。というわけで、今回は「継続」の話をしておこうと思うんでゲソ。[*3]

4.1 「継続」について勉強しなイカ?

まず最初に「そもそも継続ってなんなのか」を簡単に説明しておくでゲソ。

といっても、こんな本を読んでいるくらいだから「継続」っていう言葉は既に聞いたことがあるんじゃなイカ? そんなよく訓練されたお主たちのために超スピードで説明してみるでゲソ!

(1) 継続って何かというと「その後に行われる計算処理全体」のことでゲソ。
(2) さらに、一部の言語ではこの「継続」を実行時に捕捉して使うことができるんでゲソ (一部の言語っていうのは、例えば Scheme とか Scheme とか Scheme とかでゲソ。後は Ruby とか Standard ML とかもそうでゲソ)。
(3) で、そういった言語では「捕捉した継続」を好きなときに「起動」できるんでゲソ。継続を起動すると、捕捉されていた「その後に行われる計算処理」が実行されるんでゲソ。

... うむ、正直、何がなにやらよく分からないでゲソね (汗。なんというか、「わけがわからないよ」とか「この概念は出来損ないだ。使えないよ」という雰囲気がプンプンするでゲソ。

じゃあ、これをちょっと別の言い方で説明してみるでゲソ。

(1) 継続って何かというと、「その時点でのスレッドの状態」のことでゲソ。
(2) 一部の言語には、「スレッドの状態のコピーを取る」機能が用意されているんでゲソ。(そしてこの「コピーを取る処理」のことを「継続の捕捉」と呼ぶんでゲソ)

[*1] といっても、前回の話との繋がりは一切ないので今回から読んでもらって全く問題ないでゲソ。
[*2] もちろん正確にはもっと一杯あるでゲソ。なにしろ Scheme は全てが大事と言っても過言ではないでゲソ。
[*3] ところで、タイトルはちょっとニンジャ○レイヤーっぽいでゲソが、中身はまるっきり関係ないでゲソ。タイトルだけの完全な出落ちでゲソね。まぁ、バー○ンハウスにでも引っかかったと思って許して欲しいでゲソ。

(3) で、取得したコピーを使うと、スレッドの状態を「コピーを取ったときの状態」に変更することができるんでゲソ。(そしてこの「状態を変更する処理」のことを「継続の起動」と呼ぶんでゲソ)

… え、さっきと言ってることが違い過ぎなイカって? うーん、確かに一見すると全然違う話のように見えるでゲソね。だけど言ってることはほとんど同じなんでゲソ。

さっきの話だと、継続というのは「今後行う計算処理」だったでゲソ。だから継続を捕捉するというのは「今後行う計算処理を取得すること」で、継続を起動するのは「取得した残りの計算処理を行うこと」ということでゲソ?

ところで、この「今後行う計算の取得」と「その実行」って、OS が日常的に行ってる「ある処理」に似てなイカ?

そう、スレッドのコンテキストスイッチでゲソ! スレッドを「いったん寝かせて処理を停止」しておいて、別のタイミングで「起こして計算を再開」させるというのは、まさに「今後行う計算を取得」しておいて「その処理を実行」することでゲソ。実際問題、計算処理を実行しているスレッドの状態を完全に記録することができれば、そのスレッドがこれから行おうとしていた処理を完全に再現することは可能でゲソ? こう考えれば、「継続」=「その後に行われる計算処理」=「スレッドの状態」というのも自然じゃなイカ[*4]。

さて、これでなんとなく分かってきたかもしれないでゲソが、「まだ具体的なイメージが湧かないよ」という人もいると思うので、もうちょっとだけ具体的に説明しておくでゲソ。

「スレッドの状態」というのは (最近の大抵の言語処理系では)「スタックの中身」のことでゲソ? [*5]例えば、今プログラムのどこを実行しているか (いわゆるプログラムカウンタの値でゲソね) とか、リターンが起こった時にどこに制御を戻すか (いわゆるリターンアドレスでゲソね) とか、今後の実行を続けるために必要な情報は全部スタックの中に入っているじゃなイカ?

だからぶっちゃけてしまうと、(大抵のケースでは) 次のように考えても問題ないでゲソ[*6]。

(1) 継続って何かというと、「スタックの中身」のことでゲソ。
(2) 一部の言語には、「スタックの中身のコピーを取る」機能が用意されているんでゲソ。(そしてこの「コピーを取る処理」のことを「継続の捕捉」と呼ぶんでゲソ)
(3) で、取得したコピーを使うと、スタックの内容を「コピーを取ったときの中身」に変更することができるんでゲソ。(そしてこの「スタックを変更する処理」のことを「継続の起動」と呼ぶんでゲソ)

次のページに図も描いてみたのでそれも参考にして欲しいでゲソ。

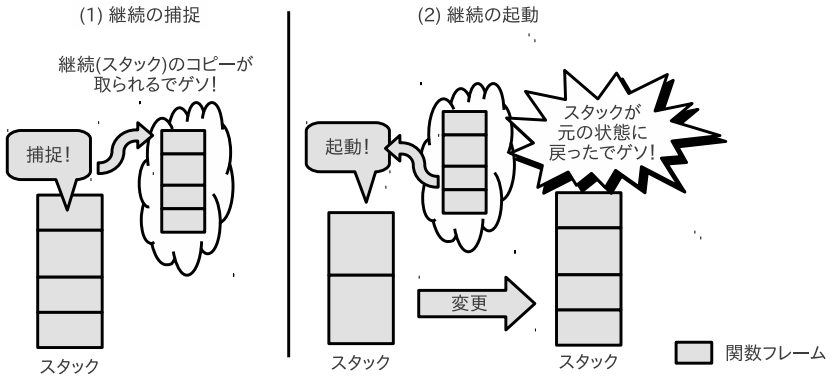
で、この継続でゲソが、面白いのはやっぱり (2)(3) の捕捉処理&起動処理を用意してある言語の場合でゲソ。(もちろん捕捉処理がない言語でも (1) はあるわけでゲソが、これだけだと「だから何だ」という感じじゃなイカ? [*7])

[*4] まだ納得いかないという人のためにもうちょっと説明すると、昔の論文の中には本当に継続相当の代物を “state” と呼んでるものもあったんでゲソ [10] [11]。ちなみに、この論文を書いた Peter Landin という人は、ML や Haskell の祖先である ISWIM という言語を作ったり、1965 年の時点で IO monad に近いものを提案してたりと [9]、黎明期の関数型言語の立役者のような人でゲソ。

[*5] ところで細かい話でゲソが、レジスタがスタックに待避されていない状態を想定するなら「スタックの中身 + レジスタの中身」ということになるでゲソ。でももう面倒だから、継続を捕捉する処理が実行される際には全てのレジスタはスタックに待避されてる、ということにして話を進めるでゲソ。

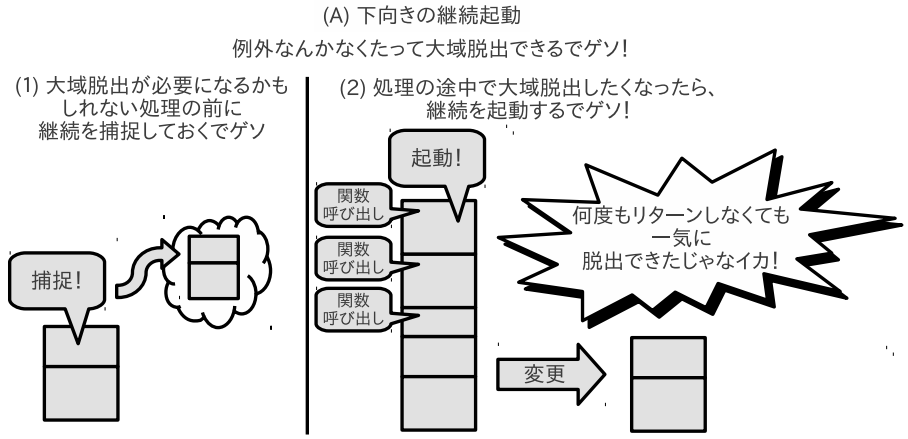
[*6] 実際にはこんな風にはコピーしない実装もいろいろあったりするでゲソが、今回はイメージ重視ということで許して欲しいでゲソ。あと、Scheme みたいに set! とかで環境を破壊的に変更できる場合には、「環境を共有してる他の継続やクロージャーにその変更が見えなきゃ」とか言い出すと単純なコピーだけじゃ駄目なケースがあったりなかったりでゲソが、そういう細かい話も今回は無視するでゲソ。

[*7] 実際には (2)(3) がない言語でも「スタックを積まないようにプログラムを書けば捕捉処理なんかそもそも必要ない

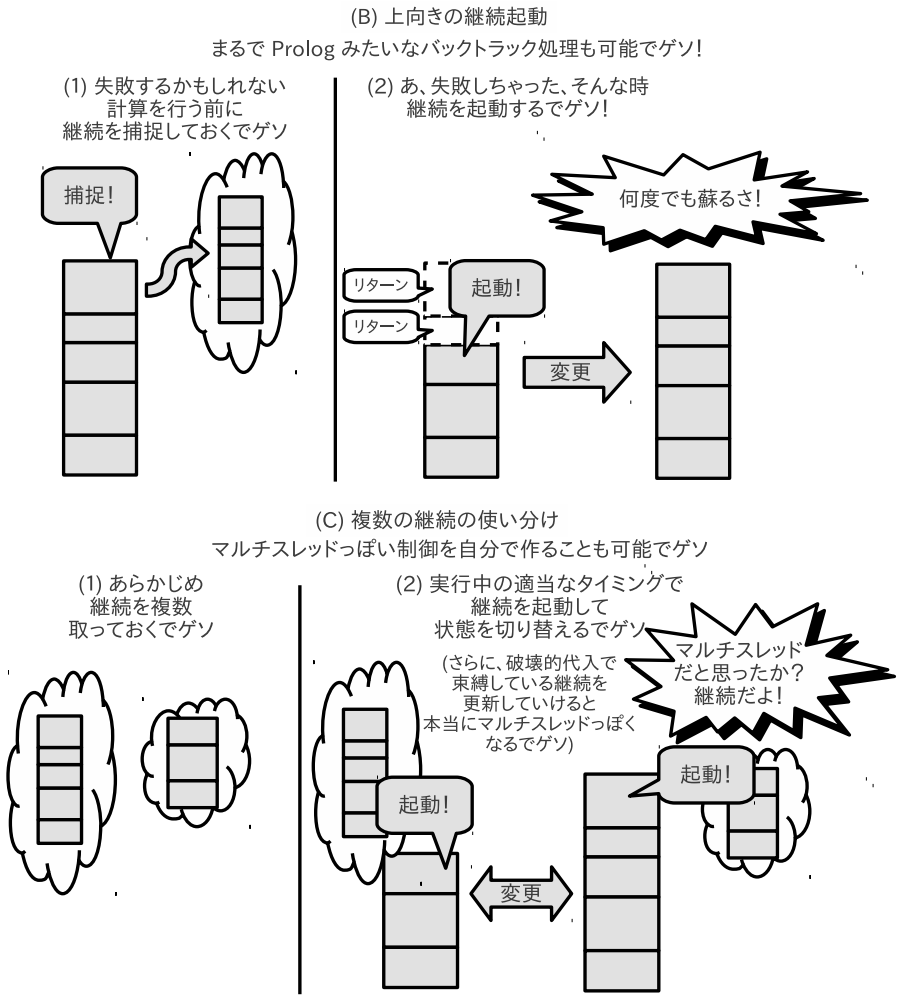


「継続の捕捉/起動が出来る = スタックの待避/復帰が出来る」といってもいいので、工夫次第で結構いろんなことに使えるんでゲソ。例えば次のような使い道が有名でゲソ。

(A) 大域脱出 (例外送出的な処理, setjmp() 的な処理, etc)
(B) バックトラック的な処理 (amb 関数, etc)
(C) ユーザーレベルスレッド (グリーンスレッド, コルーチン, ジェネレータ, ファイバー, etc)



ぜ」っていう「継続渡しスタイル (Continuation-passing style, CPS)」という話があるんでゲソが、それは後で説明するでゲソ。



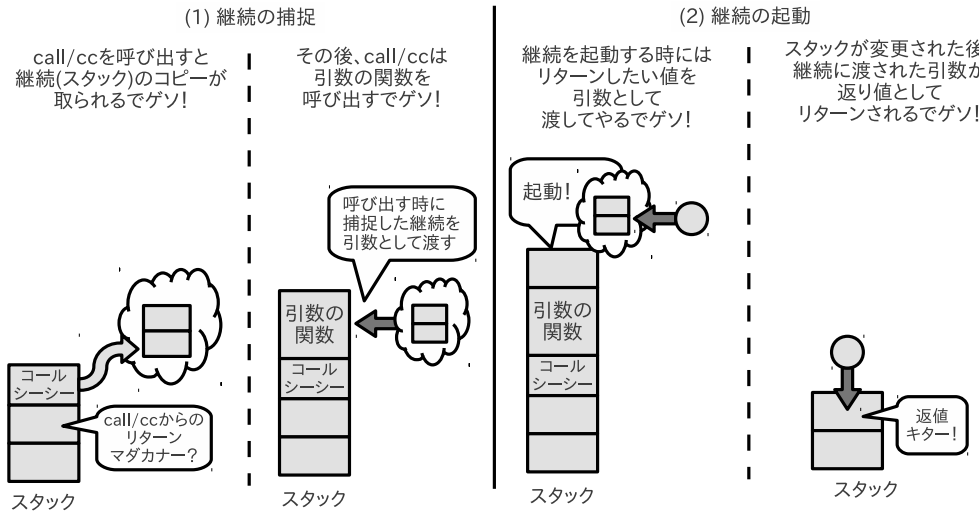
4.1.1 Scheme で「継続の捕捉」を体験してみるでゲソ

さて、何となくイメージが湧いたところで、実際に Scheme で継続の捕捉をしてみようじゃなイカ! Scheme では call/cc [*8] という関数で継続の捕捉が行えるんでゲソ。

最初に call/cc の挙動をちょっとだけ説明しておくでゲソ。下に図があるのでそれを見ながら読んで欲しいでゲソ。まず、call/cc は関数を引数に取るんでゲソ。そして「call/cc が捕捉した継続」がその関数に引数として渡されるんでゲソよ。次に、捕捉した継続を起動したくなったら関数のように引数を適用すればいいんでゲソ[*9]。継続が起動されると、スタックが元の状態に戻された後、継続に渡された引数が返り値として返されるんでゲソ。

[*8] ちなみにこの関数、正式名称は call-with-current-continuation というんでゲソ。でもあまりに長すぎて、正式名称より略称で呼ばれることの方が多いでゲソ。まるで「モジャ公」みたいなものでゲソね。

[*9] ちなみに何で関数のように引数を適用するのかというと、継続と関数には関連性があるからなんでゲソ。それについては CPS の節で一緒に説明するでゲソ。



じゃあ、実際にプログラムの例を紹介してみるでゲソ。call/cc を使ったプログラムは、例えばこんな感じになるでゲソ。

```
(* (call/cc            ; (1) call/cc が継続を捕捉するんでゲソ。
    (lambda (k)        ; (2) 捕捉した継続を引数として、
      (* 42            ;         この関数が呼び出されるでゲソ。
         (k 4))))      ;-- (ここまでが call/cc 呼び出しの式でゲソ)
   (- 0
      (/ 4 2)))
```

さて、この式が評価されるとどうなるか考えてみるでゲソ。

まず、この例で call/cc が捕捉する継続について考えてみようじゃないカ。1 行目から始まっている (call/cc ...) という式の「継続」ってどういうものなんでゲソね? ところで、上では「継続」=「その後に行われる計算処理」=「スレッドの状態」=「スレッドのスタックの中身」と説明したでゲソが、「スレッドの状態」とか「スレッドのスタックの中身」は文字で書き表すのが面倒でゲソ。というわけで、ここでは主に「その後に行われる計算処理」として考えて、「捕捉される継続」がどういう計算処理をするものなのかを考えてみるでゲソ (「継続」=「スレッドの状態」と考えるなら「そういった処理を実行する直前でのスレッドの状態」、「継続」=「スレッドのスタックの中身」と考えるなら「そういった処理を実行する直前でのスタックの中身」と考えてもらえばいいでゲソ)。

じゃあ「継続」が行う処理 (=「call/cc のリターン後に行う計算処理」) について考えてみると、上のプログラムを素直に見ていると、何となく次のような計算処理という気がしなイカ?

```
(* □              ; □と書いている箇所は
   (- 0           ;    「もう既に計算が終わっている箇所」だと思ってほしいでゲソ。
      (/ 4 2)))   ;    (ちなみに□の箇所は元々は call/cc 呼び出しの式があった箇所でゲソ)
```

実際これで正解でゲソ。

あるいはこれを関数のようなものと考えると、この後で行われる計算は次のような関数だとも言えるでゲソ。継続を使うときには関数のように引数を適用するので、関数っぽく考えた方が分かりやすいかもしれないでゲソ。というわけで、この先の話ではこの形で考えてみるでゲソ。

```
(lambda (n)
  (* n
     (- 0
        (/ 4 2))))
```

じゃあ継続を捕捉するところまでは分かったので、次に call/cc が引数の関数を呼び出した後について考えてみるでゲソ。引数の関数が呼び出された後の式の形は、だいたいこんな感じでゲソ?

```
(* (* 42        ; この 2 行分が引数の関数の中身でゲソ
     (k 4))    ;   (ちなみに k は call/cc から引数として渡された継続でゲソ)
  (- 0
     (/ 4 2)))
```

ところで、今回の引数の関数は (* 42 (k 4)) という式を計算するんでゲソが、この k は call/cc から渡された継続でゲソ。ということは、ここに出てくる (k 4) という式は、関数の適用じゃなくて「継続を起動する式」でゲソ。

じゃあ次に、この継続 k が起動されるとどうなるかを考えてみるでゲソ。上の図にも描いたように、継続が起動されるとその時点での継続は全て捨てられて、代わりに起動された k で置き換えられるんでゲソ (上の図では、スタックの中身が捨てられて継続の中身に置き換えられる、という感じになっているけど要は同じことでゲソ)。

というわけで、(k 4) という式を評価した瞬間、式全体は次のようになるでゲソ。

```
((lambda (n)                  ;-- ここからが call/cc が捕捉した継続
   (* n                       ;
      (- 0                    ;
         (/ 4 2))))           ;-- ここまでが捕捉した継続
 4)                           ; ここの 4 が継続に適用される引数
```

(k 4) を評価する前はその外側に (* (* 42 □) (- 0 (/ 4 2))) という式があったはずでゲソが、そこの部分は「継続の起動」によって捨てられてるんでゲソ [*10]。

で、さらに評価を進めてみると、式はこんな感じになるでゲソ。

```
-8      ; 評価値 -8 が得られたでゲソ
```

というわけで、最終的な評価値は -8 でゲソ。

さて、何となく「継続の捕捉」ってどんな感じか分かったでゲソ? 始めて見るとなかなかエキセントリックな感じに見えるでゲソが、よくよく見てみるとそんなに変な感じでもない (はず) でゲソよ。

ちなみに、今回の例の結果をプログラムの最初の式と見比べてみると、「継続の起動」によって (* 42 □) という計算を省略して脱出したようにも見えなイカ? 実際、今回の例は上の (A) (B) (C) で言うと (A) の大域脱出に近い使い方でゲソ [*11]。じゃあ call/cc をどう使ったら (B) や (C) が実現でき

[*10] ところで、この挙動についてもう少し考えてみると、継続は「関数みたいなものだけど絶対にリターンしてこないところが違う特殊な関数」という風にも見えなイカ? まるで exit() 関数とか execve() システムコールみたいな感じでゲソ。この「継続 = 絶対にリターンしてこない関数」という話については、CPS の節でもう一度説明するでゲソ。

[*11] といっても、今回は 1 段しか脱出してないので C 言語系列で言うところの return 程度の脱出にしかなってないでゲソが…

るのか? これについては宿題にするので少し考えてみて欲しいでゲソ。

4.2 継続って誰が考えたんでゲソね?

ところで「継続」なんていう素晴らしい概念、いったい誰が考え出したのか気にならないでゲソ?

今では「継続といえば Scheme」みたいな感じがあるので Scheme 使いが考えたと思ってるかもしれないでゲソが、実は (残念ながら) そうではないんでゲソ。その証拠に Scheme の仕様書である R6RS にも次のような記載があるんでゲソ。

[R6RS Rational 11.9 Control features より]
"Most programming languages incorporate one or more special-purpose escape constructs with names like exit, return, or even goto. In 1965, however, Peter Landin [23] invented a general-purpose escape operator called the J-operator. John Reynolds [28] described a simpler but equally powerful construct in 1972. The catch special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the catch construct could be provided by a procedure instead of by a special syntactic construct, and the name call-with-current-continuation was coined in 1982."

- 引用元: Revised6 Report on the Algorithmic Language Scheme -Rationale-, 2007
  http://www.r6rs.org/final/r6rs-rationale.pdf

[日本語訳]
"ほとんどのプログラミング言語は exit や return、さらには goto といったような名前の目的特化の脱出構文をひとつ以上組み込んでいる。しかしながら、 1965 年、 Peter Landin [23] が J 演算子と呼ばれる汎用の脱出演算子を発明した。1972 年、John Reynolds [28] は、より単純で、同様に強力な構文について述べた。1975 年に Scheme についての報告書で Sussman と Steele によって説明された catch 特殊形式は、名前こそ MacLisp のより一般性のない構文に由来したものであったが、 Reynolds の述べた構文とまったく同じであった。複数の Scheme 実装者は、 catch 構文の能力のすべては特別な構文構造ではなく手続きとして提供できることを指摘し、1982 年に call-with-current-continuation という名前が作られた。"

- 引用元: R6RS:翻訳:Rationale:11.9 Control features, 2008
  http://practical-scheme.net/wiliki/wiliki.cgi?R6RS%3A%E7%BF%BB%E8%A8%B3%3ARationale%3A11.9%20Control%20features

この通り、継続は Scheme が誕生する以前から研究されてたんでゲソよ。じゃあ継続っていったいどこからきたのか不思議じゃなイカ?

実は、継続は手続き型言語の (特に Algol 60 の) goto の研究から生まれてきたんでゲソ。今では「継続」っていうと関数型言語の世界で聞くことが多いから、ちょっと意外でゲソね。

この「継続」の概念がどんな感じで発展してきたかは、上の R6RS の文中でも触れられている John Reynolds 先生が "The Discoveries of Continuations" [12] という論文にまとめているでゲソ [*12]。

[*12] 日本語サイトだと Gauche の作者である川合史郎さんのページに分かりやすいまとめが載ってるでゲソ。
http://blog.practical-scheme.net/shiro/20120122-origin-of-continuations

詳細はこの論文を読んでもらうとして、ここでは簡単に表示的意味論 (denotational semantics) [*13]にまつわる goto と継続の関わりを紹介してみようと思うでゲソ [17]。ちなみに、なんで表示的意味論の話かというと、後で説明するように Scheme 的に少し意味があるからでゲソ! ただし、この話ちょっと面倒なので、時間が無い人は飛ばして次の節に進んでもらっても問題ないでゲソ。

4.2.1 手続き型言語を数学的に説明しちゃうでゲソ

ではさっそく表示的意味論の話に移るでゲソ。手続き型言語の場合にどうやって数学的な概念で説明するか (= どうやって表示的意味論を付けるか) というと、手続き型の本質的な内容は「プログラムの状態の変更」(もっとぶっちゃけるとメモリとかテープとかそういった類いのものの書き換え) と言えるので、プログラミング言語の「文」を「古い状態を受け取って新しい状態を返す関数」として説明するのが一般的でゲソ。この「状態」については、ほんとにメモリの内容のことだと思ってくれてもいいし、もう少し一般化して「何らかの場所を示す値 (変数名とかアドレスとか何でもいいでゲソ) から値への写像」なんだと思ってくれてもいいでゲソ。

じゃあまず goto がない場合について考えてみるでゲソ。 goto がない場合の手続き型言語の代入文は次のような関数として説明できるでゲソ。

元の代入文:

```
X = a
```

対応する数学的な関数:(数学記号を使うのが面倒なのでちょっと `Haskell` っぽく書くでゲソ)

(`\s -> s[n/X]`) (ただし、`n` は `a` という式の評価値に対応する数学的な要素でゲソ)

いきなり勝手に記号を使ったんで読み方が分からない気もするでゲソが、要は「状態 s を受け取って s[n/X] という状態を返す関数」だと思って欲しいでゲソ。で、s[n/X] というのは「X に対応する箇所の値は n で、それ以外の箇所については s と同じ」という状態だと考えて欲しいでゲソ。実際、代入文は対応する変数の箇所を書き換えてそれ以外は書き換えないんだからこれで説明が付きそうな気がしなイカ? (ところで、n については上で書いたとおり「a という式の評価値に対応する数学的な要素」でゲソが、これを具体的にどういう数学要素で説明するかは今回はあんまり重要じゃないので、特に考えないで進もうと思うでゲソ[*14]。)

次に、(goto がない場合の) 複数の文からなる逐次実行文について考えてみるでゲソ。これは、関数合成として説明できるでゲソ。要は、それぞれの文に対応する関数を合成するだけでゲソね。実際、先の文によって状態が変更されたあとで後の文によってさらに状態が変更されるんだから、関数の合成 (先の関数の返値を次の関数の引数にする) とそっくりでゲソ?

元の逐次実行文:

```
S1; S2
```

対応する数学的な関数:(数学記号を使うのが面倒なのでちょっと `Haskell` っぽく書くでゲソ)

(`S2` に対応する関数) . (`S1` に対応する関数) ('.'(ドット)は関数合成演算子でゲソ)

だから、例えば次のようなプログラムはこんな感じで説明できるんでゲソ。

[*13] 「ヒョージテキ・イミロン」というのは、「IT 系の人間はバズワードばっかで何言ってるのかワケわかんねーお。この前も、英語の次はジャバ語、っていうからジャワ島の言葉だと思って知ったかぶったら違ってたんだお。もう全部数学の言葉でしゃべって欲しいんだお」という数学者の期待に応えて、プログラミング言語の全ての概念を数学の概念だけで説明し尽くそうという研究分野でゲソ。「ソーサテキ・イミロン」や「コーリテキ・イミロン」と並んで、プログラミング言語の「意味」を考えようという分野の~~3 バカトリオ~~ 3 大巨頭の 1 つでゲソね。

[*14] きっちりやりたい人は表示的意味論の教科書を見るといいと思うでゲソ (マルナゲ!)

元々のプログラム:

```
X = 1;
Y = 2;
Z = 3;
```

対応する数学的な関数:

```
(\s -> s[3/Z]) . (\s -> s[2/Y]) . (\s -> s[1/X])
= (\s -> s[1/X][2/Y][3/Z])
```

これも勝手に記号を使っちゃってるでゲソが、s[1/X][2/Y][3/Z] というのは ((s[1/X])[2/Y])[3/Z] の略だと考えて欲しいでゲソ。つまり、「Z の箇所は 3、Y の箇所は 2、X の箇所は 1、それ以外は s と同じ」という状態でゲソ [*15]。

念のため、この関数でちゃんと元のプログラムが説明できているのかどうか考えてみるでゲソ。元々のプログラムを実行すると、変数 X の値は 1、変数 Y の値は 2、変数 Z の値は 3 になり、それ以外の変数はそのまま、でゲソね。一方対応する関数の方はというと、状態 s を受け取って「Z の箇所は 3、Y の箇所は 2、X の箇所は 1、それ以外は s と同じ」という状態を返しているでゲソ。というわけで、確かにこのプログラムの意味がちゃんと関数で表現できているじゃなイカ!

じゃあ次に、ここに goto があったらどうなるか考えてみるでゲソ。ただし goto に対応する関数はまだ決めてないので、とりあえず ("goto L1" に対応する関数) という風な書き方をしてみるでゲソ。

元々のプログラム:

```
    goto L1;  /* L1 ラベルまで goto しちゃうでゲソ */
    Y = 2;    /* この文は goto で飛ばされるので実行されないでゲソ */
L1: Z = 3;
```

対応する数学的な関数??:

```
(\s -> s[3/Z]) . (\s -> s[2/Y]) . ("goto L1"に対応する関数)
```

さて、この関数がちゃんと対応しているのかどうかを考えてみるでゲソ。元々のプログラムを実行すると、変数 Z の値は 3、それ以外の変数はそのまま、でゲソね。じゃあ、対応する関数の方はというと、("goto L1" に対応する関数) のところがはっきりしないでゲソが、`(\s -> s[3/Z])` と `(\s -> s[2/Y])` を関数合成しちゃってるので、少なくとも Z の箇所は 3、Y の箇所は 2 になった状態が返されるはずでゲソ? だけど、"Y = 2" の文は goto で飛ばされて実行されないはずだから、この効果まで関数の方に入ってしまってるのはまずいじゃなイカ!

4.2.2 どうやったら goto に対応できるんでゲソ?

というわけで、単純な拡張では goto は説明できなさそうなことが分かったところで、何でうまくいかないのかを少し考えてみようと思うでゲソ。

そもそも以前のアプローチは何をしてたかというと、「1 つ 1 つの文に対してそれぞれ対応する関数が見つかるはず」という考えで関数を見つけようとしてたでゲソ。実際、goto がない場合には 1 つ 1 つの文にちゃんと対応する関数があったので、それらを見つけて関数合成していくことでプログラム全体に対応する関数が作れたんでゲソ。

だけどこのアプローチって goto 文にも通用するんでゲソか?

というのも、goto 文ってそれ単体で意味が付くのかどうかちょっと怪しくなイカ? 単体で意味が

[*15] 正確に言うなら、「Z に対応する箇所の値は 3 で、それ以外の箇所については「Y に対応する箇所の値は 2 で、それ以外の箇所については「X に対応する箇所の値は 1 で、それ以外の箇所については s と同じ」」」ということでゲソ。

付くってことは、他の文とは無関係に、状態に (=メモリの内容とかに) どう影響を及ぼすかがはっきり言えるということでゲソ。だけど goto 文はどちらかというと「制御フローを変える (=後続の処理を変える) ことで間接的に状態に影響を与える」という感じでゲソ。ということは、goto 文については (単体では意味は付けようがなくて) 飛び先とセットで考えないと駄目なんじゃなイカ?

というわけで、ここで求められるのは発想の逆転でゲソ。「逆に考えるんだ。「1 つ 1 つの文に対して対応する関数が見つからなくってもいいさ」と考えるんだ」ということでゲソね。

で、ここまで説明してようやく「継続」の出番でゲソ。1 つ 1 つの個別の文に対して関数が見つからなくても、「その文から始まってプログラム終了までに実行される全ての文の列」に対してならどうでゲソ? 言い換えると、文単体じゃなくて「文+その文の継続」というセットで考えればいいんじゃなイカ、というのが新しいアプローチでゲソ。

4.2.3 新しいアプローチでもう一回説明し直してみるでゲソ

じゃあ具体的に「セットで考える」というのがどういうことか、例を出しながら説明してみるでゲソ。で、最初にまず注意が必要なんでゲソが、上では「文」を「古い状態を受け取って新しい状態を返す関数」として説明したじゃなイカ? で、これは「その文の実行直前の状態→その文の実行直後の状態」という関数だったでゲソ。でも、今度は「文+その文の継続」のセットで考えるので、「その文の実行直前の状態→プログラム終了時の状態」という関数として説明することになるんでゲソ (正確に言うと、これもちょっと違うので正しい形を後でもう一度紹介するでゲソ。とりあえずはこういうことにして先に進んで欲しいでゲソ)。

じゃあ、もう一度代入文について考えてみるでゲソ。

元の代入文:

```
X = a
```

対応する数学的な関数:(数学記号を使うのが面倒なのでちょっと Haskell っぽく書くでゲソ)

```
(\k  ->  k . (\s -> s[n/X]))
```

何やら記号が増えて読みづらくなってるでゲソね。でもやってること自体はそんなに難しくないでゲソ。

上で説明した「セットで考える」というのは、(この代入文だけじゃなく) その後でプログラムが終了するまでに実行される全ての文も一緒に考えないとイカん、ということでゲソ。とはいえ、この代入文の後にどんな文が実行されるかなんて、この代入文だけからでは分からないでゲソね。というわけで、分からないところは未知数として残すということで、上の関数だと k という引数でそれを表しているんでゲソ [*16]。

で、この k、具体的には何かと言うと「この代入文の次の文からプログラム終了時までに実行される全ての文」に対応する関数でゲソ。言い換えると、「この代入文の次の文の実行直前での状態→プログラム終了時の状態」という関数でゲソ。

さて、元々の目的はこの代入文を関数として説明することだったでゲソ。そのために今作りたいのは「この代入文の実行直前の状態→プログラム終了時の状態」という関数でゲソ。で、上のような k が引数として与えられたとすると、これを作るのは簡単じゃなイカ?

具体的にどうするかと言うと、まず "`(\s -> s[n/X])`" という関数を作ると、(これは上で説明したとおり)「この代入文の実行直前の状態→この代入文の実行直後の状態」という関数になってるでゲソ。で、この関数と上の k を関数合成すると、「この代入文の実行直前の状態→この代入文の実行直後の状態」と「この代入文の次の文の実行直前での状態→プログラム終了時の状態」の合成

[*16] 継続について詳しい人は「継続渡しスタイル (Continuation-passing style, CPS)」だと思ってくれればいいでゲソ。CPS については後で説明するでゲソ。

と言うことになるでゲソ? これって、「この代入文の実行直前の状態→プログラム終了時の状態」という関数のことじゃなイカ！ 実際に上で紹介した数学的な関数の定義を見てみると、この２つの関数を合成しているだけでゲソ。

というわけで、代入文についてはちゃんと説明できそうじゃなイカ。ただし、説明に使用する関数の形は以前とは少し変わってるでゲソね? 具体的に言うと、こんな感じになってるでゲソ。

[以前のアプローチ]
その文の実行直前の状態→
その文の実行直後の状態

[今回のアプローチ]
（次の文の実行直前での状態→プログラム終了時の状態）→
その文の実行直前の状態→
プログラム終了時の状態

これだけ見るとなんだか無駄に複雑になっているように見えるでゲソ... でも全ての文をこの形の関数で説明すると、goto 文にも上手く説明が付くんでゲソ。まぁとりあえずは次に進もうじゃなイカ。

じゃあ次に、逐次実行文について考えてみるでゲソ。

元の逐次実行文:
`S1; S2`
対応する数学的な関数:（数学記号を使うのが面倒なのでちょっと `Haskell` っぽく書くでゲソ）
（`\k -> S1` に対応する関数（`S2` に対応する関数 `k`））

逐次文の場合も、「この逐次文から始まってプログラム終了時までの文全体」を考えないとイカんということになるでゲソ。といっても、この場合も逐次文の後にどんな文が実行されるかなんて分からないでゲソ。なので、分からない部分を k という引数として受け取る、という形にしているのは代入文の場合と同じでゲソ。

じゃあ代入文の時と同じように、対応する関数についてもう少しよく考えてみるでゲソ。今欲しかったのは、上で説明したように「(この逐次文全体の次の文の実行直前での状態→プログラム終了時の状態) →この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数でゲソ。

で、まずこの逐次文の要素のうち S2 文の方について考えてみるでゲソ。上と同じように考えると、S2 文に対応する関数は「(S2 の次の文の実行直前での状態→プログラム終了時の状態) → S2 の実行直前の状態→プログラム終了時の状態」ということになるでゲソ?

ところで、「S2 の次の文」って「この逐次文全体の次の文」のことじゃなイカ?

ということは、S2 に対応する関数の第一引数である「(S2 の次の文の実行直前での状態→プログラム終了時の状態)」というのは、引数の k のことでゲソ! ということで、S2 に対応する関数に k を適用してやれば、「S2 の実行直前の状態→プログラム終了時の状態」という関数が作れるでゲソ (カリー化してると考えて欲しいでゲソ)。

次に S1 文の方について考えてみるでゲソ。 S1 文に対応する関数も上と同じで「(S1 の次の文の実行直前での状態→プログラム終了時の状態) → S1 の実行直前の状態→プログラム終了時の状態」ということになるでゲソ。

ところで、「S1 の次の文」って S2 でゲソね?

ということは、この関数の第一引数である「(S1 の次の文の実行直前での状態→プログラム終了

時の状態)」って、さっき作った「S2 の実行直前の状態→プログラム終了時の状態」のことじゃなイカ。なので、さっきと同じように S1 に対応する関数に “(S2 に対応する関数 k)” を適用してやれば、「S1 の実行直前の状態→プログラム終了時の状態」という関数が作れるでゲソ!

で、そもそも何が作りたかったのかをもう一度考えてみると、作りたかったのは「(この逐次文全体の次の文の実行直前での状態→プログラム終了時の状態) →この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数だったでゲソ? ところで、「この逐次文全体の実行直前の状態」って「S1 の実行直前の状態」と同じ意味じゃなイカ?

ということは、「この逐次文全体の実行直前の状態→プログラム終了時の状態」という関数は、さっき作った “(S1 に対応する関数 (S2 に対応する関数 k))” のことでゲソ。

というわけで、引数 k を受け取って (S1 に対応する関数 (S2 に対応する関数 k)) を返せば目的の関数になるでゲソ! (実際、上で紹介した数学的な関数の定義はその通りになっているでゲソね)

うーん、なんだか代入文や逐次実行文は新しい方式でも上手く説明できそうでゲソね。

それでは、最後にいよいよ goto 文について考えてみるでゲソ。 goto 文についても同じように「(goto 文の次の文の実行直前での状態→プログラム終了時の状態) → goto 文の実行直前の状態→プログラム終了時の状態」という関数として説明できるでゲソか?

元の `goto` 文:

```
goto L1;
```

対応する数学的な関数:(数学記号を使うのが面倒なのでちょっと `Haskell` っぽく書くでゲソ)

(`\k -> k1`) (ただし、`k1` は「`L1` ラベルが付いている文 + その継続」に対応する関数)

さて、goto 文は (予想に反して (?)) なんだか結構シンプルな関数になってるでゲソね。

じゃあ、まずは第 1 引数である「(goto 文の次の文の実行直前での状態→プログラム終了時の状態)」について考えてみるでゲソ。

ところでこの「次の文」の扱いなんでゲソが、goto 文の場合はこれまで考えた代入文や逐次実行文と少し違うでゲソね? 代入文や逐次実行文は、実行が終わったらそのまま次の文に制御が移ってたじゃなイカ。でも、goto 文って (引数のラベルに飛んでっちゃうから) 次の文とか関係ないでゲソ。

というわけで、goto 文の場合は第 1 引数の「(goto 文の次の文の実行直前での状態→プログラム終了時の状態)」については全く使う必要がないんでゲソ (というか、むしろ使っちゃ駄目というか)。実際、上の数学的な関数の中でも引数の k は使われてないでゲソね。そしてこれが以前のアプローチとの違いなんでゲソ。つまり、以前のアプローチだと問答無用で「次の文」と関数合成されちゃってたんでゲソが、今回のアプローチだと「次の文」が重要じゃなければその効果を勝手に捨てちゃうことができるんでゲソ。

じゃあ次に、残りの「goto 文の実行直前の状態→プログラム終了時の状態」の部分を考えてみるでゲソ。といっても、goto 文自体はプログラムの状態を変更したりはしないじゃなイカ? goto 文が実行されたら飛び先の文にジャンプして、後はそこから普通に実行が続けられるだけでゲソ。ということは、この「goto 文の実行直前の状態→プログラム終了時の状態」って「goto 文の飛び先の文の実行直前の状態→プログラム終了時の状態」と同じでないと変でゲソ?

というわけで、この話をまとめてみると、(引数 k を受け取るんだけど、この k についてはさっぱり無視して)「飛び先の文 + その継続」に対応する関数を返せばいい、ということでゲソね。これが上で紹介した数学的な関数の箇所に書いてある内容でゲソ。

どうでゲソ? たしかに goto 文の挙動が説明できていそうな感じじゃなイカ? じゃあ、この新しいアプローチでちゃんとプログラムが説明できるのか確かめてみるでゲソ!

例 1：以前のアプローチでも上手くいっていた例

元々のプログラム：

```
X = 1;
Y = 2;
Z = 3;
```

対応する数学的な関数：

```
(\k ->                                  # この逐次文全体の継続 k を受け取る
     (\k1 -> k1 . (\s -> s[1/X]))         #（これが 1 行目に対応する関数）
        (\k2 -> k2 . (\s -> s[2/Y]))      #（これが 2 行目〃）
           ((\k3 -> k3 . (\s -> s[3/Z])) #（〃 3 行目〃）
              k))                         # 全体の継続である k を適用
= (\k -> \s ->  k (s[1/X][2/Y][3/Z]))  # 整理してまとめるとこんな感じ
```

例 2：以前のアプローチでは上手くいかなかった例

元々のプログラム：

```
     goto L1;  /* L1 ラベルまで goto する，という意味でゲソ */
     Y = 2;    /* この文は goto で飛ばされるので実行されないでゲソ */
L1: Z = 3;
```

対応する数学的な関数：

```
(\k ->                                  # この逐次文全体の継続 k を受け取る
     (\k1 ->                              #（これが 1 行目に対応する関数）
          ((\k3 -> k3 . (\s -> s[3/Z]))  #   飛び先である 3 行目に対応する関数と
               k))                       #   飛び先の継続である k を
                                         #   合成したものを返す
        (\k2 -> k2 . (\s -> s[2/Y]))     #（これが 2 行目〃）
           ((\k3 -> k3 . (\s -> s[3/Z])) #（〃 3 行目〃）
              k))                         # 全体の継続である k を適用
= (\k -> \s ->  k (s[3/Z]))             # 整理してまとめるとこんな感じ
```

まずは元々上手くいっていた例について、もう一度見ておくでゲソ (もちろんさっき上手くいったからもう考えなくてもいいんでゲソが、念のため新しいアプローチでも大丈夫かどうかを確認しておくでゲソ)。ちょっと注意が必要なのは、さっきと違って、このプログラムだけじゃなく「このプログラム + その継続」というセットで考えるんでゲソ。(つまり、この例では3つの文しかないでゲソが、この後ろにも色々と処理があるんだと思って考えて欲しいでゲソ。[*17])

元々のプログラムを実行した時に起こることを (その継続まで含めて) 考えると、まず上の 3 行によって「変数 X の値は 1、変数 Y の値は 2、変数 Z の値は 3、それ以外の変数はそのまま」という状態になるでゲソね。そしてその後、このプログラムの「継続」によってさらに状態が変更されるでゲソ。

一方対応する関数の方はというと、継続 k と状態 s を受け取って「Z の箇所は 3、Y の箇所は 2、

[*17] あるいは、この後ろに本当に処理がない場合なら NOP(何もしない命令) が継続だと考えてくれればいいでゲソ。その場合、数学的な関数の方では「恒等関数 (\x -> x) が継続」ということに対応するでゲソ。

X の箇所は 1、それ以外は s と同じ」という状態を作り、それを k に適用した結果を返すでゲソ。というわけで、やっぱりこのプログラムの意味がちゃんと関数で表現できているじゃなイカ!

じゃあ次に、前は上手くいかなかった例について考えてみるでゲソ。今度もさっきと同じように「このプログラム + その継続」というセットで考えてみるでゲソよ。

元々のプログラムを実行すると、上の 3 行によって「変数 Z の値は 3、それ以外の変数はそのまま」という状態になり、その後でさらにこのプログラムの「継続」によって状態が変更されるでゲソ。

じゃあ対応する関数の方はというと、継続 k と状態 s を受け取って「Z の箇所は 3、それ以外は s と同じ」という状態を作り、それを k に適用した結果を返すでゲソ。うむ、これはまさしく上のプログラムの挙動そのものじゃなイカ!

というわけで、「継続」という考え方を入れてみると goto 文があってもプログラムの意味をちゃんと関数で説明できるということが分かったでゲソ? 今回紹介したのは簡単な例でゲソが、if とか while、あるいは関数みたいな制御構造と組み合わせても同じやり方でちゃんと goto 文が説明できるんでゲソ。気になる人は、上の論文を見ながら自分で色々な例を確かめてみて欲しいでゲソ。

4.2.4 最後に Scheme との関係を話しておくでゲソ

さて、長々と説明してきたんでゲソが、このちょっと面倒な表示的意味論の話をしたのは Scheme 的には 1 つ理由があるんでゲソ。 Scheme の意味論も R5RS(1 世代前の言語仕様) [2] の頃は表示的意味論で定義されてたんでゲソが[*18]、そのベースになっていたのがこの Algol 用に作られた「継続」を使ったやり方なんでゲソ。ということは、この表示的意味論の読み方を知っていれば Scheme の意味についてはバッチリ [*19]、ということでゲソ！ 嘘だと思ったら R5RS に書かれているルールとこの論文のルールを見比べてみるでゲソ。 if 式だとか nop だとか逐次実行みたいに、(表記法はちょっと違うでゲソが内容としては) ほとんど丸写しになっている箇所まであるでゲソよ。

[*18] ただし現行仕様である R6RS になった時に操作的意味論による定義に変更されてしまったでゲソが…

[*19] 上に書いたように「R5RS については」でゲソが…

付録 : Algol 60 と愉快な goto たち

上で簡単に触れた通り、継続の研究に大きな影響を与えたのは Algol 60 なんでゲソが、これは Algol 60 の goto が特段ややこしかったからという理由もあるんでゲソ。 Algol 60 は lexical scope を導入した言語としても有名でゲソが、さらに関数の中で関数を定義できる (nested function) という機能も持っていたんでゲソ。
え、「それがどうしたんでゲソ? 」って?
問題になったのは goto が使用する「飛び先のラベル」も lexical scope を持ってたってことでゲソ。つまり、「内側の関数から外側の関数内のラベルに goto する」という芸当が Algol 60 ではできたんでゲソ！
少し例を書いてみるでゲソ。

```
procedure S;
begin
  procedure T;
  begin
    integer procedure U;
    begin
      ...
      goto L1      # (3) U の中からラベル L1 まで 2 段階の大域脱出でゲソ！
    end U;
    integer a;
    a := U         # (2) T の中で、U が呼び出されるでゲソ。
  end T;
  T;               # (1) 関数 S の中で、まず T が呼ばれるでゲソ。
L1:                # (4) 関数 S よ！ 私は帰ってきた！！
                   #     (ってそれは goto じゃなくてガトー (Gato) さんでゲソ)
end S;
```

さぁ、何が起こるか分かるでゲソ? この場合、関数呼び出し 2 つ分の大域脱出になるので、当然スタックフレームを 2 つ破棄しないといけないんでゲソ。というわけで、goto なのにスタックの中身も一緒に考えないと説明できなくなったんでゲソ。
実際問題として、こんな面倒くさい挙動がなければ、プログラムを goto のない部分ごとに分割して (コンパイラ用語で言うところの基本ブロックでゲソね) それぞれに対して対応する関数を見つけてうまくつなぎ合わせる、という方法も取れたんでゲソ。だけど、こんな風に関数の外側に向けて goto されたりすると流石に基本ブロックとかでは説明が付かないので、もうちょっと手の込んだ解決策が必要になったんでゲソ。これが「goto って結局何なんだ? 」という話になって、継続の研究につながったそうでゲソ。
ちなみに、Algol 60 は後で説明するように Scheme にも大きな影響を及ぼしたんでゲソ。いろんな意味ですごい言語だったんでゲソね。

4.3　なんで Scheme と継続が関係あるのか不思議じゃなイカ?

前の節で説明したように、元々は「継続」は Algol とか goto に絡んだ話だったんでゲソ。それがなんで Scheme と結びついたのか不思議じゃなイカ?

実はこれ、Scheme の生い立ちと関係があるんでゲソ。というわけで、少し Scheme の誕生のいきさつを話しておこうと思うでゲソ [*20]。

Scheme が誕生したのは 1970 年代の~~まちゃちゅーちぇっちゅちゅー~~マサチューセッツ工科大学 (MIT) でゲソ。当時の MIT は人工知能研究のメッカで、人工知能の研究用にいろんなプログラミング言語が作られてたんでゲソ。特に人工知能にはどんな制御構造がいいのか、という話が盛んで (例えば後の Prolog で使われるバックトラックみたいなものも含めて) いろんな制御構造が研究されていたそうでゲソ。

そんなある日、当時生まれたばかりの Simula や Smalltalk といったオブジェクト指向言語を見ていた MIT の Carl Hewitt 先生は、アクターモデル (Actor model) という新しいプログラミング・パラダイムを思いつくんでゲソ。このアクターモデル、オブジェクト指向言語から思いついたので基本的にはオブジェクト指向に似た感じのパラダイム [*21] なんでゲソが、特徴としてすごく画期的な制御構造を持っていたんでゲソ。

こう言われると、アクターモデルの制御構造が気になってこなイカ? 実はアクターモデルの制御構造は基本的に 1 つしかなくて、アクターからアクターにメッセージを送信する (メッセージパッシングする)、というものだったんでゲソ!

えっ、あんまり画期的な感じがしない? というか、普通のオブジェクト指向言語における「メソッド呼び出し」のことじゃなイカって?

うむ、確かにメッセージパッシングは少し「メソッド呼び出し」と似てるでゲソ。ただし、この 2 つには大きく違うところがあるんでゲソ。それは、メッセージパッシングでは「呼び出し元に値をリターンする」という概念がないことでゲソ。つまり、アクターでは「メッセージを送る」という制御しかなくて、(基本的には向こうからは返ってこない) 一方通行の制御フローになるんでゲソ。

うん? なんだか画期的というよりは使い勝手が悪そうでゲソって?

確かにこの説明だけ聞くと、ちょっと柔軟性に欠けそうな感じがするかもしれないでゲソ。一応、それぞれのアクターにはインスタンスフィールドみたいなもの [*22] もあって、他のアクターへの参照を入れておけばそのアクターにはメッセージを送れるでゲソ。だから、お互いに参照を持ち合っていればメッセージを往復させたりするのは簡単でゲソ。

だけど、あらかじめ決まった相手にしかメッセージを送れないとなるとなんだか不便そうでゲソね。例えば「数学関数を実装したオブジェクト」みたいなものを作りたくなったらどうするでゲソ? 普通のオブジェクト指向言語なら「誰からでも呼び出せて数学関数の処理結果を返り値として受け取れる汎用的な数学処理オブジェクト」を作ることができるでゲソ。こんな風な機能をアクターで実現したくなったらどうすればいいんでゲソ?

でも心配はいらないんでゲソ。この制御パラダイムが画期的だったのは、あるアイデアを使うことで (こういったメソッド呼び出しどころか、もっと複雑な制御まで含めて) いろんな制御を全て実現できたからなんでゲソ。じゃあそのアイデアが気になるでゲソね? 実はそれは「メッセージ内に

[*20] この話を詳しく知りたい人は、bit 誌に掲載されていた “Scheme 過去◇現在◇未来” という記事がお勧めでゲソ。“Scheme” という名前の由来なんかも分かるでゲソよ。

[*21] 例えば、アクターモデルでは (オブジェクト指向の「オブジェクト」に相当する)「アクター」という操作対象を持つんでゲソ。そして、全てのモノをアクターとして説明するんでゲソ。オブジェクト指向でいうところの「全てのモノはオブジェクト」みたいな感じでゲソ。

[*22] アクター用語では “acquaintances(知り合い)” というようでゲソ。

計算結果の受け取り先になるアクターも一緒に入れておけばいいじゃなイカ！」というものなんでゲソ。

例がないとちょっと分かりにくいので、アクターモデルで作られた cons セル (コンスセル) を紹介してみるでゲソ [*23]。Carl Hewitt 先生がアクターの実証用に作った “Plasma” という言語で書くと、こんな感じになるみたいでゲソ (正確には、いい例が見当たらなかったので適当に Plasma の論文 [7] を見ながら作ってみた例でゲソ。文法的なレベルでかなり間違ってそうな気がするので、雰囲気だけ感じとって欲しいでゲソ)。

```
[CONS ≡
            ; (1) 次の行は CONS アクターが受け取るメッセージを示しているでゲソ
            ;     (パターンマッチしてるんでゲソが、要は A,B,C の３要素を受け取るんでゲソ)
  (≡≡>  (request: [=A =B] (reply-to: =C))

            ; (2)CONS アクターは出来た結果を C に送信するんでゲソ。
            ;     この下の (CASE ..) の部分が渡すメッセージでゲソ。
            ;     (ちなみに、この (CASE ..) 自体も
            ;      新しいアクター (cons セルアクター) でゲソ)
    C <==
        (CASES

            ; (3) cons セルアクターは、CAR というメッセージを
            ;     (K というアクターと一緒に) 受け取ると
            ;     K に 1 番目の要素 (つまり A) を送信するでゲソ。
            (≡≡>  (request: CAR (reply-to: =K))
                   K <== (reply: A))

            ; (3) cons セルアクターは、CDR というメッセージを
            ;     (K というアクターと一緒に) 受け取ると
            ;     K に 2 番目の要素 (つまり B) を送信するでゲソ。
            (≡≡>  (request: CDR (reply-to: =K))
                   K <== (reply: B))

            ; (3) cons セルアクターは、LIST?というメッセージを
            ;     (K というアクターと一緒に) 受け取ると
            ;     K に「そうです (YES)」と送信するでゲソ
            (≡≡>  (request: LIST? (reply-to: =K))
                   K <== (reply: YES))))]
```

慣れない文法のソースコードは読みにくいでゲソね。でも何となく意味は分かったりしないでゲソ? “≡≡> ” というのがメッセージ受信時の処理を、“<==” というのがメッセージ送信を表してるんでゲソ。

[*23] ところで cons セルって Lisper じゃない人にどのくらい知名度があるのか若干気になるでゲソ。知らない人は普通に 2 要素のタプルだと考えてもらえばいいでゲソ。ちなみに、歴史的な経緯により、1 番目の要素を取り出す関数を car(カー)、２番目の要素を取り出す関数を cdr(クダー) と呼ぶんでゲソ。

さて、格好いい (?) 文法とかはさておくとして、ここで重要なのは C とか K なんでゲソ。CONS アクターも cons セルアクターも、メッセージの一部として結果の受け取り先となるアクター (C とか K) を受け取るんでゲソ。こんな風に受け取り先を指定できるようになってるので、どのアクターからでも CONS アクターや cons セルアクターが利用できるんでゲソ。ところでこの C とか K って、CONS アクターや cons セルアクターの立場から見ると「自分が渡した値を使ってこの後の計算処理を行うアクター」ということにならなイカ?「この後の計算処理」つまり「継続」でゲソ。

実際、上の論文中でもこの C とか K は「継続」って呼ばれてるんでゲソ。つまりアクターモデルでは、メッセージパッシングという一方通行な制御だけで関数的な制御を実現するために、「継続」を渡す、というテクニックを使ってるんでゲソ。で、メッセージを送られた側では計算結果をリターンする代わりに「継続」に結果を送信するんでゲソ。

初めて見た人にはかなり変わった制御のように見えるかもしれないでゲソ。だけど、Carl Hewitt 先生はこのメッセージパッシングと継続という構造だけでいろいろな制御構造が作れる、ということを次々に発見していくんでゲソ [7]。

で、このアクターモデルに興味を持ったのが Gerald Sussman 先生と Guy Steele 先生。Scheme の生みの親になる 2 人でゲソ。

ところが、当時のアクターモデルは初心者にはなかなか厳しいものだったようでゲソ。というのもさっきの Plasma 言語が複雑すぎて、アクターの何がそんなにすごいのか、2 人にもよく分からなかったそうでゲソ (さっきの例だけ見ても、Plasma が結構複雑な言語だというのは理解してもらえるんじゃなイカ?)。そこで 2 人は「自分たちでアクター用の言語を 1 つ作ってみればアクターの本質が分かるんじゃなイカ」と思いついたんでゲソ、、、って勘のいい人ならここら辺で気づくかもしれないでゲソが、この言語こそが Scheme なんでゲソ。つまり、Scheme は元々はアクターモデルを勉強するための言語だったんでゲソね [*24]。

というわけで、これが Scheme と「継続」の縁が深い理由なんでゲソ。そもそもの Scheme が生まれるきっかけになったアクターモデルが「継続」をとても重要視したんでゲソね。

ところで、今の Scheme には存在しないでゲソが、開発当初の Scheme にはちゃんとアクターを作る関数 [*25] が用意されていたようでゲソ [*26]。

```
(alpha (parameters ... ) body)
```

この alpha は関数を作る lambda に (見た目も含めて) よく似てるんでゲソが、アクターなので値をリターンしたりはしないんでゲソ。代わりに計算結果を他のアクターにメッセージパッシングするんでゲソよ。例えば階乗計算を行うアクターはこんな感じになるんでゲソ (もう面倒だから説明はしないでゲソ。CONS アクターが読めたならきっとこれも読めるはずでゲソ (?))。

[*24] ところで、Scheme が Lisp っぽい言語になったのは当時 MIT で Lisp が主流だったからでゲソが、伝統的な Lisp ってみんな dynamic scope だったんでゲソ。それなのになんで Scheme が lexical scope を持つようになったかというと、(アクター理論では lexical scope が必要そうに見えたというのもあるんでゲソが、もう 1 つの理由として) Sussmann 先生が Algol 60 にハマっていたから、というのもあるんでゲソ。つまり Algol がなければ Scheme は全然違った言語になっていたかもしれないんでゲソ。で、実際この lexical scope を採用したことが後で大きく効いてくるんでゲソ。上で書いた「Algol が Scheme に大きな影響を与えた」というのはこういう意味なんでゲソ。

[*25] (細かいことなのでどうでもいいでゲソが) 正確には関数じゃなくていわゆる「特殊形式 (Special form)」でゲソ。

[*26] ちなみに、アクターにメッセージパッシングする send という特殊形式もあったようでゲソ。構文としては “(send actor argument1 … argumentN)” のように書いたそうでゲソ。ただ、残念なことに (?) インタープリタ内では関数とアクターは区別が付いちゃうんでゲソ。ということで、(関数呼び出しもメッセージパッシングも) どっちも関数適用と同じ記法にしておいて、関数なら関数呼び出し、アクターならメッセージパッシングにした方が便利じゃなイカ、ということでこちらは早々に削除されたようでゲソ [15]。

```
(define factorial
  (alpha (n c)
    (= n 0
      (alpha () (c 1))
      (alpha ()
        (- n 1
          (alpha (z)
            (factorial z
              (alpha (y)
                (* n y c)))))))))
```

ところが、実際に Scheme インタープリタが出来てみると、驚くべきことが分かったんでゲソ。なんと、インタープリタ内でアクターを処理する部分と関数 (クロージャ) を処理する部分の実装が同じになってしまってたんでゲソ！ これはどういうことでゲソ? 本質的に違いがあるものなら全く同じに処理できるわけはないでゲソ。ということは...

だいたい予想が付いたかもしれないでゲソが、この結果を見て 2 人が出した結論は、「アクターは Lisp の関数と同一のものだ」というものだったんでゲソ[*27]。ちなみに、アクターのメッセージパッシングは Lisp の関数では末尾位置での呼び出しに対応したんでゲソ。というわけでより正確に言うと「アクターは末尾呼び出ししかしない Lisp の関数だ」でゲソね [*28]。(ついでに言うと、「継続」は「末尾位置で呼び出される関数」のことでゲソね)

上で紹介した alpha が今の Scheme に残っていない理由もこれで納得いったでゲソ？ alpha と lambda は本質的に同じなんだから両方用意しておく必要はないでゲソ。

ただし、アクターを Lisp の関数と同一視するにはちょっと問題もあったんでゲソ。アクターモデルでは、アクター同士で何度メッセージパッシングしてても特に問題は起こらないんでゲソ。ところが Lisp の関数の場合は、関数呼び出しをナイーブに実装していると (末尾だろうと末尾でなかろうと) とにかく関数を呼び出すたびにスタックフレームを積んでしまうでゲソ。ということは、アクターを真似して末尾呼び出しを延々と繰り返していると、そのうちスタックオーバーフローしてしまうじゃなイカ！ これでは、(理論上はさておき) 実用上はとても同一のものとは言えないでゲソ。

そこで、アクターパラダイムを完全に Lisp の世界に取り込むために「末尾呼び出しは最適化して goto にすべし (スタックを積んではならん)」という言語仕様が Scheme に取り込まれたんでゲソ。これが、Scheme において「継続」と並んで「末尾呼び出しの最適化 (Tail Call Optimization, TCO)」が特別に重要視される理由なんでゲソ。

ちなみに末尾呼び出しの最適化という概念自体、関数型の世界に登場したのはこのときが初めてだったようでゲソ。今となっては当たり前の末尾呼び出しの最適化にも、そもそもはこういう経緯があったんでゲソ。

[*27] ちなみに、これが lexical scope を採用したことによる効果だったんでゲソ。つまり、伝統的な dynamic scope な Lisp ではアクターは関数とは一致しないでゲソ。

[*28] ちなみに、Carl Hewitt 先生は「アクターは並列性とかの面でクロージャーに勝っていてむしろアクターの方がすごいだろ」と主張されてたりするようでゲソ。でも、今回は主に Scheme 派の主張を取り上げるでゲソよ。気になる人はアクターの方も調べてみるといいと思うでゲソ。

付録: Lambda-ka: The Ultimate Invader
(または Lisp が如何にして心配するのを止めてラムダ計算を愛するようになったか)

上で説明したように Scheme はアクター理論と Lisp を結びつけたんでゲソが、実はもう一つ大事なものを結びつけたんでゲソ。 Scheme の特徴である「lexical scope」と「末尾呼び出しの最適化 (TCO)」、これって、Lisp というより、その元になったラムダ計算に近くなイカ?
例えば、ラムダ計算の評価規則であるβ簡約って、変数の参照先はプログラム中の構文構造で決まるでゲソ? これはまさしく lexical scope でゲソね。
もう一つの TCO については、次の fact の例を見てみるでゲソ。(もちろん本当のラムダ計算には数値とか named-let みたいなものはないでゲソが、ここではチャーチ数とか Y コンビネータのシンタックスシュガーだと思って欲しいでゲソ)

末尾再帰の場合:

```
(define (fact n)
 (let loop ((n n) (acc 1))
  (if (= n 1) acc
   (loop (- n 1) (* acc n))))))
```

末尾再帰じゃない場合:

```
(define (fact n)
 (if (= n 1) 1
  (* n (fact (- n 1))))))
```

この 2 つをβ簡約していくと次のようになるでゲソ ? ここで見て欲しいのは簡約途中のネストの数でゲソ。β簡約していくと、末尾再帰じゃない方はネストが増えていくのに末尾再帰版は増えないじゃなイカ! これって TCO のことだとは考えられないでゲソ?

末尾再帰の場合:

```
(fact 3)
→ (loop 3 1)
→ (loop (- 3 1) (* 3 1))
→ (loop 2 3)
→ (loop (- 2 1) (* 2 3))
→ (loop 1 6)
→ 6
```

末尾再帰じゃない場合:

```
(fact 3)
→ (* 3 (fact 2))
→ (* 3 (* 2 (fact 1)))
→ (* 3 (* 2 1))
→ (* 3 2)
→ 6
```

というわけで、Scheme はアクターと Lisp にラムダ計算まで結びつけたんでゲソ。アクターと Lisp の関連性も想定外だったそうでゲソが、ラムダ計算との親和性についても当初は全く考えていなかったそうでゲソ (これに気づいた後でラムダ計算を考え出した Alonzo Church 先生の 20 年以上前の論文を読んだら、既に cons セルアクターみたいなものが書いてあって驚いたんだとか)。このことでラムダ計算自体の表現力の高さに気づいた 2 人は、それ以降 “Lambda Papers” と呼ばれる論文を発表してラムダ計算の凄さを布教して回ったんでゲソ。そしてこの過程でラムダ計算の強力さが関数型の世界で再認識されていったんでゲソよ。
今からだとなかなか想像できないでゲソが、Scheme 登場以前の関数型の世界では dynamic scope な Lisp が主流だったので純粋なラムダ計算からは少し遠ざかってたんでゲソ。それどころか「lexical scope だと高速な実装が作れないんじゃなイカ」みたいなことまで言われてたでゲソ。だけど、これをきっかけに原点であるラムダ計算を見直す動きが進んでいったんでゲソよ。ということは、もし Scheme がなかったら Haskell みたいな純粋関数型言語も誕生しなかったかもしれないでゲソ。みんなもっと Scheme の偉大さを称えるべきじゃなイカ?

4.4 そろそろ CPS についてひとこと言っておくでゲソ

さて「アクターモデルがメッセージパッシングだけで色々な制御構造を作れる」ということは「ラムダ計算でも末尾呼び出しだけでプログラムを記述すれば色々な制御構造が作れる」ということにならなイカ? [*29]実際、この「末尾呼び出しだけでプログラムを書く」というスタイルは非常に表現力が高いので、“Continuation-passing style”(継続渡しスタイル, CPS) と名前もついているほどでゲソ [*30]。

ところで、もしまだ「CPS の表現力が高い」というのが信じられない人は、こう考えればどうでゲソ? 末尾呼び出しって最適化されれば単なる goto になるでゲソ? ところで、goto と if(条件分岐) さえあればどんな制御構造だって作れるでゲソ? [*31]ということは、末尾呼び出しと if さえあればどんな制御構造だって作れるはずじゃなイカ! [*32]

参考までに、CPS で書いた例を 1 つ出しておくでゲソ。さっきのアクター版と比較しやすいように factorial の例でゲソ

(といっても Scheme には CPS 用の述語とか数学関数 (具体的には = とか - とか * とか) はデフォルトでは用意されてないので、さっきの例と比べるとちょっとアクター感 (CPS 感) が足りないかもしれないでゲソ。こういった述語や数学関数もさっきのアクター版では継続を受け取ってくれたんでゲソが、ここではしょうが無いので普通に関数として返り値をリターンさせて使ってるでゲソ)。

```
(define cps_factorial
  (lambda (n c)
    (if (= n 0)
        (c 1)
        (cps_factorial (- n 1)
                       (lambda (m)
                         (c (* m n)))))))
```

4.4.1 ここらで「継続」についてもう一度整理しておくでゲソ

今回の話の最初に、「継続」=「その後に行う計算処理」=「スレッドの状態」=(大抵のケースで)「スタックの中身」という説明をしたでゲソ。だけどさっきのアクターや CPS の話で出てきた「継続」は「スタックの中身」とはちょっと違うような気がしなイカ?

というわけで、もう一度「継続」について整理しておくでゲソ。

で、そもそもなんでスタックなんていう構造が必要なのかと言えば、リターンするためじゃなイカ? 普通の関数呼び出しはリターンしないといけないでゲソね。だから、リターン先を覚えておくためにスタックという LIFO (Last-in-First-out) な構造でリターンアドレスを管理してるでゲソ[*33]

[*29] というわけで、アクターパラダイムはラムダ計算上でのものすごく強力なデザインパターン、とも考えられるでゲソね。

[*30] この CPS に既に馴染みがあった人は、CPS って元から関数型の世界の概念だと思ってなかったでゲソ? 実際には、CPS は全然違うアクターパラダイムというパラダイムから輸入されてきた概念だったんでゲソね。といっても、今となっては「なじむ　実に！　なじむぞ」ってくらいラムダの世界に溶け込んじゃってるのでどっちでもいいでゲソが。

[*31] というか、(普通の CPU って大雑把に言えばその 2 つくらいしか制御構造がないから) それで作れなかったらコンピュータ上で動かないでゲソ。

[*32] 正確に言えば、さっきのアクター版 factorial でやっていたように、条件分岐も述語関数による継続の起動としてしまえば全部末尾呼び出しで説明することも可能でゲソ。

[*33] 余談でゲソが、“stack” という用語を作ったのは Edsger Dijkstra 先生 [4] で、そのときの動機は「再帰呼び出しを可能にするにはリターンアドレスを管理する必要があるから」というものだったんでゲソ。で、なんで再帰呼び出し限定かというと、再帰呼び出しがなければ各関数が予めリターンアドレス用の領域を 1 つずつ確保しておけばなんとかなるじゃなイカ。実際、再帰呼び出しがなかった昔の Fortran や COBOL はそういう実装になっていたらしいでゲソよ。

[*34]。

だけど、メッセージパッシングにはリターンという概念はなかったじゃなイカ。そして、そこから派生した CPS にもリターンという概念はないでゲソ。というわけで、メッセージパッシングや CPS はスタックを必要としないんでゲソ (というか、一方通行の制御だから LIFO な構造なんてあっても使い道がないというか…)。

さっき言ったとおり、(大抵のケースでは)「継続」=「スレッドの状態」=「スタックの中身」なんでゲソが、メッセージパッシングや CPS においては最後のスタックの中身って (ほとんどあるいは全く) スッカラカンでゲソ。代わりに、(スタックで管理するんじゃなくて) 自分で明示的にアクターやクロージャーを受け渡すことで管理してるんでゲソね。

というわけで、メッセージパッシングや CPS のような一方通行な制御では、「継続」=「その後に行う計算処理」=「スレッドの状態」=「そのスレッドに引数として渡されてきているアクター or クロージャー」ということになるんでゲソ。

4.4.2 ついでに、継続を捕捉する演算子 (call/cc) についても整理しておくでゲソ

上で説明したように、CPS はものすごく表現能力が高いから、プログラムを全部 CPS で書けば制御フローなんて思いのままでゲソ。

だけど、現実問題としてプログラム全体を CPS で書くのは結構大変だし、いろんな事情でどうしても CPS にできないことも多いでゲソ。

で、しょうがないから普通は一般的な関数スタイルでプログラムを書いてしまうでゲソよね? でもそうなると、制御フローの管理は「スタック構造で管理されるリターンアドレス」にお任せすることになってしまうでゲソ。そして、こうなってしまうと制御フローはスタック構造 (行ったら必ず帰ってこなきゃいけないという制約付きのフロー) にしばられるから CPS ほど柔軟には制御フローをコントロールできなくなってしまうでゲソ。

だけどそれだとなんだか悲しくなイカ? そこで、そういう場合でも CPS 並の強力な制御能力を使えるように、と導入されたのが継続を捕捉する演算子 call/cc なんでゲソ。つまり、call/cc というのは「(人手でプログラムを CPS 形式に直すのが難しいときに) プログラムを自動的に CPS に直して、継続に当たる部分を取り出してくれる演算子」と考えてもいいんでゲソ。

というわけで、そもそもプログラムを全部 CPS で書けるなら call/cc なんて要らないんでゲソよ。それに、関数的な「行ったら必ず返ってくる」フローで満足している場合にも必要ないんでゲソ。全部 CPS では書けないけど CPS ばりの制御力が欲しい、という時に使えるのが call/cc なんでゲソ[*35]。

4.4.3 ところで CPS って何かに似てなイカ?

ここで紹介した CPS には、この他にもいろいろと面白い性質があるでゲソ。例えばその一つに「完全に CPS で書くと評価順が関係なくなる」[*36] というものがあるんでゲソ。もうちょっと

[*34] こう言うと「局所変数を管理するという役割もあるじゃなイカ」と言われるかもしれないでゲソが、そもそも変数用のメモリ領域の管理とコントロールフローの管理を同期させなきゃいけない理由はないでゲソ? (Algol 以降のブロック構造を持つ手続き型言語にとっては、それが自然な実装だったのは確かでゲソが…)。というかむしろ関数型の言語では、クロージャとかに捕捉された自由変数はクロージャーが生き続ける限り永遠に生き続ける (格好よく言うと「無限のエクステントを持つ」) から、スタックと同期させちゃ駄目でゲソ。というわけで、ここではスタックの役割についてはリターンアドレスの管理だけを考えることにするでゲソ。

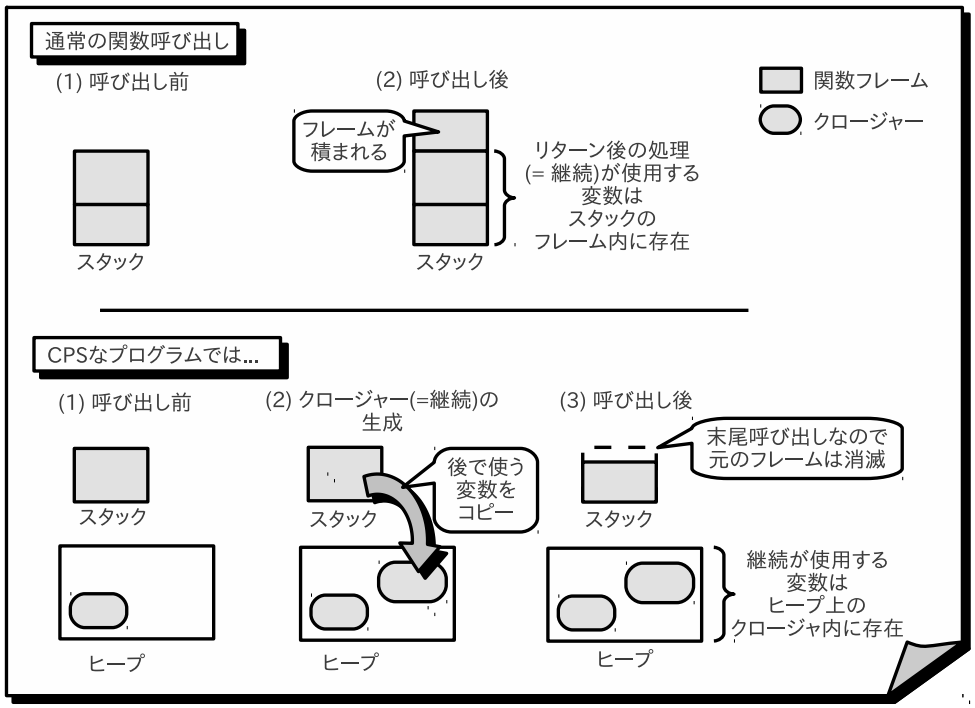
[*35] ちなみに、前の方の call/cc を説明していた箇所の注釈で、「継続と関数には関連性がある」とか「継続= 絶対にリターンしてこない関数」とか書いてあったのを覚えてるでゲソ? これは、要するに call/cc が捕捉する継続というのは「CPS で書かれたプログラムにおける末尾呼び出しされる関数」だからなんでゲソ。この関数は末尾呼び出しされるわけだから決してリターンしないでゲソ? だから、call/cc が捕捉した継続も決してリターンしないんでゲソ。

[*36] ただし、「完全に」と言うのがポイントでゲソ。上の cps_factorial みたいな (ちょっと中途半端な) CPS ではそうはならないでゲソ。

かみ砕いて言うと、(評価順というとセーキジュンジョ (Normal Order) とかテキヨージュンジョ (Applicative Order) とかいろいろあるでゲソが) どういう評価順の下でも常に同じ順番で評価が行われるようになるんでゲソ。これだけ聞くと不思議かもしれないでゲソが、そもそも評価順って「簡約できる箇所が複数あるときにどこから簡約するか」という話だったじゃなイカ? ところが完全に CPS で書くと、簡約できる箇所は常に 1 箇所 (式中の一番外側) しかなくなるんでゲソ。なぜかというと、もし式の内側に簡約できる箇所があったら、それって (関数呼び出しだと考えれば) その式の評価値をリターンしてくることにならなイカ? 完全に CPS で書いた場合にはリターンという概念はなくなるのでそんな式は存在しないんでゲソ。

ところで、こういった評価順に関する話は特に副作用が出てきたときに問題になるでゲソね。最近の関数型の世界では、評価順を解決する方法として「モナド」が有名でゲソ。だけど、評価順の問題であれば、この通り CPS でも解決できるでゲソ。というかもっと言うと、モナドと CPS には関連性があるんでゲソよ。モナドを使って書かれたプログラムと CPS で書かれたプログラムって見た目が似てないでゲソ? (CPS で継続として関数を連ねていく書き方と、モナドで bind 演算子による関数合成を連ねていく書き方、よくよく見るとそっくりじゃなイカ?)。実のところ、モナドは CPS を一般化したデザインパターン (あるいは CPS はモナドの特殊例 [*37]) とも考えられるんでゲソ [18] [6]。これは CPS とモナドの間にこういう類似性があるからでゲソね。

(ところで、この図は次の付録用のものでゲソ。気になった人は付録を見るでゲソ。)



[*37] というか Haskell の Continuation モナドがまさにそれでゲソね

付録 : CPS についての余談

さて「CPS はスタックを必要としない」と言ったんでゲソが、実際に CPS で書いたプログラムを動かすとどうなるのか気にならなイカ? 普通の処理系では関数呼び出しの際にはスタックフレームを積むことが多いので、CPS で書かれたプログラムでもスタックを全く使わないわけではないでゲソ。ただし、全部 CPS で書いた場合には (ちゃんと TCO する処理系なら) 全部の呼び出しが最適化されて goto になるわけでゲソ? だから、スタックフレームは常に 1 個しか存在しなくなるんでゲソ。というわけで CPS で書いたプログラムは、スタックは (使うには使うけど) ほとんど消費しない、ということになるでゲソ。
ところで、これについてもうちょっと考えると「CPS で書けばスタックフレームが積まれないからスタックオーバーフローも起こらない！ キタコレ！」みたいな気もしてこなイカ? だけどなんだか変でゲソね? CPS にするだけでそんなにいいことがあるなら、なんで皆しないんでゲソ?
これについての答えは上の図に書いたとおりでゲソが、スタックフレームを積まなくなる代わりに、その分ヒープを食うんでゲソ[a]。そもそも末尾呼び出し最適化で呼び出し時にスタックフレームを破棄しちゃって問題ないのは、その後の計算処理 (=継続) が必要とする値はヒープ上のクロージャーにコピーしているからでゲソ。
ところで、ここまでの内容を逆に考えると、call/cc みたいな「継続を捕捉する処理」には 2 通りの実装方法があるということにもなるでゲソ。1 つは (最初に紹介したように) 普通の関数スタイルで書いておいて継続を捕捉したくなった時点でスタックをコピーする方法、もう 1 つは予めプログラムを CPS 形式で書いておく方法でゲソ。もちろんどっちを使っても実現できるんでゲソが、スタックを効率的にコピー/復帰する手段が提供されていない Java VM のような処理系上では、プログラムを CPS 変換するアプローチの方が主流[b] でゲソ。例えば、Kilim [14] や Apache JavaFlow [1] というフレームワークでは、バイトコードレベルで Java のプログラムを CPS 変換することで継続捕捉操作を実現してるでゲソ。他にも、(最近人気が高い Scala なんかでも) Scala コンパイラがソースコードレベルで CPS 変換を掛けて継続[c]を取り出してたりするでゲソ [13]。

---

[a] 正確に言うと、クロージャーの実現方法にはいろいろあるから常にヒープを消費するとは限らないでゲソ。ただし、本質的に必要なデータ量が減らない限り、スタックにせよヒープにせよどこかを消費しないといけないのは間違いないでゲソ。というわけで、CPS にしたらメモリの使用量が減る、なんてのは絶対に嘘でゲソ。

[b] 主流と言っても、Java VM 上で継続を使うこと自体がものすごくマイナーな気もするでゲソが…

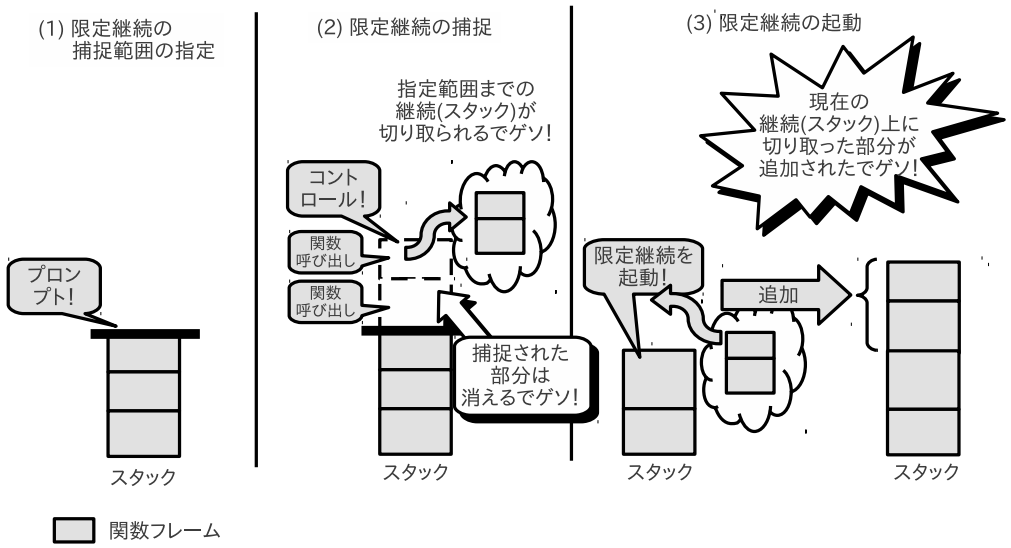
[c] 細かい話でゲソが、正確には「継続」じゃなくて「限定継続」なんでゲソ。限定継続については次の節で説明するでゲソ。

4.5 「ゲンテーケーゾク」って聞いたことはないでゲソ?

ところで、これまでは普通の「継続」について話をしてきたんでゲソが、最近は「限定継続 (Delimited Continuation)」という言葉の方がよく耳にするんじゃなイカ?「限定継続」というのは、ちょっと変わった新手の [*38] 継続でゲソ。ちなみに、限定継続という呼び名の他に、「部分継続 (Partial Continuation)」とか「合成可能継続 (Composable Continuation)」といった呼び名もあるんでゲソ。

限定継続が生まれたいきさつは後で説明するとして、まずどういうものかだけ簡単に説明してみるでゲソ。

継続は call/cc のような演算子で捕捉できたでゲソが、限定継続では二種類の演算子で捕捉を行うんでゲソ。限定継続にはいくつか流儀があってそれぞれに演算子の名前とかが違ったりするでゲソが、それも後で話すとして、とりあえずここでは control と prompt という演算子で説明してみるでゲソ。この演算子では、prompt で継続捕捉の終端点を指定し、control が prompt で指定された点までの継続を捕捉するんでゲソ。図にすると次のような感じでゲソ (図は引き続き「継続=スタック」という感じの例でお届けするでゲソ)。



こんな感じで、普通の継続と違って捕捉する範囲が control と prompt の間に限定されている (一部分しか捕捉しない) ので「限定継続 (部分継続)」と呼ばれるんでゲソ。あと、普通の継続だと呼び出したら決して返ってこないんでゲソが、部分継続は普通の関数のように呼び出し元にリターンしてくる、というのもポイントでゲソ (現在の継続を置き換えるんじゃなく、現在の継続の上に追加されるような感じになるんでゲソ)。このため、限定継続は他の処理と合成しやすい (composability がいい)、ということで「合成可能継続 (Composable Continuation)」とも呼ばれるんでゲソ。あと、(微妙な違いでゲソが) call/cc では捕捉した継続もそのまま残っていたのに限定継続では捕捉した分は現在の継続からは消える、というのも違う点でゲソ。

図だけだと具体的な動きが分かりにくいので、限定継続を使った例を 1 つ書いておくでゲソ。

[*38] といっても初めて登場したのは 1988 年なのでもうあんまり新手って感じじゃないでゲソが...

```
(+ 1 (prompt (+ 2 (control k (k (k 42))))))
```

さて、この場合に何が起こるか見ていこうじゃなイカ。

まず、control が捕捉するのは「prompt と control の間」でゲソ。だからこの例で捕捉される限定継続は下のような式でゲソ。

```
(+ 2 □)
```

あるいは、これは以下のような関数のことだと考えてもいいでゲソね。

```
(lambda (n)  (+ 2 n))
```

というわけで、上の式はこんな感じで評価されるんでゲソ。

```
(+ 1  (prompt (+ 2 (control k
    (k
      (k
         42))))))

→ (+ 1                      ; control によって、捕捉範囲の式が消えるでゲソ
     ((lambda (n) (+ 2 n))  ; 代わりに変数 k が捕捉した部分継続で置き換えられるでゲソ
      ((lambda (n) (+ 2 n)) ; (もう一つの k も同じく置き換えられるでゲソ)
        42)))

→ 47                        ; これが最終的な評価値でゲソ
```

さて、一応説明してみたんでゲソが、何となく雰囲気は分かったでゲソ? 初めて見るとすごく奇妙な動きに見えたりするんでゲソが、そういう時はコルーチンをイメージすると理解しやすいかもしれないでゲソ。

先にコルーチンを知らない人のために簡単に説明しておくと、コルーチンというのは一種の関数なんでゲソ。ただし、普通の関数と違って状態のようなものを持つんでゲソ。もうちょっと具体的に言うと、コルーチンは呼び出されると前回リターンしたところから実行が再開されるんでゲソ。

残念ながら Scheme とか Haskell にはデフォルトではコルーチンを作る機能が用意されてないので、とりあえず Lua での例を出してみるでゲソ。

```
coro = coroutine.create(
  function()
    coroutine.yield("イヤー！ ")     -- 1回目は "イヤー！ " を返すでゲソ
    coroutine.yield("グワー！ ")     -- 2回目には "グワー！ " を返すでゲソ
    coroutine.yield("アイエエエ！ ") -- 3回目には "アイエエエ！ " を返すでゲソ
  end)

print(coroutine.resume(coro))      -- 1回目の呼び出し
> true     イヤー！
print(coroutine.resume(coro))      -- 2回目の呼び出し
> true     グワー！
```

```
print(coroutine.resume(coro))      -- 3回目の呼び出し
> true    アイエエエ！
```

Lua では coroutine.create() という関数でコルーチンを作れるんでゲソ。コルーチン内でリターンしたい時には coroutine.yield() を使えばいいんでゲソ。また、(関数呼び出しならぬ) コルーチン呼び出しをしたいときには coroutine.resume() でゲソ。この例の通り、作ったコルーチンは呼び出す度に前回 yield した箇所から実行が再開されるんでゲソ (ちなみに true と出てるのは coroutine.resume() が呼び出しの成否を示すための真偽値も一緒に返すからでゲソ)。

で、これと限定継続の関係なんでゲソが、限定継続の方も「prompt で囲われた範囲」を 1 つの関数と考えると、コルーチンの動きと少し似ているんでゲソ。コルーチンの挙動は「yield すると関数からリターン、次に呼び出されると前回の yield から実行再開」だったでゲソね。限定継続の場合は「control すると prompt で囲われた範囲からリターン、control で得られた限定継続を呼び出すと control した所から実行再開」なんでゲソ。

```
(define coro-modoki
  (prompt
    (string-append "「コワイ!」"       ; この文字列連結処理が
                                       ;   「残りの処理 (限定継続)」になるでゲソ。
      (control k                       ; 最初は、残りの処理 (限定継続 k) と
        (cons k                        ;   "「アイエエエ!?」..." を返すでゲソ
          "「アイエエエ!?」「ケイゾク!?ケイゾクナンデ!?」")))))

(cdr coro-modoki)                      ; とりあえず最初の結果を
> 「アイエエエ!?」「ケイゾク!?ケイゾクナンデ!?」   ;   見てみるでゲソ

((car coro-modoki) "「ゴボボーッ!」")          ; 引き続き、残りの処理も実行して
> 「コワイ!」「ゴボボーッ!」                   ;   結果を見てみるでゲソ
```

ただし、コルーチンの場合は (勝手に状態が更新されていくので) 元の状態に戻したりすることは出来ないでゲソが、限定継続の場合は (control で捕捉した限定継続をどう使うかはユーザーに任されているので) 限定継続を使わずにもう一度最初から実行させることも出来るし、あるいは同じ限定継続を何度も実行することも出来るでゲソ。というわけで、制御構造としては限定継続の方が柔軟でゲソね (実際、限定継続さえあればコルーチンを作るのなんて楽勝でゲソ)。

4.6 どうして限定継続が考え出されたのか気にならなイカ?

以上で限定継続について一通りの説明は終わったんでゲソが、そもそもなんでこんなのが考え出されたのか不思議じゃなイカ? ここまでの説明だけだと (INTERCAL の COMEFROM みたいな) キワモノ制御演算子の一種に見えるかもしれないので、少し誕生の経緯も説明しておくでゲソ。

ところで、上の継続の説明で「継続=スレッドの状態」という話を見た時に、名状しがたい冒涜的な気持ちになった人はいないでゲソ? 確かに「状態」なんて言葉を聞くと、ついつい参照透明性のことが心配になってしまうでゲソね。

で、実は嫌な予感の通り、call/cc のような「継続を捕捉する演算子」は参照透明性を持たないんでゲソ[*39]。意外に思ったかもしれないでゲソが[*40]、次のように考えてみれば納得できるんじゃなイカ?

・参照透明性
「同じ関数に同じ引数が適用されれば、(文脈や状態といったそれ以外のものに関わらず) 常に同じ値が返る性質」
・継続を捕捉する演算子
「その演算子が呼び出された時点での継続 (≒スレッドの状態) を取り出す演算子」

call/cc が取り出す「呼び出された時点でのスレッドの状態」は、どう考えても参照透明性が要求する「文脈や状態に依存しない値」とは思えないでゲソ。実際、call/cc を使って次のような関数を作ってみると簡単に参照透明性を壊せるでゲソ。

```
(call/cc (lambda (k) k)) ; call/cc の引数が継続をそのまま返すだけなので、
                         ;    この式を評価すると「その時点での継続」になるでゲソ
```

次の 2 つのプログラムを考えてみるでゲソ。

```
(1) (let ((a (call/cc (lambda (k) k)))
          (b (call/cc (lambda (k) k))))
      (cond
        ((eq? a 0) #t)
        ((eq? b 0) #f)
        (else      (a 0))))   ; <= 違いはここだけ (こっちは a を使用)

(2) (let ((a (call/cc (lambda (k) k)))
          (b (call/cc (lambda (k) k))))
      (cond
        ((eq? a 0) #t)
        ((eq? b 0) #f)
        (else      (b 0))))   ; <= 違いはここだけ (こっちは b を使用)
```

[*39] この話については、川合史郎さんの次のページに詳しい話が載っているのでお勧めでゲソ。
http://practical-scheme.net/wiliki/wiliki.cgi?Scheme%3Acall%2Fcc%e3%81%a8%e5%89%af%e4%bd%9c%e7%94%a8

[*40] 例えば、Haskell の Continuation モナドを知ってる人は、参照透明な Haskell にも callCC があるじゃなイカって思ったかもしれないでゲソ。だけど、もう一度その callCC をよく見て欲しいでゲソ。Haskell の callCC は「継続を捕捉」したりはしないので参照透明なんでゲソ。

この (1) と (2)、ほとんど同じプログラムでゲソ。違いはたった一カ所で、最後の行で a に 0 を適用するか b に 0 を適用するかだけなんでゲソ。しかも a と b はどちらも (call/cc (lambda (k) k)) という式の評価値じゃなイカ。ということは、(call/cc (lambda (k) k)) が参照透明性を持つなら、この a と b を入れ替えただけのプログラムは同じ評価値を返すはずでゲソ? ところが実際にやってみると (1) は #t になり (2) は #f になるでゲソ[*41][*42]

とまぁこんな問題があったんでゲソが、1980 年代の末になるまでは call/cc のような「継続を捕捉する演算子」はあくまでプログラミング言語の機能としてしか考えられていなかったんでゲソ。逆に言うと、元々のラムダ計算の世界でどうなるのかは考えられていなかったんでゲソね。そのせいで、それを取り入れた計算体系の性質 (チャーチロッサーセーとかサンショートーメーセーとか) はよく分かってなかったんでゲソ。そこで、継続捕捉操作で拡張したラムダ計算を作るという研究 [5] が行われたんでゲソが、これが限定継続の始まりなんでゲソ。

ところで、継続が状態だと考えると参照透明な世界で表現するのは難しそうじゃなイカ? だけど、こういう一見すると参照透明じゃないものを参照透明な世界で扱う方法があったでゲソね。そう、「モナド」でゲソ。限定継続のアプローチは少しモナドに似てるから、「モナド化することで参照透明化した継続捕捉操作」とも考えられるでゲソ。とはいえ、元々の論文のアイデアは全然モナドじゃない (というかモナドには一言も触れてない) ので、今回はあんまりモナドにこだわらずに説明してみるでゲソ。

では、元々の論文のアイデアを簡単に説明してみるでゲソ。まず、元々の論文ではラムダ計算に対して次のような F という演算子[*43]を追加しようとしてたんでゲソ。

`F M` ; ただし、`M` は「引数として継続を受け取るラムダ式」に簡約される式

この F 演算子、ほとんど call/cc とそっくりなんでゲソが、捕捉された継続が消えてなくなるというのが call/cc との違いでゲソ。(図は引き続き「継続=スタック」の例で説明するでゲソ)

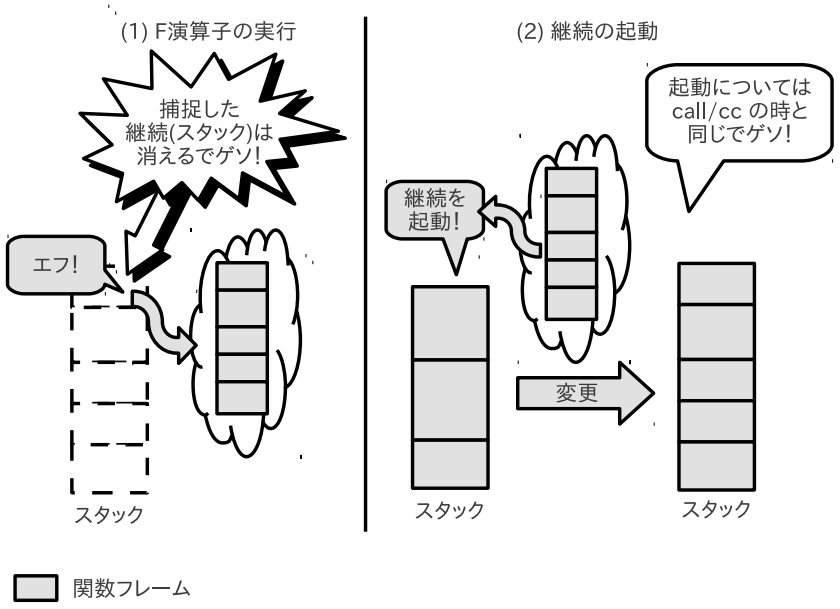
まぁあんまり大きな違いじゃないでゲソが一応どういうことかを説明すると、(引数である M の中で捕捉した継続が起動されなかった場合) 継続が全部なくなっているので M の返値がそのままプログラム全体の評価値になるんでゲソ。言い換えれば、M の返値を持って一気にトップレベルまで大域脱出してしまう感じになるんでゲソ[*44]。

[*41] ところで、この話ちょっと紛らわしいので注意が必要でゲソ。ここで言っているのは「継続自体に参照透明性がない」という話ではないでゲソ。実際、継続自体は単に末尾位置で呼び出されるだけの関数だったりするわけで、(その中で明示的に副作用とかを起こさない限り) 参照透明でゲソ。だから、CPS でプログラムを書いたりするのは全く問題ないんでゲソ。ここで問題になってるのは、「継続」自体じゃなくて、「継続を捕捉する演算子」でゲソ。

[*42] (ついでに、上で紹介した川合史郎さんのページにも書いてある話でゲソが、念のために紹介しておくと) 上で書いたように call/cc って単に「プログラムを CPS の形式に変換してくれる」演算子なんでゲソ。だから、プログラムを手動で CPS 変換すれば call/cc なんて必要ないでゲソ。ところでこのとき、call/cc を使った方は参照透明じゃなくて、同じ計算処理をするのに CPS に直した方は参照透明性があるということになるんでゲソが、ちょっと不思議じゃなイカ? でもこれは次のように考えればたぶん納得できるでゲソ。プログラムを CPS に直すのはプログラム全体を書き直さないといけない大域的な変更でゲソ。実際、CPS で書けば call/cc なんて要らないのにわざわざ call/cc が用意されてるのは、CPS に直す作業がものすごく影響範囲が広い大変な作業だからでゲソ [16]。で、そんな大域的な変更をしていいなら、どんなに参照透明性のないものだって参照透明性のある形に直せるじゃなイカ (例えば I/O だって I/O モナドを使った参照透明な式に出来ちゃうでゲソ?)。だから、参照透明性の無いものが (大域的な変更を通して) 参照透明性のあるものと対応するのは不思議じゃないんでゲソ。

[*43] この演算子は Felleisen's F operator とか Felleisen's C operator とか呼ばれることが多いようでゲソ。

[*44] control とかが捕捉した継続を消し去るという挙動もこれに由来してるでゲソ。ちなみになんでこうなったかというと、数学的にはこっちの方がよさげだと考えたから、らしいでゲソ。確かに本当に CPS でプログラムを書いた場合は受け取った継続を使うも使わないも自由だから「現在の継続を使わない」という選択肢がある方が自然かもしれないでゲソ。



なので例えば call/cc とはこんな感じで違いが出るでゲソ (本当は F はラムダ計算に追加する演算子でゲソが、call/cc との比較のために scheme っぽく書いてみるでゲソ)。なお、書くのがちょっと面倒だから「捕捉された継続」の詳細は書かずに●で済ませてみるでゲソ。

```
・ F の場合:
   (+ (F (lambda (k) 42)) 1)
→ ((lambda (k) 42) ●) ; 継続である (+ □ 1) の部分は継続捕捉によって消えるでゲソ
→ 42                  ; 継続が無くなってるので、42 が全体の評価値になるでゲソ

・ call/cc の場合:
   (+ (call/cc (lambda (k) 42)) 1)
→ (+ ((lambda (k) 42) ●) 1)  ; 継続である (+ □ 1) は残ったままでゲソ
→ (+ 42 1)                   ; 継続が残っているので、まだ残りの処理があるでゲソ
→ 43                         ; これが最終的な評価値でゲソ
```

もちろん明示的に継続を起動すれば消えちゃった継続は元に戻せるので、大域脱出したくない場合でも心配ないでゲソ。例えば、call/cc 的に使いたい場合はイカのようにすればいいだけでゲソ。

```
(call/cc M) =                    ; (M k) を計算した後、
    (F (lambda (k) (k (M k)))) ;   k を起動して継続を元に戻せば call/cc 相当でゲソ
```

で、この F 演算子なんでゲソが、単純に導入するとやっぱり参照透明性はないんでゲソ。例えば次の 2 つの式を考えてみるでゲソ (上でも書いた通り F は本当はラムダ計算に追加する演算子でゲソが、引き続きちょっと Scheme っぽく書くでゲソ)。

```
(a) (F (lambda (k) 0))
    → ((lambda (k) 0) ●) ; F による継続捕捉
    → 0
```

```
(b) 0
    → 0
```

この2つの式、見ての通りどちらも0に評価されるでゲソ。

さて、評価値が同じなんだから、参照透明性があればこの2つの式は自由に入れ替えても問題ないでゲソ? ところがやっぱりそうはならないんでゲソ。次の式を考えてみるでゲソ。

```
(a') (exp (F (lambda (k) 0)))
     → ((lambda (k) 0) ●) ; F による継続捕捉
     → 0
(b') (exp 0)
     → 1
```

違ってるのは (F (lambda (k) 0)) と 0 だけなのに、(a') 式の評価値は 0 で (b') は 1 じゃなイカ。というわけで、明らかに参照透明性がないでゲソ!

それじゃいよいよ、限定継続を入れるとこれがどう解決されるかを説明するでゲソ。最初にアイデアを大雑把に言っておくと、解決策というのは次のようなものだったんでゲソ。

(1) F演算子は、42とか0じゃなく、特殊な値 (ここでは「限定継続値」と呼ぶでゲソ) を返すんでゲソ。
(2) この「限定継続値」は、この値を処理しようとした式を自分の中に飲み込んでいくというブラックホールみたいな代物でゲソ。
(3) ただし、この「限定継続値」に飲み込まれない演算子も1つ追加するんでゲソ。その演算子は、限定継続値に飲み込まれた式を取り出して評価を行うんでゲソ。

じゃあアイデアを順に説明していくでゲソ。ただしこの「限定継続値」については、Schemeの式の中で区別が付きやすいように $(...) とでも書くことにするでゲソ。

さてまず (1) でゲソが、どういうことなのか具体的に説明すると、「(F (lambda (k) 0)) という式は0じゃなくて $(lambda (k) 0) という値に評価される」、ということでゲソ[*45]。

つまりさっきの (a') 式は、一段だけ評価を進めると (exp $(lambda (k) 0)) になるんでゲソ。これなら (b') 式の (exp 0) と評価結果が違っても問題ないでゲソね。参照透明性は「同じ関数に同じ引数が適用されたら同じ値になる」という性質だったじゃなイカ。exp関数の引数が違うなら評価結果が違っても参照透明性違反じゃないでゲソ。

でもそうすると今度は、指数関数の評価の方で困ってしまわなイカ? 指数関数は算術用の関数だから数値は引数に取れても限定継続値なんて定義域に入ってないでゲソ。それに、そもそもF演算子が作る部分継続は、この指数関数を継続として捕捉しないといけないんでゲソ。だから指数の計算が行われちゃっても困るでゲソ。さて、どうするでゲソ?

ここで出てくるのが (2) でゲソ。まず、指数関数が引数として限定継続値を取れないなら、取れるように定義域を拡張してしまえばいいだけの話でゲソ。そして引数として限定継続値が与えられた場合には、(指数を計算する代わりに) その限定継続値に「指数を計算するという継続」を足し込んだ新しい限定継続値を返す、ということにすればいいんでゲソ。つまりこんな感じでゲソ。

```
(exp n)     = e の n 乗の値       ; ただし、これは引数が数値の場合でゲソ
(exp $(M)) = $(lambda (k)        ; 限定継続値だったら、新しい限定継続値を返すでゲソ
                 (M
```

[*45] ところで、モナドが好きな人は「F演算子はモナドを作る演算子 (return)」なんだと思ってくれればいいでゲソ。

```
                (lambda (n) (k (exp n)))) ; この部分が exp を計算する継続でゲソ
```

どうでゲソ? *46 ちょっと読みにくいでゲソが、まるで exp 関数が限定継続値の中に飲み込まれたみたいに見えなイカ? *47 というわけで、「F 演算子が指数関数を継続として捕捉する」という挙動が見事に実現できてるでゲソ!

もちろん、指数関数以外の関数についても同じように拡張を行うんでゲソ。これで、限定継続が周りの全ての計算処理を継続として取り込めるようになるでゲソ。*48

さて、これで参照透明性の問題は無くなったし、F 演算子が周りの計算を継続として取り込むという挙動も実現できたでゲソ。というわけで問題は全部解決したように思えるでゲソ?

ところが、まだ話は終わってないんでゲソ。というのも、限定継続値なんていう変な値だけ作れても困るでゲソ! そもそも、今回の (a') の例では最終的な評価値として 0 が出てきて欲しかったんでゲソ。今のままだとどこまで行っても限定継続値しか出てこないから 0 にならないでゲソ。

何が問題かというと (2) のルールだけだと限定継続値の中に継続を構築できるというだけで、実際にそれを使うことが出来ないんでゲソ。つまり、限定継続値の中に構築された継続を取り出す方法が必要なんでゲソ。というわけで (3) のルール、もっと具体的に言うと prompt 演算子の出番でゲソ *49。限定継続で二種類の演算子を使うのはこういう訳なんでゲソ。

で、prompt 演算子の挙動を説明するとこんな感じでゲソ。ちなみに、ここでは元の論文と同じく prompt 演算子を # 記号で表してみたでゲソ *50。

```
(# $(M))  = (M (lambda (x) x)) ; 限定継続値の中身 M を取り出し、
                               ;   さらに恒等関数を適用することで M を実行するゲソ
(# v)     = v                  ; ちなみに、引数が限定継続値じゃない場合は
                               ;   単にその値を返すだけでゲソ
```

じゃあ、この規則を入れたら (a') 式と (b') 式がどうなるかをもう一度考えてみるでゲソ。ただし、前のままだと (a') 式の方が限定継続値にしかならないので、どちらも prompt 演算子を付けて考えてみるでゲソ。

```
(a'') (# (exp (F (lambda (k) 0))))
      → (# (exp $(lambda (k) 0))) ; F 演算子が限定継続値を作り出すでゲソ
      → (# $(lambda (k)           ; exp 関数が限定継続値に飲み込まれるでゲソ
              ((lambda (k) 0)
                (lambda (n) (k (exp n))))))
```

*46 えっ、関数の定義域や値域が恣意的で気持ちが悪い? うーん、でも例えば exp 関数って「無限ループする式」を引数に受け取ったら自分も停止しないでゲソ? それって exp(⊥) = ⊥ ということじゃなイカ。上の定義が気持ち悪いなら、これだって同じくらい気持ち悪くないでゲソ?

*47 正確には、限定継続値が飲み込むと言うよりは、指数関数が自発的に限定継続値の中に入っていくような感じでゲソが…

*48 ところでモナドが好きな人は、(2) は (Haskell の do 構文とか Scala の for 構文みたいな感じで) モナドの bind 演算子とかが暗黙的に存在しているんだと考えてもらえばいいでゲソ。つまり、上の (a') 式は (return (\k -> 0)) >>= (\m -> return (\k -> k (m (\n -> exp(n))))) のシンタックスシュガーみたいなものでゲソ。

*49 モナドが好きな人なら「モナドからその中身を取り出す演算子」だと思ってくれればいいでゲソ。ところで、限定継続が状態みたいなものなら、モナドの中身を取り出したりしたら危険な気もしなイカ? だけど限定継続が表してるのは、状態は状態でも「prompt 内だけの局所的な状態」なんでゲソ。同じように局所的な状態を表してる Haskell の State モナドなんかも中身を取り出せるでゲソ? だから限定継続も中身を取り出して問題ないんでゲソ。

*50 余談でゲソが、元論文では (Scheme っぽく書いたりはしてなかったので) prompt 式の見た目は "# ..." という感じだったんでゲソ。で、この # 記号が先頭に来る見た目が「なんだかコマンドラインのプロンプトっぽい」というので "prompt" という名前が付いたんでゲソ。ところでプロンプトが # ってことは普段 root で作業してたってことでゲソ?

```
      → (# $(lambda (k) 0))          ; ちょっと限定継続値の中の式を整理しておくでゲソ
      → ((lambda (k) 0)              ; # 演算子が限定継続値の中身を取り出すでゲソ
          (lambda (x) x))
      → 0                            ; 見事 0 になったでゲソ!
(b'') (# (exp 0))
      → (# 1)
      → 1                            ; こちらも見事に 1 になったでゲソ!
```

うむ、見事に説明が付いたじゃなイカ!

じゃあ最後に、(この例だけだと捕捉した継続を使ってなくてつまらないので) 別の例も出しておくでゲソ。さっきの例に似てるでゲソが今度は捕捉した継続をちゃんと使う例でゲソよ。ちなみに今回捕捉する継続は (+ 1 (exp □)) といった式なので、これを 0 に適用して、評価値は (+ 1 (exp 0)) = 2 になるはずでゲソ。

(ところで + 関数がどういう風に限定継続値に飲み込まれるかは説明してないでゲソが、まぁなんとなく分かるでゲソ?)

```
(c)   (# (+ 1
           (exp (F (lambda (k) (k 0))))))
      → (# (+ 1                      ; F 演算子が限定継続値を作り出すでゲソ
           (exp $(lambda (k) (k 0)))))
      → (# (+ 1                      ; exp 関数が限定継続値に飲み込まれるでゲソ
           $(lambda (k)
              ((lambda (k) (k 0))
                (lambda (n)  (k (exp n)))))))
      → (# (+ 1                      ; ちょっと限定継続値の中の式を整理しておくでゲソ
           $(lambda (k) (k (exp 0)))))
      → (# $(lambda (k)              ; + 関数が限定継続値に飲み込まれるでゲソ
              ((lambda (k) (k (exp 0)))
               (lambda (n) (k (+ 1 n))))))
      → (# $(lambda (k)              ; ちょっと限定継続値の中の式を整理しておくでゲソ
              (k (+ 1 (exp 0)))))
      → ((lambda (k)                 ; # 演算子が限定継続値の中身を取り出すでゲソ
           (k (+ 1 (exp 0))))
         (lambda (x) x))
      → ((lambda (x) x)
           (+ 1 (exp 0)))
      → 2                            ; うむ、バッチリじゃなイカ!
```

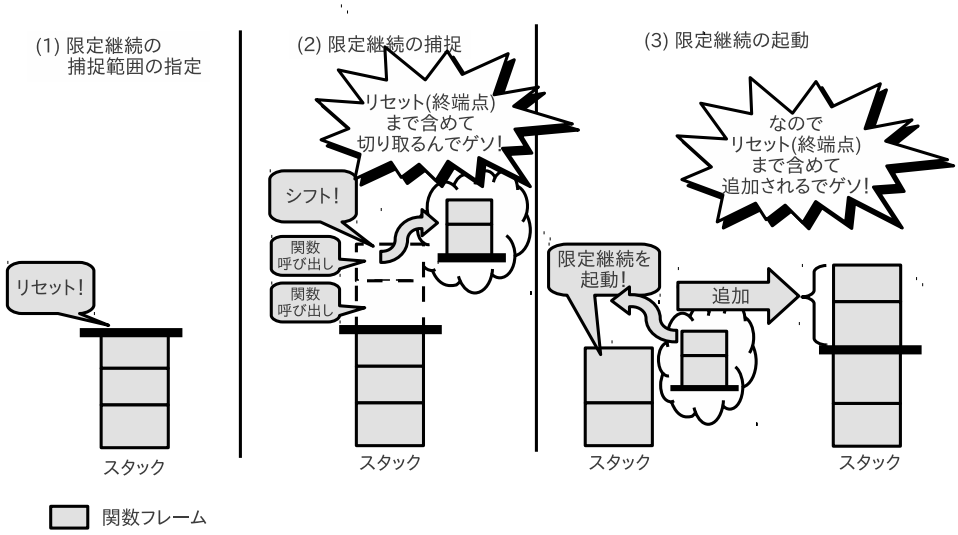
どうだったでゲソ? これでもう参照透明性の心配をせずに思う存分継続を捕捉できるでゲソね。

4.7 他の種類の限定継続も紹介しておくでゲソ

さっき紹介した control/prompt がオリジナルの限定継続の流儀なんでゲソが、その後で色々な限定継続が発見されてきたんでゲソ。その中でも一番有名なのが shift/reset と呼ばれる流儀でゲソ (ちなみに、呼び名の通り shift と reset という演算子を使うんでゲソ)。

また新しい演算子が出てきて面倒だと思ったかもしれないでゲソが、実は control/prompt と

shift/reset の違いは 1 箇所だけなんでゲソ。下にあるのが shift/reset の動作を表した図でゲソが、control/prompt によく似てなイカ?



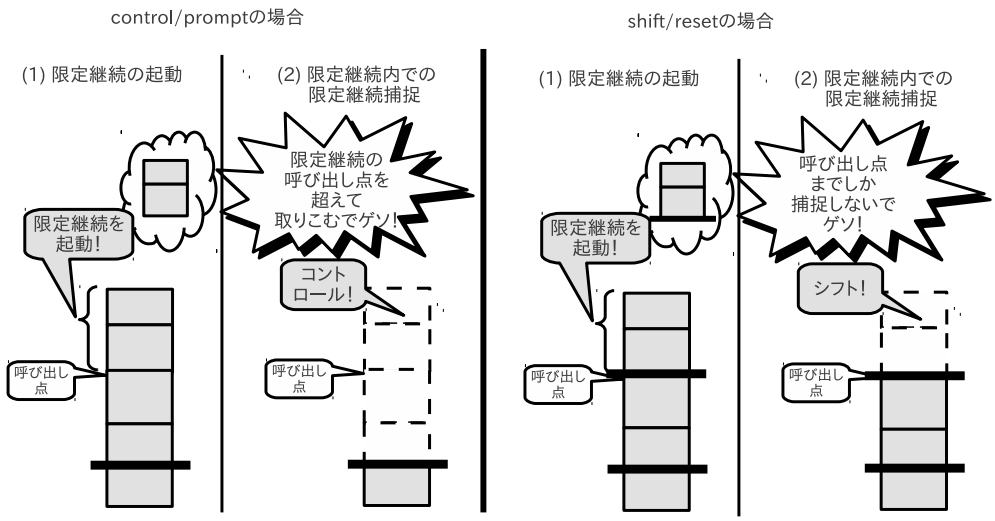
この 2 つの何が違うかというと、control 演算子では限定継続を捕捉する時に prompt 自体は取り込まなかったでゲソ? ところが、shift 演算子が限定継続を捕捉する時には reset まで含めて取り込むんでゲソ。

これでどういう違いが出るかというと、(捕捉した限定継続を呼び出した際に) その中でもう一度限定継続を捕捉すると違いが出るんでゲソ。次のページに図を描いたので、それも見ながら読んで欲しいでゲソ。

control/prompt の場合は、捕捉した限定継続自体には prompt が含まれないでゲソ? ということは、限定継続内で限定継続の捕捉を行うと (その限定継続内に他に prompt がなければ) 限定継続の呼び出し元の継続まで捕捉範囲に入るんでゲソ。

逆に shift/reset の場合は shift で捕捉した限定継続に reset が含まれるでゲソ? となると、起動した限定継続内で限定継続の捕捉を行っても、(仮に限定継続内に reset がなかったとしても) 最悪、限定継続の呼び出し元の地点で捕捉が止まるんでゲソ。

確かに、コルーチンとかを作る場合を考えると、コルーチンの呼び出し点で止まってくれた方がいいかもしれないでゲソね。限定継続を捕捉する操作は (その分の継続が消えちゃうから) 大域脱出を伴うことがあるでゲソ? コルーチン内で yield したらコルーチンの呼び出し元よりも前に大域脱出しちゃったりすると、ちょっと使いにくそうでゲソ。



とはいえ、これは control/prompt と shift/reset の戦力の決定的差ではないんでゲソ。というのも、control/prompt でこの挙動をマネするのは簡単だからでゲソ。実際問題として、control/prompt があれば shift/reset は簡単に作れるんでゲソ。

(ちなみに、上の説明で何となく気づいたかもしれないでゲソが、prompt と reset は全く同じものでゲソ。違いがあるのは control と shift だけなので、control を使った shift の作り方だけを書いてみるでゲソ。)

```
(shift k M) =
  (control _k                          ; 限定継続 _k をそのまま渡す代わりに
    (M                                 ;   「prompt をセットしてから _k を呼び出す」
     (lambda (x) (prompt (_k x))))) ;   という関数を渡せば、shift 相当
```

一方、shift/reset から control/prompt は作れるか？ というと、 Scheme のような形の無い世界なら作れることが分かっているでゲソ。一方、型のある世界では作れないことも分かっているでゲソ[*51]。

え、それじゃ shift/reset より control/prompt の方がいいんじゃなイカって? 確かに表現力だけ見ると control/prompt の方が良さそうでゲソ。だけど、shift/reset の良さは別の所にあるんでゲソ。

4.7.1 何で shift/reset の方がよく使われるんでゲソ?

shift/reset の良さは CPS (Continuation-passing style) と関係があるんでゲソ。

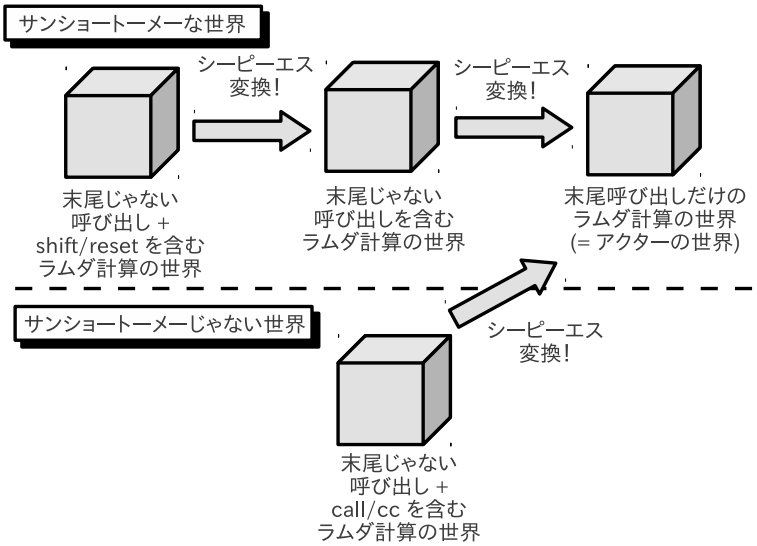
少し思い出して欲しいんでゲソが、限定継続って普通の関数のようにリターンしてくるじゃなイカ? で、「リターンしてくる継続」ってどういうことでゲソ?

(限定じゃない) 普通の継続は、前に説明したように「CPS で書かれたプログラムにおける末尾呼び出しされる関数」に対応してたんでゲソ。で、末尾呼び出しだから決して返ってこなかったんでゲソ。じゃあ逆に考えると、「CPS で書かれたプログラムにおける末尾じゃないところで呼び出される関数」って限定継続ということにならなイカ?

というわけで「CPS 変換しても末尾呼び出しにならない制御構造」を研究していたところ、見つ

[*51] 本質的に control/prompt の方が表現力が高いんでゲソ。なにせ control/prompt では「型がつくけど停止しない式」まで作れてしまうんでゲソ (カッコよく言うなら「キョーセーキカセー (強正規化性) がない」ということでゲソ) [8]

かったのが shift/reset だったんでゲソ [3] [*52]。



で、こんな風に CPS 変換した世界に対応物があると色々と便利そうじゃなイカ? 例えば、control/prompt とか shift/reset の型を考えようと思ったとき、そのままだとよく分からないでゲソ? でも CPS 変換で普通の関数に対応すると分かっていれば、その関数の型を手がかりに研究が始められるというわけでゲソ。それに実際に使う上でも、CPS な世界に対応物があれば (実行時にスタックをコピーする代わりに) コンパイル時に CPS 変換することで継続捕捉処理を実装できる、というメリットもあるでゲソ。

4.8 おわりに

さて、今回は「継続」について色々と話してきたでゲソが、どうだったでゲソ?

継続は強力な反面けっこう癖も強いので、普段はなかなか使いどころがないかもしれないでゲソ (正直、私もそんなに上手く使いこなせているとは言えないでゲソ…)。でも、ここぞという場面で上手く使えると (きっと) カッコいい (にちがいない) でゲソよ!

最近は、軽量スレッドがブームになったり Scala に限定継続が実装されたりで、継続が活用できる場面も増えてきているような気がするでゲソ。もしかしたら、来年あたりには継続を使った素敵な関数型のフレームワークが流行っちゃったりするんじゃなイカ (妄想)。というわけで、今から素敵な継続使いを目指して頑張ろうじゃなイカ!

[*52] ところで「CPS なのに末尾呼び出しじゃない」ってなんだか自己矛盾しているようにも聞こえるでゲソね。というわけで少し整理しておくと、CPS は (スタックに積まれるリターンアドレスじゃなくて) 明示的に受け渡すクロージャーで制御フローを管理するやり方でゲソ? 逆に、末尾じゃない関数呼び出しはスタックに積まれるリターンアドレスで制御フローを管理するんでゲソね。じゃあ「CPS だけど末尾じゃない関数呼び出しを含むプログラム」はと言うと、「明示的に受け渡すクロージャーとスタックに積まれるリターンアドレスの 2 つで継続を管理する」ということなんでゲソ。つまり「2 種類の継続を用意しておいて使い分ける」というのが shift/reset の本質でゲソ。というわけで、明示的に 2 種類の継続をクロージャーとして受け渡すようにすれば、同じことを全部末尾呼び出しだけで書くこともできるでゲソ。それで、明示的に管理するのを 1 種類だけにしたいなら「CPS+ 末尾じゃない呼び出し」にすればいいし、両方とも明示的に管理したくないなら shift/reset を使えばいいだけでゲソ。

参考文献

[1] JavaFlow: The Apache Software Foundation. `http://jakarta.apache.org/commons/sandbox/javaflow`.

[2] ABELSON, H., DYBVIG, R., HAYNES, C., ROZAS, G., ADAMS, N., FRIEDMAN, D., KOHLBECKER, E., STEELE, G., BARTLEY, D., HALSTEAD, R., OXLEY, D., SUSSMAN, G., BROOKS, G., HANSON, C., PITMAN, K., AND WAND, M. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11* (1998), 7–105. 10.1023/A:1010051815785.

[3] DANVY, O., AND FILINSKI, A. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming* (New York, NY, USA, 1990), LFP '90, ACM, pp. 151–160.

[4] DIJKSTRA, E. W. Recursive programming. *Numerische Mathematik 2* (1960), 312–318.

[5] FELLEISEN, M. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1988), POPL '88, ACM, pp. 180–190.

[6] FILINSKI, A. Representing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1994), POPL '94, ACM, pp. 446–457.

[7] HEWITT, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence 8*, 3 (June 1977), 323–364.

[8] KAMEYAMA, Y., AND YONEZAWA, T. Typed dynamic control operators for delimited continuations. In *Proceedings of the 9th international conference on Functional and logic programming* (Berlin, Heidelberg, 2008), FLOPS'08, Springer-Verlag, pp. 239–254.

[9] KISELYOV, O. IO monad realized in 1965. `http://okmij.org/ftp/Computation/IO-monad-history.html`, 2012.

[10] LANDIN, P. J. Correspondence between algol 60 and church's lambda-notation: part i. *Commun. ACM 8*, 2 (Feb. 1965), 89–101.

[11] LANDIN, P. J. A generalization of jumps and labels. *Higher Order Symbol. Comput. 11*, 2 (Sept. 1998), 125–143.

[12] REYNOLDS, J. C. The discoveries of continuations. *Lisp Symb. Comput. 6*, 3-4 (Nov. 1993), 233–248.

[13] ROMPF, T., MAIER, I., AND ODERSKY, M. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *SIGPLAN Not. 44*, 9 (Aug. 2009), 317–328.

[14] SRINIVASAN, S., AND MYCROFT, A. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming* (Berlin, Heidelberg, 2008), ECOOP '08, Springer-Verlag, pp. 104–128.

[15] STEELE, G. The history of scheme. In *JAOO Tenth Annual Conference* (Oct. 2006).

[16] STEELE, G. L., AND SUSSMAN, G. J. Lambda: The ultimate imperative. Tech. rep., Cambridge, MA, USA, 1976.

[17] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling fulljumps. *Higher Order Symbol. Comput. 13*, 1-2 (Apr. 2000), 135–152.

[18] WADLER, P. The essence of functional programming. Prentice Hall, pp. 1–14.

第5章

愚者のコントロール

— @tanakh

5.1 I

「わたし、気になります！」

やれやれ、また始まった。「やらなくていいことなら、やらない。やらなければいけないことなら、ギリギリまでやらない」がモットーの俺にとって、これは悪い予兆だ。子供のような無邪気な眼差しで四路戸えるに見つめられると、そんな信条とは裏腹に、どういう訳だかいつも面倒事に巻き込まれてしまう。

「私気になりません」

僅かばかりの抵抗を試みるが、最後には今回もこの眼差しに絡め取られてしまうのだろう。まったく厄介な役回りを引き受けてしまったものだ。

「いつもの遅延評価かい、奉太郎。そんなに無碍にしなくてもいいじゃないか。この謎、僕も気になるしさ」

複文里志はいかにも楽しそうだ。やめてくれ、俺はお前とは違う。ただ平穏な学校生活を送れればそれでいい。取り柄も何もない、普通の高校生なんだ。

「おはよう。……どうしたの、折具？」

部屋に入るなり川中摩耶花はそう言った。憂鬱な気分が入り口からも分かるほど顔に張り付いているのだろうか。だが、いよいよもって逃げられそうになくなってきた。このままはぐらかすのと、四路戸に答えを提示するのと、どちらが評価コストが低いのだろう。必ずしも正しい答えを提示する必要はない。ただ、彼女が納得する答えを導き出せばいいのだ。俺は覚悟を決めた。

「ああ、分かったよ……」

5.2 II

英国はエディンバラに留学している姉貴の言付けで、それなりに伝統ある古典論理部に入部したのがこの春のことである。そんなことでもなければ部活などという酔狂は、俺には無縁だったはずなのだ。高校に入学したとはいえ、特に何もやりたいこともない故、何をするかもよく分からない古典論理部なるものに入部したのがやはり何の間違いだったのか。

廃部寸前の古典論理部に入部しようという珍妙な奴がもう一人。部室に充てがわれている特別棟四階の地学講義室に入ると、そこに四路戸はいた。「豪農」四路戸家の一人娘にして、お嬢様らしい整った容姿。そしてそれに似合わぬ、旺盛な好奇心。答えを求めて俺を見つめる真っ直ぐな目には、何か逆らえぬものを感じた。

曰く、彼女の叔父は 40 年前のλ文字山高校古典論理部に所属していたらしい。そこで何やらとても悲しい出来事が起こり、彼は退学になってしまった。幼い頃その話を聞いてとても悲しい思い出のある四路戸は、40 年前の真相を確かめ、叔父の無念を晴らしたい、とまあそんな理由で古典論

理部に入部したとのことだ。その謎を俺がたまたま解いてしまったもんだから、事あるごとに気になることを付きつけられることになるわけだが、その話は又の機会に譲るとしよう。

そもそも古典論理部とは何をする部なのか？ そもそもの話として、古典論理とはなんなのか。実のところ俺にもよく分かっていない。姉貴に言われるままに入部しただけなのだ。部長を務める四路戸にもそこのところが分かっているのかは疑問である。俺達は未だに部活動らしきものをしたことがない。今のところ判明しているのは、文化祭で伝統ある文集「評価」を出すことだけだ。一体、何のためにある部活なのだろうか。

5.3 III

「どうしてこのプログラムがうまく動かないのでしょうか？ わたし、気になります」

特に活動のない俺達は、部室に据え付けられた PC で、プログラミングをすることが当面の活動になっていた。特にそのことに理由や目的があるわけではない。たまたま部室に「Learn You a Haskell for Great Good!」なる書籍の変な表紙の和訳本が転がっていたから、手慰みにいじってみたに過ぎない。

Haskell というのは純粋関数型プログラミング言語というご大層な枕詞を冠するプログラミング言語で、遅延評価を特徴とする。「やらなくていいことなら、やらない。やらなければいけないことならギリギリまでやらない」俺の信条に遅延評価とやらがどういうことかぴったり合致したもんだから、Haskell を書くのは今のところそんなに億劫ではない。

「これ、昔の『評価』に書かれていた式なんですけど」

「へえ、評価ってプログラミングの記事も載ってたんだ」

里志が驚いてそう言った。俺もそんな話は初耳だったが、考えてみれば当然の話だ。俺は評価を読む必要に迫られていなかったから、読んでいなかったのだ。

「ええ、そうみたいですね。これ、このフィボナッチ数の計算なんですけど」

四路戸の手元の「評価」には、幾つかのプログラムらしきコード、それと「フィボナッチ数の定数時間アルゴリズム」というタイトルが見える。

「フィボナッチ数列はご存知ですよね。初項と第二項をそれぞれ 1, 1 と定義して、第三項以降を一つ前の項と二つ前の項の和と……」

「大丈夫だ、さすがにフィボナッチ数列は知っている。それより要点を手短に頼む」

「ちょっと折具、私はよく知らないんだけど」

「摩耶花には僕から説明しとくよ。それより四路戸さん、続きを頼むよ」

不満気な表情を見せる摩耶花を里志がなだめる。

「いいよ、ふぉーちゃん。続けて」

「あ、はい」

話を遮られていた四路戸が、コホンと口に手を当てて、改めて話し始める。

「この記事には、フィボナッチ数の第 n 項を求めるアルゴリズムが幾つか載っていました。最終的には第 n 項が定数オーダで求まるようになるようなのですが、わたしにはまだ理解できていません」

難しそうな顔をしてそう言う。と次の瞬間には急にしかつめらしい表情になって、

「それはともかく、記事の最初には指数オーダのプログラムが載っていたんです。それは再帰で書かれたとっても自明なアルゴリズムで、当然のことですけど、それは入力の大きさに対して指数関数的な時間がかかってしまいます」

ふむ、当然といえば当然だ。しかし気になるところが一つある。

「なあ四路戸」

「何でしょう、折具さん」

「さっき、第 n 項が定数オーダで、と言ったか？」

「はい、そうですが……」

そう言って、四路戸は「評価」に載っている数式を指さした。

$$F_n = \frac{1}{\sqrt{5}}\left\{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right\}$$

「フィボナッチ数の一般項がこれで表されるそうなんです。どうして整数のみの定義から平方根を含む式が出てくるのでしょうか？ とても不思議です」

「ああ、そうだな」

それはそれで面白い話だが、今回気になったのはそれじゃない。

「しかし、問題はそこじゃない。この式を用いても、その値は定数時間では計算出来ない。フィボナッチ数は n に対して指数的に大きくなることが知られている。大きな数をコンピュータで表現するには、それに見合う量のメモリが必要だ。整数 n を表現するのに必要なビット数はいくつだ？」

「ええと… $\log n$ でしょうか？」

「そうだ。指数オーダの数の対数を取れば結局元の値に対して線形のメモリが必要になる。入力に線形のサイズのメモリを出力するプログラムは必然的に…」

「線形時間を要する、というわけだね」

「ああ、そうだ。それに加えて、この式には累乗が含まれている。累乗は場合にもよるが普通は計算するのに対数時間がかかる」

「トータルでは $O(n\log n)$ ほど掛かりそうね」

「おそらくそうなるだろう」

四路戸はまたもしばらく難しそうな顔をしていたが、突然ハッとしたように顔を上げて、目を丸くしている。

「あっ、じゃあこの記事が間違っているのでしょうか？」

「これだけではなんとも言えないが――――」

当時の高校生が書いた記事だ。当然誤りが含まれることもあるだろう。

「計算量の議論を行うためには、計算モデルを予め仮定する必要がある。計算モデルによって計算量が変わってくるからだ。一般的には、定数時間でメモリにランダムアクセスできる『RAM モデル』というものを仮定することが多いが」

ふう、と一息ついて

「いずれにせよ、大きな数を扱うためには、どういう操作を定数時間で行えるのかきちんと定義する必要がある」

「へえ、なるほどね。奉太郎、いつの間に計算量の話に詳しくなったんだい？」

「詳しくなんかないさ、たまたま知ってただけだ」

感心したように四路戸はこちらを見つめる。

「で、お前が気になってたってのは、どういう話なんだ？」

「ええと、何の話でしたっけ？」

5.4 IV

「あっ、そうでした。メモ化の話です」

しばらくうんうんと唸って、ようやく思い出したかのように四路戸は言った。

「ちょっとまって、ふぉーちゃん。そんな話だったっけ？」

「あれ、違いましたっけ」

「プログラムが上手く動かないという話じゃなかったか？」

「あ、そうです。フィボナッチ数なんですけど」

四路戸には結論を先走り、間の説明を飛ばしがちなところがある。集中すると周りが見えていないとも言えるか。

「指数時間の次に載っていたのが、線形時間アルゴリズムなんです。折具さんの話を踏まえると、これも線形時間かどうかは怪しいのですが、とりあえずそのことは置いておきますね。それで、これがそのプログラムなんですけど」

```
int fib1(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    return fib1(n-1) + fib1(n-2);
}
```

```
int fib2(int n)
{
    int a = 1, b = 1;
    for (int i = 0; i < n; ++i) {
        int c = a + b;
        a = b;
        b = c;
    }
    return a;
}
```

「fib1 が指数アルゴリズムで、fib2 が線形アルゴリズムというわけだな」

「その通りです」

満足気に首を縦に振っている。表情がくるくると変わって、見ているだけで飽きない。

「fib1 は、フィボナッチ数列の定義をそのまま書き下した形、fib2 はループで実装した形です、それでここからなのですが」

ぐっと体を乗り出して説明を続ける。

「メモ化というのが次に登場するんです」

「なるほどね。ループというのは、ある意味好ましくない。再帰的な定義をそれを計算する手続きに変換しているわけだからね。指数アルゴリズムの方は、再帰的な、あるいは宣言的なとも言えるね。この実装の問題点は、同じ値を何度も計算するところさ。だから...」

「メモ化を使えば、宣言的な実装のまま、線形アルゴリズムにできるというわけね」

「その通りです」

```
int tbl[100];

int fib3(int n)
{
    assert(n < 100);
    if (tbl[n] > 0) return tbl[n];
    if (n == 0) return 1;
    if (n == 1) return 1;
    return tbl[n] = fib3(n-1) + fib3(n-2);
}
```

fib(4) の計算を考えてみる。fib(4) は fib(3)+fib(2) に展開されるので、次は fib(3) を求める。fib(3) は fib(2)+fib(1) に展開され、fib(2) は fib(1)+fib(0)=2 である。fib(1) は 1 なので、fib(3) は 3 ということになる。さて、今度は fib(2) だが、fib(2) は fib(3) を計算する過程で求めたはずだ。ナイーブな実装では、馬鹿正直に fib(1)+fib(0) と展開し、計算する必要がある。だが、fib(n) は引数 n に対して一意な値を返すのだから、本来は何度も計算する必要はないはずだ。最初に計算した時に、その結果を覚えておけば良いのだ。これが今回のプログラムに対するメモ化の目的だろう。

「そこで、これからが本題なのですが、メモ化と聞いて、折具さんは何か思い出しませんか？」

「ん...、特に思い当たるフシはないが」

「Haskell のグラフ簡約だね、四路戸さん」

「その通りです！ わたしはこの問題、Haskell なら楽に書けるのではないかと思ったんです。Haskell は評価した式をデフォルトでメモ化すると聞きました。遅延評価を実現するためですね。そこで Haskell を用いて、再帰的なフィボナッチ数のプログラムを書いてみたんです」

そう言って、エディタの画面を開いてみせた。そこには数行の小さなプログラムが書かれていた。

```
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

「定義をそのまま書き下した形ね」

「はい。fib の呼び出しがメモ化されるなら、これで十分な速度が出ると思ったのですが...」

四路戸は次に ghci の端末を開いてみせた。

```
ghci> fib 50
_
```

ghci は沈黙したままカーソルを点滅させていた。

「うーん、確かに気になるね。奉太郎」

里志はその様子を見て、顎に手を手をあてしきりにさすっている。言葉とは違い、表情は至って楽しげだ。

「四路戸」

「何でしょう？」

「まずはじめに言っておくが、グラフ簡約とメモ化は違う。グラフ簡約においてメモ化は行われるが、それは今回お前の期待するようなものではない」

「なるほど、そうでしたか」

四路戸はがっくりと肩を落とす。

「では、どうすれば正しくメモ化されるようになるのでしょう？ 純粋関数型の Haskell を使えば、簡単にできると思ったのですが……。これがメモ化されないのなら、参照透過性とは何のためにあるんでしょう。変数を書き換えられない Haskell で、どうやってメモ化を行えばいいんでしょう。どうすれば――」

「ちょっと待て」

まくし立てる四路戸を遮って、俺は、また面倒になったものだと思った。しかし、今回も納得するまで帰らせてはもらえなさそうだ。なんとか手短に解説したいところだが。

「純粋関数型言語というものは、そんなに特別なものではない。評価戦略が違うだけだ」

俺は、自分で言ってから首を捻り、

「いや、違うな。評価戦略はプログラミング言語のカテゴリとは関係はない。たまたま Haskell が遅延評価を採用している、それに過ぎない。同じように、Haskell は参照透明であることを言語の大元となるコンセプトとして採用した。の二つは不可分なものでもないし、特に相関のあるものでもない。ただ、遅延評価を行う上では、参照透明であることは有利に働くことはある」

「うーん、ではどういう場合にメモ化が行われるのでしょう？」

「グラフ簡約では、同じノードの評価結果は二回以上評価されない。それを実現するために、Haskell では評価の結果を保存するようになっている」

「それで、先ほどの関数がメモ化されないというのは…？」

「グラフ簡約のノード、Haskell ではサンク（thunk）と呼ばれるものだが、それが関数呼び出しごとに違うものが作られるからだ。`fib 4` の評価は `fib 3` と `fib 2` のサンクを作って、`fib 3` の評価は `fib 2` と `fib 1` のサンクをそれぞれ作るが、ここで最初に作られた `fib 2` と二回目に作られた `fib 2` のサンクがそれぞれ異なるものになるということだな」

「どうしてそのようになっているのでしょう？」

「もちろん、そういうものを同じサンクにする実装も考えられる。参照透明のでは同じ引数に対して同じ値が返ることが保証されているからな。だが、そういうものをすべて記録していたら、呼び出した引数の数だけどこかにその結果が保存されることになる。保存された値は、それが後に利用されないことを確認することができないか、あるいはできたとしても難しい。つまりそのようにしてメモ化された値は、GC によって回収することが困難になる。プログラムの実行につれて、メモリ使用量が無尽蔵に増えていく処理系なんてのは、とても扱いづらいものになるだろう」

「ははあ、なるほど」

「もう一つ、そういったメモ化を実現するためには、ある引数での評価結果がすでに存在しているか、確かめなければいけない。だが、一般的にそういう操作を行うのは困難だ」

「ええと……、実行コストの問題でしょうか？ 例えば、ハッシュテーブルを用いるとか……？」

「ハッシュテーブルは利用できない。ハッシュ表を引くとき、どういう操作が必要になるかを考えてみてくれ」

「うーん、まずキーに対してハッシュ値を計算して、それからその値で配列を引いて……あっ、ハッシュ値ってどうやって計算すればいいのでしょう？」

「任意の型の値に対して、正しいハッシュ値を計算する関数がない。正しいハッシュ値というのは……」

「同じ値に対しては、同じハッシュ値になるということだね、奉太郎」

「そうだ。たとえそれができたとしても、次に問題になるのはハッシュ関数の品質の問題だ。ハッ

シュ関数の品質というのは……」
「ハッシュ値の衝突が少ないってことよね、折具」
「……そうだ。これらの理由から、ハッシュテーブルを用いてデフォルトのメモ化を実装するというのは良い選択とは言えないだろう」
「赤黒木などの、平衡二分木ならどうでしょう」
「今度はキーに全順序が付けられるという条件がつく。尤も、Haskell では全順序を持つ型、つまり Ord クラスのインスタンスは自動で導出できるわけだが」
「いずれにせよ、自動でメモ化が行われると、思わぬところで困ってしまいそうですね」

5.5 V

ひとまずは四路戸の納得する説明を与えられたようだ。
「それでだ」
漸く本題の方に入れそうだ。
「フィボナッチ数列のメモ化を行うにはどうすればいいのか」
「そうでした！」
すっかり忘れていたみたいだ。
「有名なコードはこんな感じのものだな」
俺はカタカタとキーボードを叩いて、次のコードを画面に写した。

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

「ああ、このコードなら僕も見たことがあるよ」
「Haskell の遅延評価と引き合いに出される、有名なやつよね」
さすがに有名なコードだ。
「でも、いまいちこれがどういう仕組みでうまく動くのか分からないのよね」
「このコードで言いたいのは、動作原理じゃないんじゃないかな。右にある `fibs` の呼び出しが、このコードではどうしてメモ化されるのか？」
「ご名答」
ghci に式を打ち込むと、今度はすぐに結果を返した。

```
ghci> fibs !! 50
20365011074
```

「あの、ちょっと待ってください」
四路戸が怪訝そうな顔をして、
「ここの右辺にある `fibs` の参照って、呼び出しなんでしょうか？ 変数の参照に見えるのですが」
「ああ、そうだったな」
もっともな質問だ。
「まず Haskell では、定数と 0 引数関数の区別がない。つまりそもそも関数と変数の区別はない。区別する必要がないからだ」
「ええと、参照透明性のおかげでしょうか？ 関数は引数の値にのみ依存して結果の値を返す…と言うことはつまり、0 引数の関数はいつも同じ値を返さなければいけないということですね」

「そういうことだ。それで、この” 関数呼び出し” が、関数呼び出しであると同時に単なる変数への参照でもあるわけだ。つまり、ここでの `fibs` への参照は、同じサンクへの参照になるわけで、最終的にはメモ化される事になる」

「なるほど。0 引数の関数ならメモ化されるというわけですね」

「実際の GHC は正確にはもう少し違う挙動になる。Lambda Lifting も考慮すると、トップレベルにメモ化されるサンクはもっと多くなるからな。GHC はこういう関数のことを “CAF（Constant Applicative Form）” と呼んでいる」

「折具さん、CAF というのは……？」

さすがにそこまで説明するのはめんどくさいし、今それは必要とされていないだろう。

「CAF については HaskellWiki に詳しい説明が書いてあるから、そっちを読んでくれ。とりあえず、自動的にメモ化されるのは CAF と呼ばれるものだけだということ、CAF は 0 引数の関数だということだ」

「なるほど、だいたい分かりました、ただ...」

「ただ……？」

「ただ、まだ少し気になるところが」

「何だろうか」

「はい、CAF というものがメモ化されるということは分かりました。でも、CAF が大きくなることはないんでしょうか？ 例えば先ほどの……」

そう言ってターミナルに式を打ち込む。

```
Prelude> fibs !! 100000
420269270299515438631900510129391513177391570263223450330471608719833
573145727622663393847726701366096253366170285832918664116229882221533
...
994892588234011350360387511421993302025047776808575499810068887160787
732953050111241465547976334636931551115834357110038285979669707537501
```

「あ、大丈夫でしたね」

ターミナルに数万桁の数字が表示されたのを見届けると、四路戸はそう言った。

「へえ、分かってはいたけど、本当に大きな数を扱えるものなんだね」

里志は少し驚いたような素振りを見せている。

「で、このメモリ使用量なんですけど」

四路戸は top を起動してメモリ使用量を探しているようだ。

「ええと……あ、ありました。これでだいたい 600MB 消費しているようです」

「そうだな、意外と食うもんだ」

「はい。それでこれが問題にならないのかなって」

「こんなに多くのメモリが、GC に回収できない状態になって、大丈夫かってことね」

「はい！ そうです」

「そうだな、確かに問題になることはある。だけど、基本的に CAF には大きくなり続けるデータを置かない。つまり、この `fibs` のようなものは普通はやらないし、やることはあまり良くない」

「へえ、そうなのかい。僕はてっきりこういう書き方をするのが良いものだと思っていたのだけど」

「メモリが増え続けるバグっていうのは、他の種類のバグの入り込みにくい Haskell にとっては相対的に大きな問題になる。そういう時にはヒーププロファイラなんかを使って問題を特定すること

になるんだが、大きな原因の一つが、遅延評価に伴ってサンクが成長し続けるスペースリークで、もう一つの大きな原因が」
「CAF が成長し続けるってことね」
「ああ、そうだ」
「……なるほど、だいたい分かりました。ではメモ化をやるうまい方法は」
「遅延評価をうまく利用しつつというのは難しいだろうな」
「そうですか……」
そう言って四路戸は俯く。残念だという風だが、ともあれ納得はしてくれたようだ。
「しかし、意外だね」
中空に式を書くように、指をクルクルさせながら里志が言う。
「何が意外なんだ？」
「奉太郎がここまで Haskell を勉強していたとはね。必要じゃないことはやらない。必要なことも、必要になるまではやらない、いつもの奉太郎はどこに行ったんだい？」
「たまたまだ。偶然詳しかっただけで」
「……そうかい。まあいいさ、僕は奉太郎が終に無気力な学生生活に別れを告げて、自分の使命を全うし始めたのかと思ったのだけどね」
「買い被り過ぎだ」
俺は PC の電源を落とし、帰り支度を始めた。

5.6 VI

λ須先輩の誘いを断りきれず、どういうわけだかこうして二人で喫茶店にいるのは、それから幾日も経たない日のことだった。
「—— 君の高名は聞いている。随分とプログラミングが得意のようだね」
「そんな、たまたまです」
面倒な事になった。λ文字山高校理科部のλ須冬実、先輩の部ではこの夏に開催される UltraCon なるプログラミングのコンテストに参加する予定だったらしい。ところが、プログラムの提出期限も間近というところで、メインで書いていたプログラマが失踪してしまったらしい。プログラムはバグでうまく動かない。それで、残されたプログラムの完成を古典論理部に依頼してきたというわけだ。
「たまたまというのは、そう何度も続くものではないだろう。折具、私は君が非凡な才能を持っていると思っている」
「買い被り過ぎです」
「そうだろうか。君の活躍は四路戸から聞いている。そこから判断した、私が保証しよう」
本当に俺は、特別な才能を持っているのだろうか。平凡な高校生だと、たまたまだと思い込もとしていただけなのだろうか。
「才能を持つ者は、それを正しく利用しなければならない。それが才能を持たない者への責務だ。—— この話、考えておいてくれ」
λ文字山高校の「女帝」は、そう言って立ち去った。

5.7 VII

「引き受けることにした」
「えっ、どういう心境の変化なんだい？」
「特に理由はないさ」
「へえ、まあ詳しくは聞かないよ」

里志は肩を竦めた。

さて、今回のあらましはこうだ。λ須先輩の部では UltraCon に参加するために、まずその予選の改題を提出しなければならなかった。予選課題とはいえ、プログラムには正確さとしっかりとした速度が求められる。そのためにプログラムは Haskell で書かれる事になった。しかし締め切りも間近となった昨日、提出するためのプログラムを書いていた生徒が失踪したというわけだ。

提出するプログラムを任せられる部員が他にいなかったため、俺のことを聞きつけたλ須先輩によって、古典論理部にお鉢が回ってきたのだ。そもそもそういうプログラムを部外者に任せていいものなのか、よくわからないのだが……。

「それで、どういった課題だったのでしょうか？」

いつの間にかやってきていた四路戸が、俺の背後から PC のモニタをのぞき込んでいた。依頼を受けた際に、課題と完成途中の課題の一式をあずかっていた。プログラムをエディタで開く。

しばらくプログラムを眺めていると、

「このプログラム、どこが未完成なんでしょうか？ 一見するとそういったところは見当たらないようですが……」

「えーと、このプログラムは」

コンパイラにかけてみる。エラーも出ずにコンパイルは成功した。適当な入力を試してみると、そのプログラムは期待通りの答えを返す。

「ふむ、うまく動いているように見えるね」

「ところがだな」

俺は適当なプログラムを書き始めた。

「何を書かれているんでしょう？」

「大きな入力を生成するプログラムを書いているんだ」

しばらくしてプログラムを書き終えると、それが生成したデータを、λ須先輩から受け取ったプログラムに入力してみた。

「……うーん、全然ダメね」

「なるほど、実行速度がいけないんですね！ でも、このプログラムは一体どこが……」

つまり、こういうことだ。正しく動作するプログラムはできているのだが、見たところ満足な速度では動作していない。λ須先輩から聞いたところによると、これを書いた生徒には、パフォーマンスに関してはあまり心配していなかったという。ここから僅かの変更で十分な速度を達成できる公算があったということだろう。

「この前のサンク、評価するときが案外早く来たのかもしれないな」

5.8 VIII

渡されたプログラムは、比較的単純な部類に入る。一通り読めば、どういうことをやっているのかは簡単に理解できた。つまり、

「この探索問題を解くために、一番単純な深さ優先探索でアルゴリズムが実装されているようだな」

「へえ、でもそれじゃあ、速度が遅いのは当たり前じゃないか」

「そうね、それでどう改良するつもりだったのかしら」

「それを考えるのが、俺達の役目だというわけだ」

古典論理部に改善を依頼したといっても、元のプログラムの意志は継いで欲しいと。プログラムを書いたやつの意図を、そのプログラムから読み取って、見事プログラムを完成させる。そんなことが、俺にできるのだろうか。

「なんでも、このコードを書いた生徒は、GHC のバージョンにえらくこだわっていたようだが、

それが何に影響するのかは今のところよくわからない」
「最適化、正格性解析が新しい GHC だとうまくいくとかだったりして？」
「そこまでこだわりが必要なところだろうか」
まあ、案ずるより産むが易しだ。
「ところで、理科部の何人かもプログラム改善のアイデアを出してきたらしい。まずはそれを参考にして欲しいということだ」
「折具さん、それを早く言ってください！」

5.9 IX

「なるほど。大雑把に言うと、このプログラムは基本的に探索を行うもので、探索空間はそんなに広くない。だから、メモ化すれば高速に動作するはずだ。そういうわけだね」
しばらく全員でファイルに目を通してから、里志が口を開いた。
「まあ、そういうことになる。これを書いたやつも、おそらくメモ化をしたかったんだろう。だが、肝心のその方法が…」
「わからないというわけね」
そうなのだ。Haskell ではメモ化は自動的には行われない。かと言って、適当なテーブルに関数の結果を書き込むのは実は Haskell では自明ではない。Haskell は純粋関数型言語、参照透明だからだ。普通の関数がその中で結果をキャッシュするということはできない。
「テーブルへ書き込むには、関数を IO モナドにする必要があるね」
「ST モナドでも大丈夫ですよ」
「そうだな。いずれかにせよ、結果をメモリにキャッシュするという操作は参照透明性を破る副作用に相当する。つまり元の関数の形をかなり大きく崩さなければならないことになる。それはあまり綺麗とは言わないのではないだろうか」
「コードの綺麗さか。意外だね。奉太郎がそんなことを口にするなんて」
「他のものは知らんが、プログラムを綺麗に保つというのは、それなりに重要なことだ。プログラムの理解しやすさといじりやすさは、そのままメンテナンス性に繋がるからな」
「でも、プログラムの綺麗さというのは、なんだか曖昧な気がします。何を綺麗で、何をそうでないのかというのは、とても主観的な基準だと思います」
たしかに、そういう側面もあるだろう。俺は首を捻りながら、
「そうだな、しかし実際に利益をもたらすものだというと、そこに求められる性質も自ずと普遍的になる。少なくとも考えられるのは、プログラムが抽象的であることだな。同じコードの複数回の繰り返しはひとつにまとめたい。そのまとめ方が無理なく自然に行えていることが、プログラムが綺麗だといえるだろう」
「それで、IO モナドは良くないという話かな」
「IO モナドというよりも、メモ化という操作それ自体を抽象化したいという話でしょうか？ IO を使ってテーブルに書き込む操作は、何も抽象化されていません。メモ化したい関数ごとにすべての操作を手で記述する必要があります」
「そういうことだ。わざわざ探索プログラムを Haskell で書こうというやつが、そういう手法を用いようとしていただろうか」
なんだか話が逸れて来たので、本題に戻すとしよう。
「では、順番に見ていこうか」

5.10　X

「...つまり、このプログラムは DP をしようってのに、単なる深さ優先になってるってことだろ？なら要するに、メモ化をすればいいってことだ。

どうやってメモ化をするのかって？ なんでもいいんじゃないのか、とりあえずうまく動きさえすれば。例えば、CAF にリストを置くとかな。CAF に置かれたものはグラフ簡約の同一のノードへの参照としてキャッシュされることになるんだったな。全部の答えを格納するリストをグローバルに置いときゃいいんだ。例えばだな、簡単のためにフィボナッチ関数のメモ化で考えてみると、

```
fibs :: [Integer]
fibs = map fib [0..] where
  fib 0 = 1
  fib 1 = 1
  fib n = fibs !! (n - 1) + fibs !! (n - 2)
```

こうなる。フィボナッチといえば、有名なあのコードがあるが、それはひとまず忘れてくれ。`fibs = map fib [0..]` これですべての答えの入ったリストがいっちょ上がりだ。あとはいつもどおりに関数を書けばいい。気をつけるのは、再帰呼び出しの代わりに、この『アカシックレコード』から答えを引っ張ってくることだな。依存関係が DAG になっていれば、この問い合わせはループしないことが保証される。どうだ、シンプルだし、これで文句ないだろ。

GC 対象にならない？ 知ったこっちゃねぇよ。こんなライフタイムの短いプログラムのメモリ増加を気にする必要があるか？ リストのアクセス時間？ そんなのは指数時間に比べれば無視できるんじゃないか。

とにかく、もう締め切りまで時間がない。俺はこれで十分だと思うぜ」

プログラムとともに受け取ったファイルの中に、理科部で行われた議論の様子を収録したビデオが入っていた。俺達はまず “1.avi” なるファイルを鑑賞することにした。ビデオは “3.avi” まであるようだ。部内でこんなに案が出ているなら、部外者である古典論理部にわざわざ依頼しに来るという理由は何だろうか？

「なんだか、大雑把そうな方でしたね」

その一つ目の動画では、えらくいかつい男が、自分の手法をまくし立てていた。

「プログラムなんてものは、なんだかんだ言っても、正しく動くことが最重要だからね。それで、奉太郎。このアイデアはどうなんだい？ 彼も言うように、このアイデアには特に問題がないと言えるかい？」

そう言って里志は何か試すような視線をこちらに投げかけてくる。暫し考えた俺の答えは、

「.....却下だ」

「何がいけないの、折具？ 私には特に問題なく感じたんだけど」

「引数が整数引数一つに固定、しかも小さい範囲でなければいけないところでしょうか？」

首を傾げながら四路戸が尋ねる。

「いや、それはそこまで問題にはならないと思う。二引数の関数、そうだな、簡単な例として LCS を実装してみようか」

「LCS とは何でしょう...？」

「Longest-common subsequence、日本語だと最長一致部分文字列かな。二つの文字列が与えられた時に、両方に部分文字列として含まれる文字列のうち、最長のものを求めるという問題さ、例えば...」

里志がカタカタとキーボードを叩き、例を示す。

「文字列 “hello” と “haloo” があったとする。これらの部分文字列として一番長いものは、“hlo” という 3 文字の文字列になるよね。これを求める問題というわけさ」

「なるほど、部分文字列というのは、元の文字列から順序を変えずに部分集合を取り出したものという意味ですか」

納得した様子で、四路戸は頷いている。確かに部分文字列という表現は若干曖昧なところがある。少なくとも、今回考える LCS での部分文字列の定義というのはそういうものになる。

「ところで LCS には、それぞれの文字列の長さを m、n として、O(mn) の効率的な DP を用いた解放が存在する。これは、m× n のサイズのテーブルを用いることになるが、再帰とメモ化を用いて実装すれば、二引数の関数をメモ化する問題と同じくなる」

「なんだかイメージが湧かないわね」

摩耶花はあまりこういうアルゴリズムには明るくなかったか。古典論理部の復活のために、図書部と漫画部との掛け持ちで、こんな話にまで巻き込んでしまっては何か申し訳ない気分になってくるが。

「難しく考える必要はない。最初は適当に再帰的な解法から始めればいいんだ」

俺は簡単な LCS の解法を書き始めた。

```
lcs :: String -> String -> Int
lcs [] _ = 0
lcs _ [] = 0
lcs (x:xs) (y:ys)
  | x == y = 1 + lcs xs ys
  | otherwise =
    lcs (x:xs) ys `max` lcs xs (y:ys)
```

「見ての通り、何でもいいから答えを出すコードを書けばいいならとても簡単だ。再帰的なアルゴリズムで、規定ケースはどちらかの文字列が空文字列の時で、これは明らかに 0 になる。一般のケースについては、先頭文字が一致してい残りの文字列で再帰、一致していないときは、どちらかの先頭を捨てて、良い方を採用する」

「でも、このコードは一番下のケースのおかげで、指数関数アルゴリズムになっているというわけですね！」

「ああ。だが、これは明らかに同じ計算を何度も行なっている。つまり、同じ引数については必ず同じ値になるはずだということだ。そこでメモ化が使える」

```
lcs :: String -> String -> Int
lcs a b = tbl !! 0 !! 0 where
  m = length a
  n = length b
  tbl = [ [ f i j | j <- [0..n]] | i <- [0..m] ]
  f i j
    | i == m = 0
    | j == n = 0
    | a !! i == b !! j = 1 + (tbl !! (i + 1) !! (j + 1))
    | otherwise =
```

```
    (tbl !! (i + 1) !! j) `max` (tbl !! i !! (j + 1))
```

続いて俺はメモ化を行うように改変したコードを端末に打ち込んだ。

「見ての通り、さっきのコードを素直にメモ化したものだ。元々の関数は `f` に相当する。再帰呼び出しで f を呼ぶ代わりに、すべての答えが格納されているはずの `tbl` を参照しているところがやはり違うが」

「へえ、二引数以上でも素直に実装できるんだね」

「これは `fib` での方法と比べて、別の利点もある。今回は `tbl` が CAF ではないということだ」

「えっ、CAF じゃなくてもメモ化されるんですか？」

「グラフ簡約としてのノードが等しければ、計算結果はキャッシュされる。CAF はその判断材料が必要ないということだな」

「つまり、`tbl` は一連の `f` を計算している間、常に同じ物が参照されるというわけだね」

「ああもう、よくわからなくなってきた！」

グラフ簡約と CAF がよくわかっていないとここの簡単なルールは見出しにくいのかもしれない。

「ともかく、ここで言いたいのは、この関数のしかる評価が終了した後に、`tbl` の指すリストは速やかに GC 対象になるということだ」

「たしかに、CAF じゃなくなるなら、当然そうなるね」

「今回のケースは引数によって `tbl` の値が異なるから、こういう形のコードにするのは必然だ。しかし、`fib` の場合でも、同じアプローチによってテーブルを CAF でなくすることは出来る。このあたりは実装の簡便さとパフォーマンス要求のトレードオフになるだろうな」

「んん、じゃあ結局折具はこの方法のどこが問題だというのよ？」

CAF の問題は解決可能だ。だから、問題は他のところにある。

「問題の一つは、計算量のオーダだ。確かに指数オーダと比べれば無視できるほどには小さいが、多項式時間だから良いというわけではない。$O(n^2)$ と $O(n^3)$ は現実には違いが大きい。実際のところ、リストの要素アクセスが入ると、本質的ではないところで一つオーダが上がってしまう」

「リストを使わない、例えば配列を使うといったことはできないんでしょうか？」

「配列を使う方法も考えられる。望ましいことに、メモ化のキャッシュ内容は一度決まると変更されることはない。計算の進行とともに、計算の結果が決まっていくのに、遅延評価を利用すれば、あたかも初めから定数として答えが存在していたかのようにコードを書くことが出来る。つまり、答えをすべて保持している immutable な配列を作るだけでいい。計算の順序やら、メモリへの格納はすべて Haskell の処理系が自動でよろしくやってくれる。ある意味でとても宣言的なアプローチだとも言えるだろうな」

「じゃあ、これは別段大きな問題にはならないんでしょうか？」

「配列を使えばそうなるな」

「それでもやっぱり問題はあるのかい？」

「残りの問題の一つは、これでもやはり小さい範囲の整数しか扱いにくいということだ。それともう一つ、俺が気になるのは、メモ化されたプログラムの導出の問題だ。それがいかに容易に、機械的に導出できるものだったとしても、やはり人間が関数ごとに手を動かさないといけないのは抽象化の限界と言えるだろう」

そこまで言ってから、ふうと息をつく。

「というわけで、この案は却下だ」

5.11 XI

「さて、俺の番か。メモ化をするんだったな。メモ化なんてのは、本来プログラムの変更なしで出来るようになるべきものだ。例えば Ruby なんかには、関数を受け取って、その関数の定義を書き換えて、自動でメモ化バージョンに変えるものがあるが、参照透明な Haskell ではそういうことは不可能だ。

ここは関数型言語であるところを逆手に取って、`fix` を活用出来るんじゃないか。`fix` は普通は再帰を行うために導入されるプリミティブだが、`fix` の代わりに別のものに関数を渡せば、元の関数の動作を変えられるということでもある。与えられた関数をメモ付き再帰版にする `memoFix` という関数を作るのが考えられるね。

```
fix :: (a -> a) -> a
memoFix :: (a -> a) -> a
```

こんな関数ができたとすれば、`fib` はこう書ける。

```
fib_f f n = case n of
  0 -> 1
  1 -> 1
  _ -> f (n - 1) + f (n - 2)

fib     = fix fib_f
fibMemo = memoFix fib_f
```

`fix` の扱い方に慣れていればなんということはない。`fib` が通常の再帰を行うバージョン、`fibMemo` がメモ化再帰を行うバージョン。メモ化機能を追加するために、元の関数を書き換える必要がないというところが重要なところさ。

さて `memoFix` の実装だけど、計算済みならそれを返して、そうでなければ計算してそれを返すというのを書きたいんだけど、生憎の参照透明だ。ここは『アカシックレコード』に再登場願おう。関数のすべての引数に対して、その評価値を記録しているテーブルだ。

```
memoFix :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
memoFix f = g where
  g n = tbl !! n
  tbl = map (f g) [0..]
```

見ての通り実装はとてもシンプルだ。引数が Int に固定されてしまったが、これは `Ix a` 型に簡単に変更できるだろう。引数の範囲を大きくできない問題は、アカシックレコードをテーブルからツリーに変更してやればいい。ツリーといってもとても簡単なものでいい。すべての整数に対してツリーを作るんだ。いわば『完全』『平衡』『無限』二分木を作ればいい。つまり、

```
data Tree a = Node a (Tree a) (Tree a)
```

```
mkTree :: (Int -> a) -> Tree a
mkTree f = g 0 where
  g i = Node (f i) (g $ i * 2 + 1) (g $ i * 2 + 2)

lookupTree :: Tree a -> Int -> a
lookupTree = g where
  g (Node v _ _) 0 = v
  g (Node _ l r) i
    | odd  i = g l ((i - 1) `div` 2)
    | even i = g r ((i - 2) `div` 2)
```

これで、与えられた関数の答えをすべて含む無限に大きい木が完成する。しかもすべての値へのアクセスが $O(\log n)$ で完了する。これらを組み合わせると、

```
data Tree a = Node a (Tree a) (Tree a)

mkTree :: (Int -> a) -> Tree a
mkTree f = g 0 where
  g i = Node (f i) (g $ i * 2 + 1) (g $ i * 2 + 2)

lookupTree :: Tree a -> Int -> a
lookupTree = g where
  g (Node v _ _) 0 = v
  g (Node _ l r) i
    | odd  i = g l ((i - 1) `div` 2)
    | even i = g r ((i - 2) `div` 2)

memoFix :: ((Int -> a) -> (Int -> a)) -> (Int -> a)
memoFix f = g where
  g n = lookupTree tree n
  tree = mkTree (f g)

fib_f f n = case n of
  0 -> 1
  1 -> 1
  _ -> f (n - 1) + f (n - 2)

fib = memoFix fib_f
```

ふむ、案外いいのができたんじゃないかな？」

「いかにも偏屈そうな方が出てきましたよ！」

ビデオに登場した生徒は、何度もメガネをクイとやりながら、滔々と説明をしていた。正直、こういった抽象化は、関数型言語としてはとても美しいとは思うのだが……。

「あの、折具さん。fix というのはどういったものなのでしょうか？」

ああ、そうだったな。当然のように fix が登場していたが、余程なラムダ計算オタクでないと普通はプログラミングに直接用いるものではない。

「fix というのは不動点演算子のことだな。不動点演算子というのは、その名の通り与えられた関数の不動点、つまり fix f = (f x == x になる x) を返す関数だ。型理論では、単純型付きラムダ計算にチューリング完全の力を与えるために追加するプリミティブとしても重要な役割を果たすものだ」

「ええと、折具。ちっともわからないのだけど……」

そうかもしれない。しかし、今回の話はそこが焦点ではない。

「ともかく、関数をメモ化するプリミティブを追加しようという案だということだ」

「うーん、あとで図書館でしらべとくわ……。ラムダ計算の本だっけ？」

摩耶花はイバラの道を歩むことになりそうだ。

「で、奉太郎」

そうだった。

「却下だ」

勿体も付けずにそう言った。

「へえ、どういうところがお気に召さなかったんだい？」

「技巧的すぎる」

端的に言うと、そういうことだ。

「現実的な問題としてはまず、プログラミングモデルとして、fix ベースのものを要求される。これは慣れていない者にとっては書きづらいところがある。引数の数を増やすのも大変だ。現実的には引数の数ごとに memoFix1、memoFix2 などを用意することになるだろう。ここで問題になるのは、メモ化に利用しない引数を追加するのがとても大変だということだ。まあ、この辺は Template Haskell でも使えばなんとかなるのかもしれないが、それは…」

「抽象化の敗北というわけだね」

「ああ、そうなるな」

俺の言いそうなことを見透かされているのが多少気にかかるが、

「それはともかく、やはり技巧に過ぎるものは扱いづらいという一面はあるだろう」

5.12 XII

「えっと……最後はあたしだね。

プログラム速くしたいんだよね？ だったら、遅いところを C で書けばいいんじゃない？ メモ化…だっけ？ C 言語だったらメモリだって、ドガッ、バキッって書き換え放題だよね。

せっかく FFI があるんだしさ、まどろっこしいことする必要はないよ。どーんと書き換えちゃえばいいんだって！」

「な、なんかすごい人が出てきましたよ！」

λ須先輩、最後にとんでもないのをぶつけてきたな。

「却下だ」

とは言いつつも、それが最善であるケースは常に考慮しなければならない。

5.13 XIII

「……では、うちの部のアイデアは三人とも却下というわけだな」

「はい、残念ながら」

それから数日後、何時ぞやの喫茶店で、俺はλ須先輩に結果を告げていた。

「それぞれ確かに、問題を解決する手段ではあります。しかし、どれも抽象化と速度を両立するも

のではありませんでした。果たしてそれが元のプログラマの意図かどうかはわかりませんが、抽象化を抜きにしてプログラミング技法を語るのは難しいでしょう」

「ふむ、なるほどな」

λ須先輩は長い前髪の隙間から覗く鋭い目で、こちらをじっと見つめている。

「では聞こう。君ならあれをどうやって料理する？」

「はい、……」

5.14 XIV

夏休みも近づいた、ある日のことである。

「理科部の皆さん、予選通過されたようですね」

「そうか、それは良かったな」

「本戦はどうされるんでしょうか？」

さあな、λ須先輩が何とかするんじゃないか。

「ところで奉太郎」

「なんだ？」

「結局どういう風にしてメモ化の問題を解決したのさ？」

「あ、そうでした！ 聞いてませんでしたね。わたし、気になります！」

そういえばそうだったな。とは言っても、何も大それたことはないのだが。

「モナドだ」

「なんだ、結局モナドなのかい？」

「ああ、Haskell と言ったらモナドを使うのは普通だろう。むしろ三人の中にモナドを使おうとするやつが出て来なかったのが意外なくらいだ」

「折具、あんた最初 IO や ST 使うの真っ先に否定してたじゃないの？」

「いや、そうじゃない。ここでいうモナドはそういうモナドのことじゃないんだ」

面倒だが、ここまで来たら最後まできっちり説明するとしようか。

「そもそも、モナドというものはどういうものなのか。参照透明の言語副作用を伴う IO を行うために導入されたという説明がなされることもあるが、それはモナドの一側面にすぎない」

「へえ、じゃあ奉太郎はモナドをなんだと考えているんだい？」

「計算の抽象化」

左の中空をぼんやりと眺めながら考えを巡らせる。

「あるいは、計算のコンテクスト。DSL そのもの。はたまた……」

「折具さん、話が抽象的すぎて…」

「ああ、済まない」

話が飛んでしまった。

「つまり、IO モナドというのは、副作用を伴う計算だということだ。ST モナドは同様に、副作用をメモリ操作に制限したモナド。リストモナドは少し変わって非決定計算のモナド、State モナドは状態を持つ計算のモナド。Error モナドはエラーを扱える計算のモナド」

そして、勿体をつけて、

「今回の場合は、計算結果がキャッシュされるモナドを作ればいいというわけだ」

「モナド……そんなことも出来るんですか！？」

四路戸は驚いたような手振りで俺に問う。

「モナドに不可能はない」

何やら柄にもない言葉が口をついて出たが、実際モナドはどんな計算でも表現できるはずだ。

「で、奉太郎は、モナドを使ってメモ化を行ったというわけかい」

「そうなるな」

端末を開き、この前と同じく、fib を書いていく。ただし今度はモナドを用いて。

```
fib :: MonadMemo Int Integer m => Int -> m Integer
fib 0 = return 1
fib 1 = return 1
fib 2 = (+) <$> memo fib (n - 1) <*> memo fib (n - 2)
```

「MonadMemo という型クラスは、メモ化モナドを一般化した形だ。MonadMemo k v m に対して、k がキャッシュするキーの型、v が値の型、m が親のモナド型になる。fib がモナドになった事によって、呼び出し方が若干変わっているが、これは細かいことだ。要するにここで何が重要かというと、メモ化を行う手段を抽象化しているということだ。例えば MonadMemo のインスタンスとして、全くメモ化を行わないという実装を考えることが出来るだろう。あるいは、DB を用いて永続化を行うものも考えられる。一般的な実装は、Ord を持つキーに対する平衡二分木だろうか。何れにしても、行う計算と、それをキャッシュする術が完全に分離されているということは、プログラムの composability としてとても重要なことだ」

「他の方法では、それはできないのかい？」

「できないということはないだろう。ただし、その抽象化を行うためのインターフェースを決めるのはひどく難しいと思う。あるいはモナドのインターフェースに類似したものになるだろう。すでに使用例が多く、十分に性質が研究されている、モナドというインターフェースでこれを抽象化するのは、多くの研究にタダ乗りできるという点でも、とても便利なことなんだ」

「もともとモナドで書かれていないコードをモナドでのコードに直すのは、大変じゃなかったんでしょうか？」

「基本的に、まったくの純粋なコードをモナドのコードにするのはとても簡単だ。なにせ、return をつけるだけでモナドになるんだから。注意するのは再帰的な部分で、これは若干のコード変更が必要になる。その際にはアプリカティブファンクターと呼ばれるものを利用するとコードの見通しが大幅に良くなるだろう」

さて今度こそ一通り説明が終わった。四路戸は画面を見つめて、カチャカチャとコードをいじっていたが、しばらくしてスクリとこちらを向き

「なるほど...モナドでメモ化を抽象化するなんて、思いもよらなかったです。大変勉強になりました。どういう時に、モナドを使えばいいのか、指針のようなものがあればわかりやすいのですが……」

「うーん、そうだな……」

少し考えて、

「モナドによる抽象化は、Haskell プログラミングの醍醐味だ。何かを抽象化したいと考えた時、まずモナドを検討するぐらいでいいんじゃないか」

「何でもモナド、というわけかい」

里志は楽しそうに言った。まあそういうことになるんだろう。

しかし、今回の件も漸く一件落着かといったところで、何かに気づいたような摩耶花が口を開いた。

「ちょっと待って、折具。何かおかしくない？」

何だろうか。何かを見落としてたか...？

「この話、GHC のバージョンに関係ないじゃない」

5.15 XV

これ以降の話は俺にとってはどうでもいい話ではあるが、いつもの調子で事の真相が気になった四路戸がλ須先輩を問い詰めて判明した事実である。

失踪した生徒がもともと想定していた解法は、テンプレート Haskell を用いるものだったらしい。メモ化前とメモ化後の形を同じにしたいというのはやはり考えていたらしい。ところが、関数をメモ化関数に変える関数というのは普通には記述できないので、テンプレート Haskell を使って解決しようとしていたとのことだ。テンプレート Haskell はバージョンによって比較的インターフェースの変更が大きい。それで用いる GHC のバージョンを指定したかったのだろう。

ところが、テンプレート Haskell のコードを書くのは一朝一夕ではうまく行かず、最終的に失踪してしまうこととなった。λ須先輩はそれを知っていたが、テンプレート　Haskell を使いたくなかったのか、別の方法を模索することになる。部員にアイデアを出させるも、今ひとつ満足の行くものが出てこない。そこで俺に別の解法をひねり出させようと、細かいことを隠して依頼してきたということだ。

ちなみに、λ須先輩も別の解法を考えていたそうだ。 GTA 計算と言って、探索系の問題を、解候補の生成（Generate）、解の検査（Testing）、解の集約（Aggrigate）の三つの操作に分解して記述すると、自動的にメモ化、並列化が行われるというとんでもないフレームワークがあるらしく、これを用いて高速化を目論んでいたそうだが、独自の Generator を作るのがうまく行かなかったそうだ。

5.16 Epilogue

「すまない、悪気はなかったんだ」

三度、喫茶店にて。同じくλ須先輩と。だが、今回は心なしかしおらしい表情が伺える。

「君を試すような真似をしたのは謝ろう」

「あなたは別に俺に頼る必要なんてなかった。最悪の場合でも自分でどうとでもできたはずだ。そうですね」

「実際のところ、そうなる。だが……」

少し言葉が胸につっかえるような表情を見せたが、やがて。

「今回の件で確信した。君は自分が思っているほど平凡な人間ではない」

「買い被りです」

「君がどう思おうと思わなかろうと、いずれそれを自覚する時が来るだろう。それまで、その評価値はサンクの殻の中に閉じ込めておくとしようか」

評価する必要のないサンクはそのままゴミ集め機に回収される、だから遅延評価と言うのだ。遅延評価を標榜する怠惰な一学生としては、せいぜいゴミ集め機の働きに期待したいところである。

—— 了

次回予告

21 世紀初頭。ますます成長を続けた IT 大企業は数々の国際組織をも傘下に収める存在となっていた。
世界のすべてを見える化し、ユビキタスに結合するクラウド、Web22.0 計画。完成すればサイバー時代の幕開けを告げるはずだった史上最大の計画は、史上最大の失業対策でもあり、地球人口の半数までもが人月で量って投入され、史上最大のデスマーチを産む運命であった。やがてプログラミングの自由を巡って世界大戦が勃発。—押し付けられた仕様を押し付けられた言語で書かされるのは、もう御免だ— エジャナイザ（打ち壊し）運動によりさまざまなプログラミング言語のエコシステムがほぼ完全に破壊され、人類は「納紀 A.D.00 年 (AD = After Deadline)」をもって新たな歴史をスタートする。
間もなくして、超言語連合（仮）の支配・統制により、「プログラミング」は「人の尊厳を奪うモノ」として規制される「程序禁止法」が成立。一部の地域（伝統芸能絶対防衛圏）を除いては、プログラミング活動は厳しく罰せられる事となった。プログラマが次々に姿を消していく・・・人類のココロは暗い闇に包まれていくのだった–
そして—
かつてプログラミングと呼ばれた営為は絶滅したかに思えた。だが、そんな時代に復活した言語があった。
「Haskell0098」
かつて地球の存亡をかけた戦いの中、傷ついた人々の心に光を灯すべく、最後までプログラミングの楽しみを伝え続けた「Haskell98」。そんな伝説のニンジャ達の光と魂を受け継ぎ、その名を襲名する形で、「非合法言語」として立ち上がったのだ！
非合法がゆえ、公式なライブコーディングは出来ない・・・。程序禁止の時代だからこそ、どんな危険も顧みず、あらゆる星へ強攻突入し、熱狂的なゲリラライブコーディングをファン達に送り届ける。そう、この時代の Haskell は、『遅延関数型言語』から『先攻関数型言語』として進化していたのだ！ 悪に支配された暗い世の中、彼女達の「奇跡のライブコーディング」は、夢を失いかけた人々に未来への希望の光を与え続けていく！
だが、圧倒的な影響力をもつ彼女達に魔の手が襲う・・・彼女達の活動を無視できなくなった超言語連合政府（仮）により、「クラッカー」と断定され、妨害を受ける。自分たちのハッケージを、そしてファンや市民を守る為に、自ら武器を手にとり少女達は戦う。
「待っているファンがいる限り、傷つき倒れても戦い、そして、私達は書き続ける！」
そして、時は納紀 A.D.0098…。
運命の年に、新たなスターとなるべく選ばれた第７７期研究生を中心とした物語が、今、まさに始まろうとしている。
彼女達の中から、新たな「希望の星」として憧れのハッケージに立つことが出来るのは誰か？ 友情、ライバル、プロ根性・・・・今、プログラマは新たな輝きを放つ！！！！
スピーディーなプロトタイピング！！ キュートなデバッグシーン！ 全宇宙を巻き込むスペースライブコーディング！
『愛に生き、会いに行くアイドルプログラマー達』の感動ライブストーリーが、今、幕を開ける！！
「星をこえて、みんなに会いに行くよ♡」

会員名簿じゃなイカ?

`@dif_engine` (1 章)

『数学の面白さは異常　文系は永遠に作者の気持ちでも考えてろよ』を貼りまくってるのが私なのではないか、と家族に疑われていて心苦しいです。

`@master_q` (2 章)

Linux でデジタルサイネージを作ってます。UI 設計楽しいですね!

`@xhl_kogitsune` (3 章)

千反田えるさんはきつねさんだと思います。アニメ版氷菓を見ていると耳とかしっぽとか見えますよね? 異論は認める。

`@yryozo` (4 章)

前回はなぜか Java VM の話になっちゃいましたが (笑)、今回はなんとか Scheme っぽい話が書けてほっとしてます。

`@tanakh` (5 章)

今回は氷菓を評価してしまいました。タイトルに負けました。イカ成分 0 ですいません。

..·S(S(KS)K)(S(S(KS)K)(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I))))))

「簡約(かんやく)!? λ(らむだ)カ(か)娘(むすめ)(算(さん))」

2012 年　8 月 11 日 コミックマーケット 82 初版発行
2012 年　8 月 31 日 第 2 版発行
2012 年 12 月　6 日 第 3 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, @dif_engine, @master_q,
　　@yryozo, @tanakh
表紙：@dif_engine
前回あらすじ・次回予告：@nushio
印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/


EVAL
APPLY