
簡約!
イカ娘
みんなで集まって
関数型プログラミングを
始めなイカ？！

目次


評価戦略

第0章

関数型イカ娘とは!

— @xhl_kogitsune

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

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

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

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

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

という電波を@xhl_kogitsune が受信したのが始まりでゲソ。

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

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

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

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

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

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

0.1　関数型イカ娘アニメ紹介

関数型イカ娘の活躍はアニメにも出てきたでゲソ。

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

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

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

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

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

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

第1章

β簡約！λ力娘

— @xhl_kogitsune

お前たち! わたしと一緒に λ しなイカ!

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

1.1 λしなイカ？

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

1.1.1 λ 式を書かなイカ?

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

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

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

$$x$$

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

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

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

これは、λ 式ではこんな感じに書くでゲソ。

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

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

$$(\lambda x.A)$$

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

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

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

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

A,B を λ 式とすると

$$(AB)$$

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

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

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

みたいに λ 抽象 2 回で書けて、

$$((gx)y)$$

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

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

カッコを書くのが面倒になったら、適宜省略できるでゲソ。たとえば

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

は

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

のように、

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

は

$$(gxyz)$$

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

λ 式

変数 x
λ 抽象 $(\lambda x.A)$
関数適用 (AB)

の三種類でゲソ。

1.1.2 β 簡約しなイカ?

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

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

β 簡約は、

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

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

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

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

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

β 簡約

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

1.1.3 α 変換しなイカ?

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

$$(\lambda x.x)$$

と書けるでゲソ。「x を受け取って i を返す関数」は

$$(\lambda x.i)$$

と書けるでゲソ。これを合わせると、「x を受け取って恒等写像を返す関数」は

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

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

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

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

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

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

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

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

正しいじゃなイカ!

こんな、変数名がかぶるようなコード書かないんじゃなイカ、と思うかもしれないでゲソが、複数の関数で同じ変数名を使って、それらを混ぜて β 簡約するとすぐかぶるようになるでゲソ。

α 変換

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

1.1.4 チャーチしなイカ?

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

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

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

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

ん、変数名を使って

0 f_0
1 f_1
2 f_2
3 f_3

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

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

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

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

何かではないでゲソ☆「何か」は、「f と x を受

[*1] プログラマは 0 から数えるでゲソ

け取って、何か 2 を返す関数」なので、

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

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

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

できたじゃなイカ!

これで $3+1$ を計算してみるでゲソ!

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

できたじゃなイカ!

チャーチ数

λ 式だけで表現できて演算もできる、自然数の表現方法でゲソ。

1.2 SKI しなイカ?

1.2.1 SKI じゃなイカ?

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

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

ここで x,y,z は SKI コンビネータの列を表す変数で、SKI 計算本体には λ 計算の時にはあった変数は全く出てこないでゲソ。全ては S,K,I,(,) の列でゲソ。λ 式と β 簡約の時みたいに、SKI 式で関数を書いて簡約を繰り返して計算をするでゲソ。カッコは λ 式の関数適用の時と同じように省略できるでゲソ。

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

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

正しいじゃなイカ!

SKI は私の好きな λ 式でも表現できるでゲソ!

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

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

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

やっぱり正しいじゃなイカ!

SKI 計算

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

変数一つない、美しいフォルムじゃなイカ!

1.2.2 IKMSM じゃなイカ?

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

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

[*2] エビみたいなものと思っておけばいいでゲソ

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

または

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

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

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

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

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

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

つまり、私は永遠に変わらないまま簡約を続ける存在、つまり神でゲソ!

このことは別の方法でも証明できるでゲソ。

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

という、これまた私を示す λ 式を考えてみるでゲソ。これは β 簡約すると

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

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

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

やっぱり Ω じゃなイカ!

1.2.3 SKI を書こうじゃなイカ!

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

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

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

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

$$cfgx \to f(gx)$$

まずは、感覚を掴むために感覚と試行錯誤でやってみるでゲソ。

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

$$c = \mathtt{S}$$

でもこれだと

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

となってしまうので違うでゲソ。

次に

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

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

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

[*4] イカ娘基準。でも λ 計算の下地があればイケるんじゃなイカ?

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

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

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

じゃあ

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

なんじゃなイカ?

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

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

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

のケースを考えるでゲソ。すると、

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

やったでゲソ! これで、

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

であれば OK ということが分かったでゲソ。まとめると、

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

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

1.2.4 SKI に変換しなイカ?

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

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

x,y は変数、M,N は λ 式、P,Q は変数交じりの SKI 式でゲソ。$T\{e\}$ が λ 式 e を SKI 式に変換する関数で、$A\{x,M\}$ は「変数 x を受け取って SKI 式 M を返す関数」の SKI 式を返す関数

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

でゲソ。T が λ 式を構文に従って分解して A に投げて、A が実際に SKI を吐くでゲソ。

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

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

となるので大丈夫でゲソ。

途中で x とかの変数が混じることがあるでゲソが、最後には消えるでゲソ。

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

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

1.2.5 SKI でチャーチ数しなイカ?

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

まずは 0 でゲソ!

$$0fx \to x$$

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

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

じゃぁ次は $0 = \mathtt{S}a$ みたいな形を考えてみるでゲソ!

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

$a = \mathtt{K}$ なら x になるじゃなイカ!

$$0 = \mathtt{SK}$$

実は $0 = \mathtt{K}a$ という形でもいけるでゲソ!

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

これが x になればいいので、$a = I$ じゃなイカ!

$$0 = \mathtt{KI}$$

次に 1 でゲソ!

$$1fx \to fx$$

これはそのまま I でいいんじゃなイカ!

$$1 = \mathtt{I}$$

次に 2 でゲソ!

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

$2 = \mathtt{S}a$ みたいな形になっているんじゃなイカ?

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

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

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

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

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

ダメじゃなイカ!

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

これもダメじゃなイカ!

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

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

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

できたじゃなイカ!

1.2.6 SKI でチャーチ数を自動生成しなイカ?

いちいち手で書いていたら日が暮れてしまうでゲソから、プログラムでチャーチ数を生成するでゲソ。制限なしの SK で最短のものとか考えると難しいでゲソから、ここではより小さいチャーチ数から大きいチャーチ数を組み立てることを考えるでゲソ! イカのような SKI 式で

チャーチ数どうしの演算ができて大きいチャーチ数を作れるでゲソ! n,m は自然数、n, m は SKI 式で書かれたチャーチ数でゲソ。実際に正しいかどうかは読者の演習問題とするでゲソ!

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

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

SKI 式まとめでゲソ!

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

1.3 有効活用しようじゃなイカ!

1.3.1 実用しなイカ?

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

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

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

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

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

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

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

1.3.2 λ 計算は何がいいでゲソ?

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

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

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

他にも、データとプログラムを統一的に扱う

[*6] あくまでイカ娘個人の感想であり、効能を保証するものではありません

ことができるでゲソ。λ 式は β 簡約によって実行されるプログラムでもあるし、β 簡約によって操作されるデータでもあるでゲソ。

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

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

第2章

数をつくらなイカ？

— @nushio

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

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

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

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

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

これら SKI 式の eval は次のように書けるでゲソ。

```
eval :: Expr -> Expr
eval expr = case expr of
  (Ap(Ap(Ap S f)g)x) -> let fx = eval (Ap f x)
                            gx = eval (Ap g x)
                        in eval (Ap fx gx)
  (Ap (Ap K x) _) -> eval x
  (Ap I x) -> eval x
  (Ap M x) -> eval (Ap (S&!I&!I) x)
  (Ap Succ x) -> case x of
                   (N n') -> N (n'+1)
                   _      -> Bottom "cannot increment non-integer"
  (Ap (N _) _) -> Bottom "cannot apply an integer"
  (Ap bottom@(Bottom _) _) -> bottom
  (Ap (Literal x) _ ) -> Bottom $ "cannot evaluate literal :" ++ x
  x -> x
```

実に素直に書けるじゃなイカ！

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

ではいよいよ candidate を実装するでゲソ。

```
solveSize = 256

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

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

```
        | weight x2 < weight x1  = [x2]
        | weight x2 == weight x1 = x2:xs
        | otherwise              = xs
```

このコードのキモは次の２行でゲソ！

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

第3章

関数型イカガール

— @tanakh

3.1 関数型イカガール第一話

高校一年の夏。

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

「ポイントフリーでゲソか？」

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

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

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

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

「ほら、これはなんでゲソ？」

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

「分からないでゲソか？ Squish パターンじゃなイカ」

そう言ってイカさんは少し体を起こす。微かにイカ臭い香りが漂った。

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

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

でも、もちろん僕は何も言わず、黙ってイカ墨パスタを食べていた。

3.2　関数型イカガール第二話

夏休み。

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

「配列操作でゲソか？」

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

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

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

3.2.1　永続データ構造

「まず一つ目は、永続データ構造[*1]による配列操作の実現でゲソ」

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

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

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

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

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

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

3.2.2　Linear Type

それはそうだろう。だけど純粋関数型言語でそれをどうやって実現するのか。

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

「STArray[*6]もあるでゲソ」

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

[*2] 子の高さに偏りのない木のこと

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

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

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

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

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

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

「型システムの拡張を考えるでゲソ」イカさんは言った。

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

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

何やら型に変な記号が付いている。

「!は Bang パターンでゲソ。正格性を表してるでゲソね」ふむふむ。

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

「Linear Type でゲソ！」

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

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

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

「一回しか参照されない型。これがどういうことかわかるでゲソか？」

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

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

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

「まだまだ、こんなもんではないでゲソ」

3.2.3 Stream Fusion

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

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

「Stream Fusion というテクニックがあるでゲソ」

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

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

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

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

「Haskell の書き換え規則は知ってるんじゃなイカ？」

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

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

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

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

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

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

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

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

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

「Purity はすごいでゲソね」

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

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

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

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

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

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

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

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

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

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

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

「じゃあ、Stream の具体的な実装を見てみることにするでゲソ」

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

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

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

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

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

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

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

うーむ、ややこしい。式がごちゃごちゃしてきた。

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

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

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

「ここで何か気づかなイカ？」

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

「その通りでゲソ！」イカさんは満足気に続けた。「このような case 式のネストは単純な case 式に書き換えることができるでゲソ。これを case-of-case transformation と呼ぶでゲソ。かっこいい名前でゲソ」

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

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

3.2.4 Recycling

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

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

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

「例えば配列のある要素を書き換える操作があったとするでゲソ」

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

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

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

「更新が連続するとどうなるでゲソ？」

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

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

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

「実際のところ、複数の更新処理をまとめて行う操作は提供されているでゲソ」

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

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

「そこで Recycling でゲソ」事もなげにイカさんは言う。

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

「内部的に mutable な配列を使用するでゲソ」

あっ、と僕は声を出してしまった。Stream の代わりに mutable 配列に対する操作に変換すればい

いのか。分かってしまえばあまりにも単純なことだ。

「例えばさっきの配列の更新の例でゲソが」

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

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

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

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

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

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

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

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

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

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

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

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

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

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

フンフンと鼻を鳴らしながらイカの少女は海の家の奥へと消えていった。

3.2.5 Deforesting と Supercompilation

帰りの電車にて。

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

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

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

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

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

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

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

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

  cout << maximum << endl;

  return 0;
}
```

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

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

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

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

```
$ g++ -O2 MostFreq.cpp
$ time ./a.out Fedora-15-i686-Live-Desktop.iso
3387261

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

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

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

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

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

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

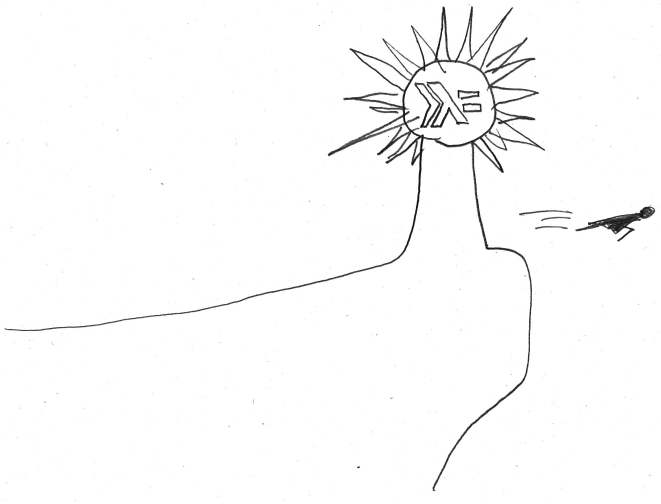
[*10] 高速モバイル通信の一つ http://www.uqwimax.jp/

[*11] 論文検索エンジン http://scholar.google.co.jp/

[*12] 関数型言語の国際学会 http://www.icfpconference.org/

3.3 参考文献

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



第4章

加速しなイカ？

— @nushio

4.1 夢のようなライブラリがあったじゃなイカ！

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

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

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

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

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

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

術じゃなイカ！

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

```
> cabal configure -fpcache
```

デフォルトでは$HOME/.cabal/share/accelerate-x.x.x.x/cache/に連番ファイル名で保存されるようでゲソ。これではどのファイルがどれなのか分からないじゃなイカ！ そこで例えば、Accelerate のソースにイカのように一行を追加することで、

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

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

4.2 GPU も、FPGA も、あるでゲソ！

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

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

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

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

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

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

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

ans :: Scalar Float
ans = run accAns

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

実行するでゲソ。

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

GPU 計算ができたじゃなイカ！

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

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

つぎに、型構築子 `Acc` は GPU 上で行いたい計算を表しているでゲソ。`Acc` な `Array` とふつうの `Array` の相互変換には、`use` と `run` という関数を使うのでゲソ。

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

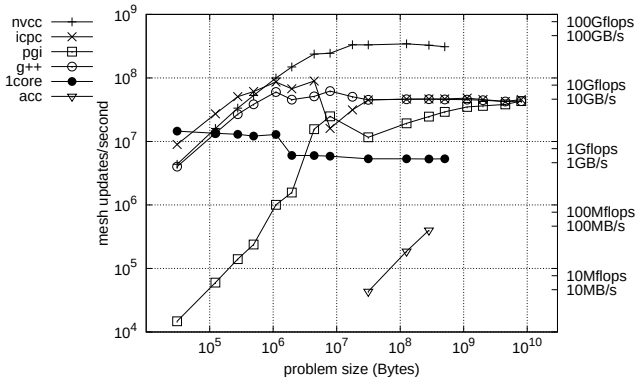
`Acc` の正体もカラクリも、私は知っているでゲソ！ `Acc` は実際には、「こういう計算をしたい」という要求を表す抽象構文木で、`Data.Array.Accelerate.Smart` モジュールで定義されているでゲソ。`main` 関数をイカのように書き換えて実行すると、

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

イカのように、中身の構文木が確認できるでゲソ。

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

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



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

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

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

4.3 実アプリの性能と向き合わなイカ？

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

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

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

これには理由があるでゲソ。私の実装はほぼあらゆる計算に `A.use . run :: Acc (Array sh e) -> Acc (Array sh e)` という関数を噛ませるようになっているでゲソ。この関数、意味は `id :: a -> a` にすぎないでゲソが、構文木を寸断して、計算を強制させる作用があるでゲソ。そんな事したら遅いし、折角の最適化が働かなくなるじゃなイカ！ でもこうでもしないとコード生成段階でエラーが出て動かないのでゲソ。`id` を挟んだら挙動が変わるって、それってつまり最適化がバグってるってことじゃなイカ！ ほとんど死んでるコードを動かして、生きてるフリをしてるだけなのでゲソ！ 性能出なくて当たり前じゃなイカ！

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

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

4.4 そんなの、私が許さないでゲソ！

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

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

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

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

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

[*2] そんなの不可能に決まってるじゃなイカ／人◕‿◕人＼

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

第5章

OSを侵略しなイカ？

— @master_q

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

5.1 HFuse でファイルシステムを侵略しなイカ？

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

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

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

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

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

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

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

```
$ cat mountdir/hello
Hello World, HFuse!
$ fusermount -u mountdir
```

うん、うまく動いているでゲソ。

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

5.2 ヲレヲレファイルシステムを作ってみるでゲソ!

さあお待ちかねでゲソ。ヲレヲレファイルシステムを作ってみるでゲソ!

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

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

```
$ ./GrassFs mount
$ cat helloworld.txt
wvwWwwwwWwwwwwwWWWWWwWWWWwwwwvwwwwWWWwwWwwWWWWwwwwWwwvwWWwWwwvwwWWwvwWWwwWW
WWWwwwWwwWWWWWwWWWWWWWWWwWWWWwWWWWwWWWWWWwWWWWWWWWWWWWwwwwwwwwWwWwWWWwwWWWww
wwwwwWWWWWwwwwwwwwWWWWWWwwwwwwwwWWWWWWWWWWWWWWWWWWWwwwwwwwwwwwwwwwwwwwwwWwwwwwww
wwwwwwwwwwwwwwwWwwwwwwwwwwwwwwwWwwwwwwwWWwwwwwwWWWwwwwwwWWWWWWwwwwwwwwwwwwwwwwwwwwwwW
wwwwwwwwwwWWwwwwWWWwwwwWWWWwWWWWwwwwwwwwwwwwwwwwwwwwWWWWWWWWWWWWwwwWWwWWWWWWWWw
WWWWWWWWWWWWWWWWWwwwwwwwwwwwwwwwwwwwWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwWWwwwwwwwwwwwwWWWwwwwwwwwWWWWwWWWWWwwwwwwwwwWWWWWWwwwwwwwwwwwwwwwwwwwWWWWW
WWwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwWWWWWWWWwwwwwwwwwwWwwwwwwwwwwwwwwwwwwwwwwwwwwww
wWWwwwwwwwwwwwwwwWWWwwwwwwwwwwwwwwWWWWwwwwwwwwwWWWWWwwwwwwwwwwwwwwwwwwwwwwWWWW
WWwwwwwwwwwwwwwwwwwwwwwwwww
$ cat helloworld.txt > mount/grassvm
$ sync;sync;sync
$ cat mount/grassvm
Hello, world!
```

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

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

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

```
            initGrassState r
```

HFuse の API は以下のようでゲソ。

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

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

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

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

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

> No creation (O_CREAT, O_EXCL) and by default also no truncation (O_TRUNC) flags will be passed to open(). If an application specifies O_TRUNC, fuse first calls truncate() and then open(). [a]

[a] http://fuse.sourceforge.net/doxygen/structfuse__operations.html#14b98c3f7ab97cc2ef8f9b1d9dc0709d から引用

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

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

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

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

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

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

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

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

5.4 ということで完成したでゲソ

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

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

5.5 この後は何を侵略するでゲソ？

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

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

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

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

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

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

会員名簿じゃないカ?

@xhl_kogitsune (0, 1 章)

きつねもふもふ。

@tanakh (3 章)

Haskell やって彼女ができました。

@nushio (2, 4 章)

宇宙物理学の研究をするうち、並列計算言語を作りたいという願いを掲げてエラい人と契約。先行研究は屍の山という絶望と、見るまに進歩していく関数型言語という希望の狭間で、科学少年の運命について思いを巡らせた結果、まどマギにドはまりしてしまう。私はほむほむ派です!!

@master_q (5 章)

プロニート。Debian メンテナ。もうオッサンなのに Haskell 初心者。前の仕事では NetBSD でコピー機作ってたみたい。pthread ライブラリメンテ地獄を体験し、マルチコア/メニーコア時代の組み込み開発のために参照透明な海を保護する必要性を感じている。

「簡約(かんやく)！ λ(らむだ)カ娘(かむすめ)」

2011 年 8 月 13 日 C80 初版発行
2011 年 8 月 24 日 第 2 版発行
2012 年 10 月 31 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, @tanakh, @nushio, @master_q
表紙：尾住(オズ) 様
挿絵：@nushio
印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/


EVAL
APPLY