簡約!? イカ娘 (二期)

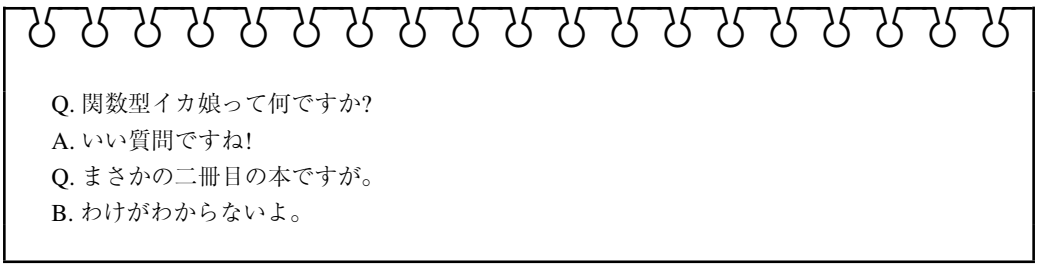
目次

第0章

まえがき

— @xhl_kogitsune

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. まさかの二冊目の本ですが。
B. わけがわからないよ。

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」に続く、二冊目の関数型イカ娘の本でゲソ。アニメ 2 期も執筆時点で放送中だし、いいタイミングじゃなイカ!?

関数型言語で地上を侵略しなイカ!

第1章

クラスタリングしなイカ？

— @dekosuke

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

というわけで、クラスタリングをやってみなイカ？

1.1 アルゴリズムとライブラリを決めなイカ？

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

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

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

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

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

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

```
cabal install kmeans-vector
```

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

[*1] もちろん私も関数型言語クラスタに含まれているでゲソ

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

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

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

[*5] kmeans-vector ライブラリ作者調べ

転載してもいいんじゃなイカ？[*6]

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

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

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

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

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

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

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

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

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

```
kmeans k points = kmeansAux points pgroups
  where pgroups = partitionPoints k points
```

説明をするでゲソ。K-Means は下記のアルゴリズムでゲソ。

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

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

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

1.2 CSV をパーズしなイカ？

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

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

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

この関数の適用結果はこんな感じでゲソ。

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

1.3 実際にやってみるでゲソ

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

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

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

```
main = do
  cs <- readFile "iris.data" --ファイルを読んで文字列して
```

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

それぞれの関数の詳細はソースコード[*7]を見ればいいんじゃなイカ？

実行結果はイカのようになったでゲソ。

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

Versicolor と Virginica の間でデータの交差があるので、正解率は 9 割程度でゲソ。

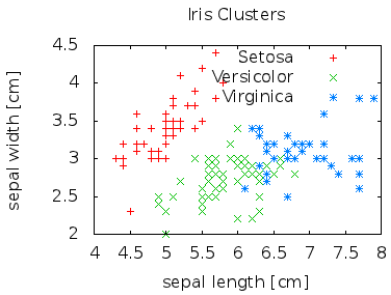
え、なんで３つのクラスタが”Setosa”, ”Versicolor”, ”Virginica”のどのラベルに対応しているかどうかわかるかって？ それは決め打ち・・・データ数が少ないときは手作業の誘惑には耐えられないもの・・・[*8]

無事データをクラスタリングできたので、データを表示する作業に入るでゲソ！ データのプロットには gnuplot を使うでゲソ。gnuplot の一行スクリプトを書くでゲソ！

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

[*7] http://github.com/dekosuke/iriscluster

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



という感じで、きれいにデータがクラスタに分けられたでゲソ！

上図のプロットでは４次元のデータ中２次元しか使ってないので一見外れ値があるように見えるでゲソが、実際は４次元空間の一番近いクラスタ中心の周りにデータが集まっているはずでゲソ。これにてクラスタリング完了でゲソ！

実行結果で示したように、正解率は 9 割弱でゲソ。つまり上のグラフの１割ぐらいは間違って・・・ゲフンゲフン！

1.4 まとめでゲソ

今回は手始めに Iris Dataset の分類を行い、クラスタリングの威力を見たでゲソ。今後はソーシャルグラフなどのさまざまな方向への侵略が期待されるでゲソ。

クラスタリングは、距離の計れる任意のデータに応用可能でゲソ。クラスタに対する意味づけは読者次第・・・なので無限の世界が広がっているでゲソ！

1.5 参考文献でゲソ

- 神嶌敏弘「クラスタリングとは」`http://www.kamishima.net/jp/clustering/`
- Wikipedia「K 平均法」`http://ja.wikipedia.org/wiki/K` **平均法**
- Hackage - kmeans-vector `http://hackage.haskell.org/package/kmeans-vector-0.1.1`
- UCI Machine Learning Repository ”Iris Dataset” `http://archive.ics.uci.edu/ml/datasets/Iris`
- 今回のクラスタリングのソース `http://github.com/dekosuke/iriscluster`

1.6 kmeans-vector の license でゲソ

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

All rights reserved.

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

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

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

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

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

第2章

Copilotへの希望と絶望の相転移

— @master_q

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

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

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

2.1 Copilot はイカが？

そんなことに悩んでいたおり

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

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

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

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

ような繰り返しを表現するでゲソ。このようにして書かれた Copilot DSL は C 言語に変換できる

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

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

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

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

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

- ソースコード: Fibonacci.hs

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

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

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

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

- ソースコード: Run.hs

```
module Main where

import Copilot.Language
import Fibonacci

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

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

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

- ソースコード: Compile.hs

```
module Main where

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

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

- ソースコード: main.c

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

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

int main()
{
  int i;

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

- ソースコード: Makefile

```
run:
  runghc Run.hs

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

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

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

.PHONY: run clean
```

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

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

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

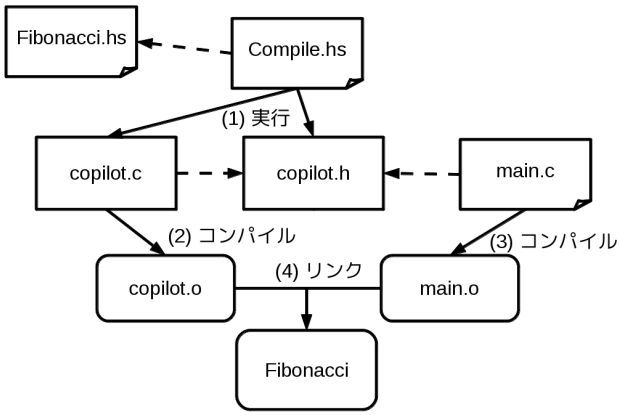
お次はコンパイル実行でゲソ。

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

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

2.2 Copilot DSL と C 言語コードの融合でゲソ

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



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

- main 関数を作りループで step 関数 (copilot.c に実体が作られる) を繰り返し呼び出す
- Fibonacci.hs から呼び出される関数、ここでは fib_out 関数をスタブとして作る

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

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

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

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

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

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

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

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

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

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

で、結局 Stream というのはなんなんでゲソ？

```
*Copilot.Language.Stream> :i Stream
data Stream $a where
  Append :: Typed a =>
            [a] -> (Maybe (Stream Bool)) -> (Stream a) -> Stream a
  Const :: Typed a => a -> Stream a
  Drop :: Typed a => Int -> (Stream a) -> Stream a
  Extern :: Typed a => String -> Stream a
  ExternFun :: Typed a => String -> [FunArg] -> Stream a
  ExternArray :: (Typed a, Typed b, Integral a) =>
                 String -> (Stream a) -> Stream b
  Local :: (Typed a, Typed b) =>
           (Stream a) -> (Stream a -> Stream b) -> Stream b
  Var :: Typed a => String -> Stream a
  Op1 :: (Typed a, Typed b) =>
         (Core.Op1 a b) -> (Stream a) -> Stream b
  Op2 :: (Typed a, Typed b, Typed c) =>
         (Core.Op2 a b c) -> (Stream a) -> (Stream b) -> Stream c
  Op3 :: (Typed a, Typed b, Typed c, Typed d) =>
         (Core.Op3 a b c d) -> (Stream a) -> (Stream b) -> (Stream
                                                              c) -> Stream d
```

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

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

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

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

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

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

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

2.4 Wii リモコンで遊ぶでゲソ

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

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

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

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

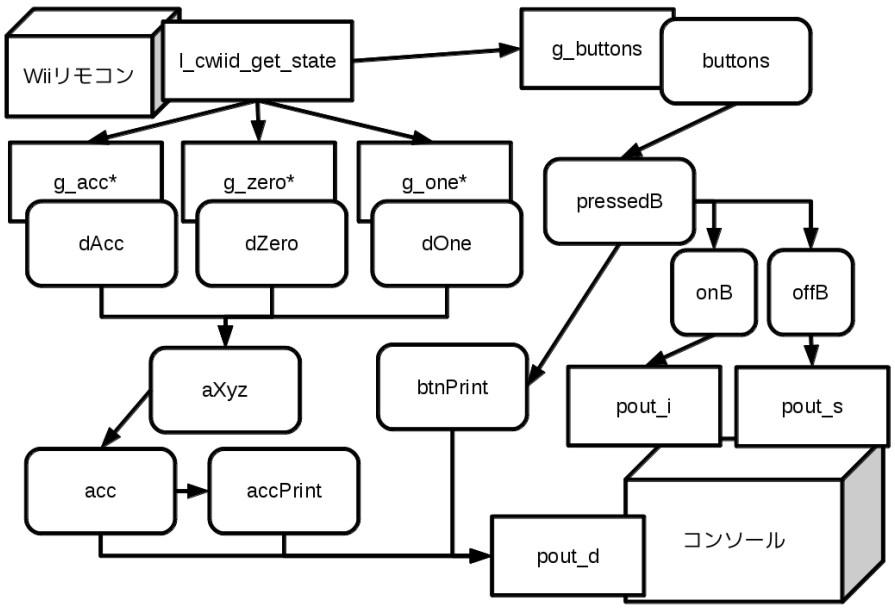
```
{-# LANGUAGE RebindableSyntax #-}
module Wiimote where

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

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

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

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

buttons :: Stream Word16
buttons = externW16 "g_buttons"

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

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

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

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

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

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

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

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

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

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

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



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

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

これで上の図の説明のスタートに辿り着いたでゲソ。図中の四角は C 言語の関数や変数、丸四

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

角は Copilot DSL の Stream でゲソ。データの流れとしては、Wii リモコンから加速度などの情報をポーリングで取り出し、Copilot DSL で加工して、最終的にコンソールにプリントされるでゲソ。図の Stream の接続を順番に説明するでゲソ。

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

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

2.5 libc のないところでも操縦できるでゲッソ!

さぁそろそろお遊びの時間は終わりでゲソ。当初の目的を覚えているでゲソか？「libgmp,libc,libpthread のようなライブラリのない環境でも参照透明な海を泳ぎたい」だったでゲソ。願ったからには作らなければならないでゲッソ! 対象として STM32 のような組み込み用途

の CPU 上で動かしても良かったでゲソが、色々な制約上 [*7] NetBSD の bootloader 上で Copilot DSL を使ったプログラミングをしてみるでゲソ。。。。ということでなんとかできたゲソ。 [*8] NetBSD の bootloader は src/sys/arch/i386/stand/boot にソースコードが置かれていて、その中の boot2.c にある boot2() 関数が bootloader の main 関数でゲソ。この boot2() 関数の中に Copilot を呼び出す step 関数をループで書けば、 Copilot DSL を bootloader 上で使えるでゲソ。 [*9]

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

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

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

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

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

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

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

[*7] 単に筆者の根性がないだけでゲソ……

[*8] https://github.com/master-q/study_copilot/tree/master/NetbsdBoot

[*9] カーネル／ VM Advent Calendar 7 日目 http://d.masterq.net/?date=20111207#p01

[*10] https://github.com/master-q/study_copilot/blob/master/NetbsdBoot/README.md

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

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

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

resetCounter :: Stream Bool
resetCounter = lGetChar == 114

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

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

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

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

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

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

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

int32_t l_getchar(void)
{
```

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

```
  static int32_t ret = 'A';
  return ret++;
}

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

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

int main()
{
  int i;

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

  return 0;
}
```

さっそく実行でゲソ。

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

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

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

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

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

  {
    static uint8_t __scheduling_clock = 2;
    if (__scheduling_clock == 0) {
      __assertion_checks(); __r0();  /* copilot.sample_fun_l_getchar_0 */
      __assertion_checks(); __r1();  /* copilot.sample_fun_l_getchar_1 */
      __assertion_checks(); __r2();  /* copilot.sample_fun_l_getchar_2 */
      __assertion_checks(); __r3();  /* copilot.sample_fun_l_getchar_3 */
      __assertion_checks(); __r4();  /* copilot.sample_fun_l_getchar_4 */
      __scheduling_clock = 6;
    }
    else {
      __scheduling_clock = __scheduling_clock - 1;
    }
  }
--snip--
void step()
{
  copilot();copilot();copilot();copilot();copilot();copilot();copilot();
```

```
}
```

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

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

2.6 けっきょく使えそうなんでゲソ？

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

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

実はあるのでゲソ。かるく検索するだけでも以下のような試みが見つかるでゲソ。

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

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

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

「全ての `POSIX` 互換インターフェイスを生まれる前に消し去りたい。すべての宇宙、過去と未来の全ての `POSIX` 互換インターフェイスを、この手で」
「そんな祈りが叶うとすれば、それはインターフェイス変更なんてレベルじゃない。`C` 言語そのものに対する反逆だ」
「希望を信じたプログラマを、私は泣かせたくない。最後まで笑顔でいて欲しい。それを邪魔する `C` 言語なんて、壊して見せる。変えて見せる」

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

[*13] GHC 本体と統合するのはいろいろ難しいみたいです。。。

第3章

加速しなイカ？！

— @nushio

3.1 これまでのあらすじ

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

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

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

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

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

— そして、現状はというと・・・

3.2 進化しなイカ？

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

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

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

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

「だるいでゲソ！ それに三行じゃとても足りないでゲソ！」

「私からもお願いできるかしら」

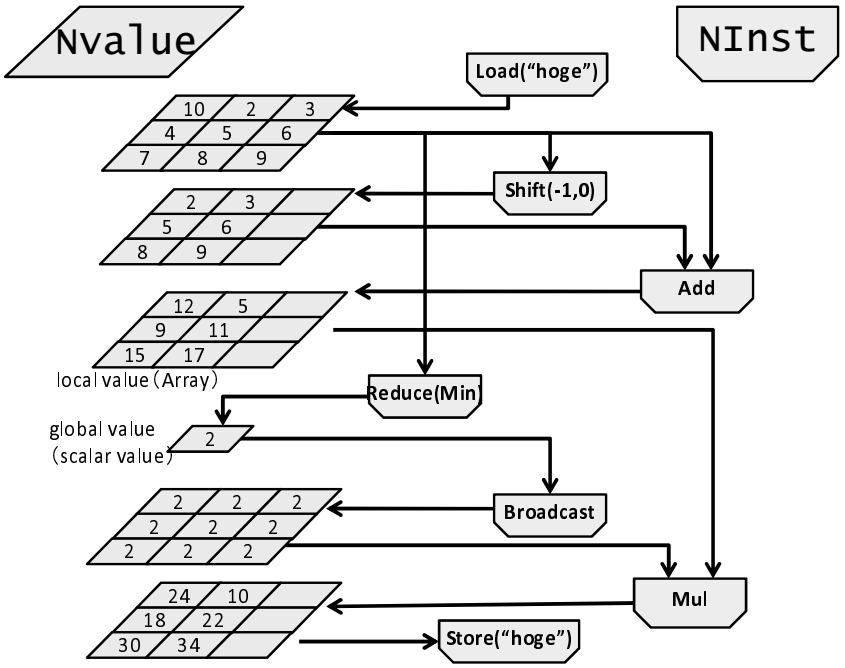


図 3.1: Orthotope Machine(OM) のデータフローグラフ。OM には複数の Kernel がある。Kernel とは一塊の計算 (ふつうにメソッドみたいなもん) であり、ひとつの Kernel がひとつのデータフローグラフに対応する。OM の変数には、Kernel の中にスコープが限られる Temporal 変数と、名前がついていて Kernel 間を生き延びる Static 変数とがあり、各 Kernel の目的は Static 変数を Load 命令で読み込んで、Store 命令でその新しい状態を書き出すことである。OM の他の命令の意味は、イメージしろ！


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


「行いたい計算を表現するデータフローグラフを、Builder モナドというフロントエンドを使って、Orthotope Machine という仮想機械の命令列として生成します。(c.f. 図 3.1)」


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

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


「まさにそこがポイントなのでゲソ！ 図 3.2 のような簡単なプログラムをとってもわかるように、計算の途中結果を配列としてメモリに置く (Manifest) か、必要に応じて再計算する (Delayed) か、実装の選択肢があるじゃなイカ？ このへんは Haskell の多次元配列ライブラリ、REPA [4] を参考にしたでゲソ。」

— 図 3.3 のように、この Manifest or Delayed の選択はグラフの分割結果、そして生成されるコー

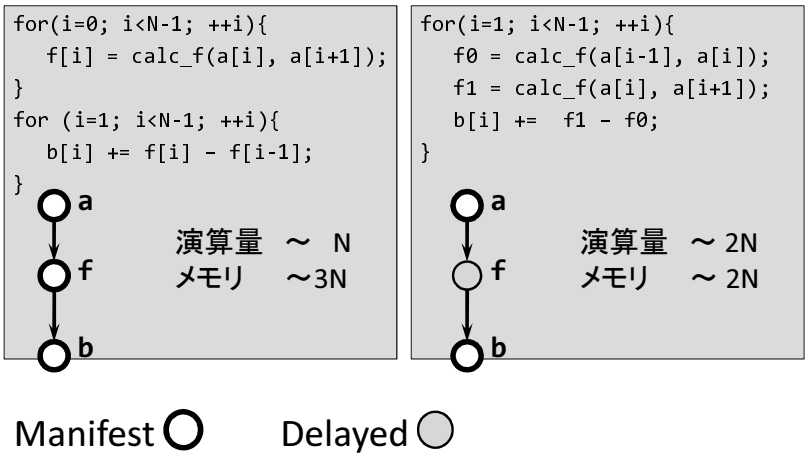


図 3.2: メモリを使ってでも演算量を減らすか、演算量を増やしてでもメモリの容量や帯域を節約するか？ Manifest or Delayed トレードオフ。

ドを大きく左右する。これが Paraiso が自動チューンしてくれるポイントの一つだ。

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

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

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

 「え、焼きなまし最強じゃなイカ？」

 「GA が焼きなましよりもうまくいったことが無いでゲソ」

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

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

のだ。[*1]

3.3 生存戦略しなイカ？

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

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

— まだ問題がある。我々はクラスター計算機の空きノードで自動チューニングを行うことを強いられているんだ！ 通常の連結熱浴法は、熱浴の並列数が固定で、一世代ごとに同期を取る。これに対し「遺伝子」のスコアは生成したコードのベンチマーク。スコアが出るまでの所要時間は遺伝子によって違うから、同期をとっていては待ち時間が生じる。おまけに空きノードの数も刻々変化するわけで、熱浴の並列数が固定というのも悩ましい。何とかして、個々の遺伝子の評価を非同期処理にできなイカ？

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

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

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

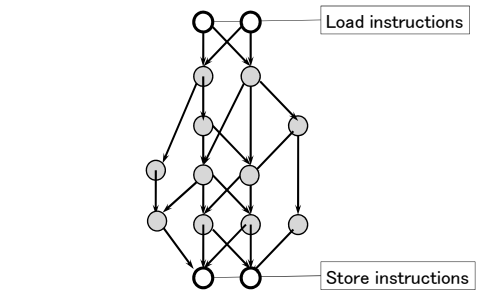
 「ひとつの個体 S についてベンチマークは 30 回程度走らせることになっているでゲソ。こうして得られたスコアの平均 $\mu(S)$ と標準偏差 $\sigma(S)$ を DB に記録しているでゲソ。」

— 標準偏差を記録しておくのって、そんなにも大切な事なの？

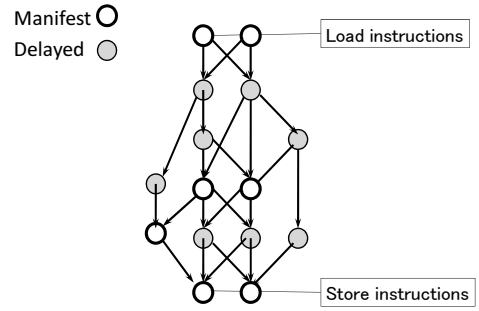
 「十分な並列度があれば、出し惜しみせずに統計を溜めることだってできる。それが並列焼きなましの強みでゲソ。」

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

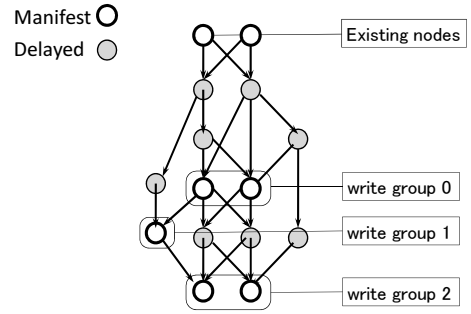
[*1] こんなのを論文より先に同人誌で出しちゃっていいのだろうか・・・だが、それがいい。



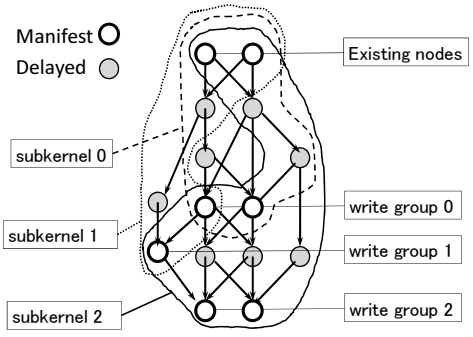
(1) Kernel には初期状態で Manifest にすべきノードが存在する。



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



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



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

図 3.3: Paraiso のコード生成メカニズム。

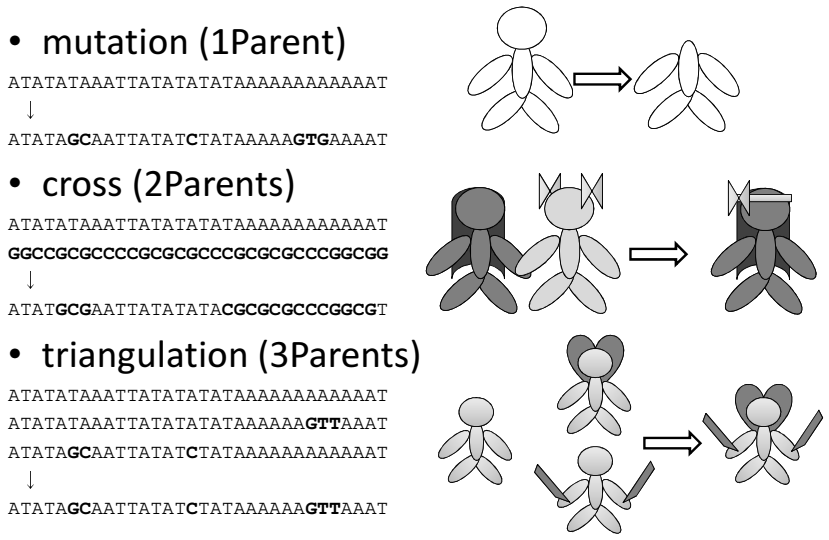


図 3.4: 新個体を創り出す三つの方法。絵が酷いですね。

確率は

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

に比例するのでゲソ。ここで S_0 は現時点で成績がトップの個体でゲソ。」

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

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

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

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

— では次に、新個体を作り出す方法について説明してくれなイカ？

「図3.4 のように、新個体を作る方法は三つ採用されているでゲソ。まず一つ目は突然変

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

表 3.1: 三角合成のビット演算表でゲソ！

異。ひとつの遺伝子を『ドロー』し、ランダムに書き換えて、別の個体をつくり出す操作でゲソ。」

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

 「二つ目は交尾。つまりまどほ m」

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

 「 三つ目は三角合成。これは独自の概念でゲソ。」

— なぜ独自の概念を導入する必要があったのだろうか。

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

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

 「だから違うのでゲソ。そういうアプローチは取らないでゲソ。全体のグラフに埋め込んだまま、いわば in vivo で各部分グラフをチューンし、その結果を合成してやれればよいじゃなイカ？」

— なるほど。具体的にはどういう操作をするのだろう。

 「基本的なアイデアはこうでゲソ。まず先祖 (Base) となるべき個体と、そいつから別々に進化した二つの個体 (Primary, Secondary) をランダムに選択するでゲソ。それから、『親に比べていづれかの子が変化していたら、その変化を取り入れる』という方針でそれらを合成。つまり表 3.1 のようなビット演算を施すでゲソ。これにより、Primary と Secondary に別々の改良点が生じていたら、それらを合成できるでゲソ。」

— でも、それだと家系図を辿るのが大変じゃなイカ？

 「そこでもうひと工夫でゲソ。単に 3 つの遺伝子を『ドロー』し、スコア順にソートして

低い方から Base, Secondary, Primary とすることにしたのでゲソ。」

— そんな手抜きで大丈夫か？

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

3.4 種をまかなイカ？

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

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

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

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

 「人間がコード書いたらこのへんメモリに置くだろ常考」

 「といえる個体を 3 つばかし作ってみたでゲソ (c.f. 表 3.2)。その中でも一番速かったのに *Shinatsuhiko* という名前をつけたでゲソ。

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

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

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

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

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

[*2] DRY 原則がとことん貫かれているという意味で

[*3] 好きにアノーテートしても決して意味が変わらないという意味で

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

表 3.2: 手でアノテートしたコードの性質と速度。生成されたコードの行数、その中の subKernel の数、流体マス目の数 N に対して何倍のメモリを消費するか、単精度と倍精度のコードの速度、など。速度の単位は一秒間に何百万メッシュ計算できたか (Mega cell update per second) で測っている。それにしても、神々産みし始原の女性に息長き風王神 (izanamishinatsuhiko)。中々よい名前だ。日本神話を愛する人ごめんなさい。

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

表 3.3: 複数の自動チューニング実験の統計。初期に与えた個体は、GA-1 と GA-S1 では *Izanami* であり、そのほかは *Shinatsuhiko* である。各実験について、採用した精度、初期にあたえた個体のスコア、チューニングにかけた実時間（単位：分）生成した個体の総数とそのうちスコアの平均値 $\mu(S)$ が最も高かった個体の ID、そしてその最高個体のスコア（単位：Mcups）を示す。Mcups は一秒あたり 10^6 メッシュを計算する程度の能力である。

3.5 統計しなイカ？

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

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

「さて、表 3.3 がこれまでに行った自動チューニング実験の統計でゲソ。一日あたりざっと一万個体を生成できるでゲソ。そして、数日放置することで、*Izanami* は *Shinatsuhiko* なみの速度になり、*Shinatsuhiko* はもとの２倍近い速度になったでゲソ。

この性能がイカほどのものか、本当のところは完全手書きでチューンしたコードと較べてみないと分からないでゲソが、同じ GPU で流体コード、というカテゴリーなら、Schive [6] らにより

単精度で C2050 一枚あたり 30Mcups（ただし彼らは 3 次元かつ計算量は多いスキームを採用）、Asunciónа [1] らにより単精度で GTX580 一枚あたり推定 6.8Mcups（こちらは 2 次元）などの性能が報告されているので、いい線いってるんじゃなイカ？」

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

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

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

参考文献

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

[2] FRIGO, M., AND JOHNSON, S. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216231.

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

[4] KELLER, G., CHAKRAVARTY, M., LESHCHINSKIY, R., PEYTON JONES, S., AND LIPPMEIER, B. Regular, shape-polymorphic, parallel arrays in haskell. *ACM SIGPLAN Notices 45*, 9 (2010), 261272.

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

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

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

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

第4章

Java VM上で tail call しなイカ？

— @yryozo, @xhl_kogitsune

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

4.1 Java VM と関数型言語は相性が悪いでゲソ？

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

Java VM の上での実装には、おおまかに 2 つの方法があるでゲソ。

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

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

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

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

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

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

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

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

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

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

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

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

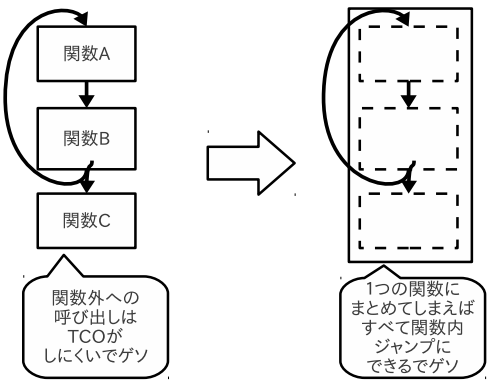
4.1.1 複数の関数をひとつにまとめるでゲソ

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

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

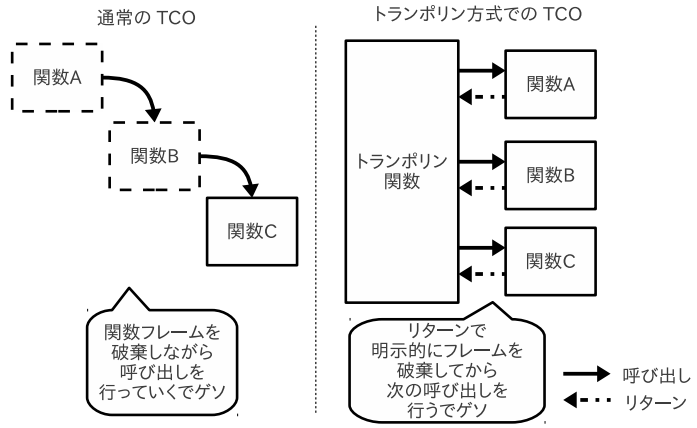
```
boolean isOdd(int n){
    int m;
isOdd:  // (odd? n)
    if( n == 0 ) return false;
    else { m = n - 1; goto isEven; } // call to even? (- n 1)
isEven: // (even? m)
    if( m == 0 ) return true;
    else { n = m - 1; goto isOdd;  } // call to odd?  (- m 1)
}
```

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



4.1.2 トランポリンするでゲソ

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



Java で書くと例えばこんな感じになるでゲソ。

```
public class OddEvenByTrampoline{
    static interface Continuation{
        public Object invoke();
    }
    static class Odd implements Continuation {
        public Odd(int n) { _n = n; }
        public Object invoke(){
            if( _n==0 ) return new Boolean(false);
            else        return new Even(_n-1); // 元は tail call
        }
        private int _n;
    }
    static class Even implements Continuation {
        public Even(int m) { _m = m; }
        public Object invoke(){
            if( _m==0 ) return new Boolean(true);
            else        return new Odd(_m-1); // 元は tail call
        }
        private int _m;
    }
    public static Object trampoline(Object o){
        while(o instanceof Continuation)
            o = ((Continuation)o).invoke();
        return o;
    }
    public static void main(String[] args){
        System.out.print(trampoline(new Even(1000 * 1000 * 1000)));
    }
}
```

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

4.1.3 トランポリンを改良するでゲソ

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

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

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

```
public class OddEvenByTrampoline2{
    static interface Continuation{
        public Object invoke();
    }
    static class Odd implements Continuation {
        public Odd(int n) { _n = n; }
        public Object invoke(){ return isOdd(0,_n); }
        private int _n;
    }
    static class Even implements Continuation {
        public Even(int m) { _m = m; }
        public Object invoke(){ return isEven(0,_m); }
        private int _m;
    }
    public static Object isOdd(int depth, int n){
        if( depth>=1000 ){ return new Odd(n); }
            // tail call 段数が上限超えたらトランポリンへ戻る
        if( n==0 ) return new Boolean(false);
        else       return isEven(depth+1,n-1);
    }
    public static Object isEven(int depth, int m){
        if( depth>=1000 ){ return new Even(m); }
            // tail call 段数が上限超えたらトランポリンへ戻る
        if( m==0 ) return new Boolean(true);
        else       return isOdd(depth+1,m-1);
    }
    public static Object trampoline(Object o){
        while(o instanceof Continuation)
            o = ((Continuation)o).invoke();
        return o;
    }
    public static void main(String[] args){
        System.out.print(trampoline(new Even(1000 * 1000 * 1000)));
    }
}
```

4.1.4 処理系を改良するでゲソ

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

これでみんなにも関数型は Java VM ごときに負けたりはしないということが分かったんじゃなイ

カ？ それでもまだ納得できないという読者諸君のために、実際に関数型の侵略の前にはどんな抵抗も無意味だという所を見せつけてやるでゲソ。

4.2 Java VM 上の関数型言語実装を紹介するでゲソ!

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

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

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

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

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

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

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

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

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

結果をざっとまとめるとこんな感じになるでゲソ♪[*4][*5]

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

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

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

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

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

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

*b プログラムの書きかたによっては×だったでゲソ。

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

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

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

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

4.2.1 Scheme

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

Kawa

TCg `http://www.gnu.org/software/kawa/`
o[W 1.11 (2010-11-11)

トランスレータ方式による Scheme 実装でゲソ。

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

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

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

また、`--full-tailcalls` オプションをつけると、general tail call elimination が有効になるでゲ

ソ[*6]。具体的にはイカのように define で相互再帰を書くとトランポリンを用いた実装に自動変換されたでゲソ。

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

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

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

Bigloo

TCg	http://www-sop.inria.fr/indes/fp/Bigloo/
o[W	3.7a (2011-09)

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

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

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

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

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

SISC

TCg	http://sisc-scheme.org/
o[W	1.16.6 (2007-02-27)

JScheme

TCg	http://jscheme.sourceforge.net/jscheme/main.html
o[W	7.2 (2006-02-04)

[*6] http://www.gnu.org/s/kawa/Restrictions.html

[*7] http://www.gnu.org/s/kawa/Method-operations.html

[*8] http://www.gnu.org/s/kawa/Evaluating-Scheme-expressions-from-Java.html

[*9] bigloo.jheap がないというエラーが出た場合は、-lib-dir /path/to/bigloo/lib/bigloo/3.7a のようなオプションで bigloo.jheap のあるパスを指定する必要があるかもしれないでゲソ

[*10] "Bigloo is not able to compile all the tail recursive call without stack consumption (however, most of the tail recursive calls are optimized by Bigloo and don't use stack activation frames).", Bigloo User manual 27.3 http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo-28.html#Limitation-of-the-JVM-back-end.

JAKLD

TCg	`http://ryujin.kuis.kyoto-u.ac.jp/~yuasa/jakld/index-j.html`
o[W	

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

4.2.2 Common Lisp

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

Armed Bear Common Lisp (ABCL)

TCg	`http://common-lisp.net/project/armedbear/`
o[W	1.0.0 (2011-10-23)

CLforJava

TCg	`http://clforjava.org/`
o[W	20091212 (2009-12-12)

Jatha

TCg	`http://jatha.sourceforge.net/`
o[W	2.8 (2007-04-25)

これらの JVM 上での実装では末尾再帰でも相互末尾再帰でもスタックオーバーフローだったでゲソ。

4.2.3 OCaml

OCaml は ML 系言語の一つでゲソ[*12]。パターンマッチやバリアントが便利でゲソ。ツリー操作とかが書きやすいので、コンパイラの実装にも使われることが多いでゲソ。たとえば、C Intermediate Language (CIL) [4][*13] は OCaml で書かれた C 言語のパーザと中間表現上での解析器でゲソ。

本家ネイティブ実装である OCaml, OCamlopt は末尾再帰や tail call optimization に対応しているので tail call し放題でゲソ。

OCamlJava

TCg	`http://ocamljava.x9c.fr/`
o[W	1.4 (2010-02-06)

Java VM 上の実装としては OCamlJava があるでゲソ。OCamlJava では、直接の末尾再帰はループに変換されるでゲソ。一般の tail call はそのまま Java のメソッド呼び出しに変換されるのでスタックオーバーフローするでゲソ。これは、ドキュメント (アーカイブ内の `ocamljava-bin.pdf` など) に未対応と書いてあるでゲソ。

[*11] SBCL など、ネイティブ実装のいくつかは tail call optimization を行うでゲソ。

[*12] OCaml 公式サイト: `http://caml.inria.fr/`

[*13] `http://cil.sourceforge.net/`

4.2.4 Funnel

Funnel

TCg	`http://lamp.epfl.ch/funnel/`
o[W	Release 9 (2002-03-14)

改良型のトランポリンが実装されているランタイムでゲソ。

デフォルトでは末尾再帰は最適化されるようでゲソが、一般の tail call は最適化されないので相互再帰とかを書くと落ちるでゲソ。

`-tail-calls:ret` または `-tail-calls:exn` オプションをつけると、4.1.3 節で説明した改良型のトランポリンが有効になるでゲソ。それぞれのオプションは、トランポリンまで戻る時にメソッドリターンを繰り返すか、例外で抜けるかに対応するでゲソ。これをつけると相互再帰でも落ちなくなるでゲソ!

4.2.5 Clojure

Clojure

TCg	`http://clojure.org/`
o[W	1.3 (2011-09-23)

Clojure は Java VM 上で実行することを考えて設計された言語でゲソ。

Clojure では、末尾再帰を明示的に書いてループを表現するために、`loop`, `recur` というものが用意されているでゲソ。これを使って書けばちゃんとループが書けるでゲソ。普通に直接末尾再帰を書くとスタックオーバーフローで落ちてダメでゲソ。こんな感じで使うでゲソ。

```
(defn fib [n0 a0 b0]
  (loop [n n0 a a0 b b0]
    (if (<= n 0)
        a
        (recur (- n 1) b (mod (+ a b) 100000007)))))
```

`loop` で、末尾再帰で呼ばれる関数を定義するようなものでゲソ。`[n n0 a a0 b b0]` というのは、この `loop` の引数は `n`, `a`, `b` で、その初期値は `n0`, `a0`, `b0` である、という意味でゲソ。そして `recur` で `loop` を呼び出すでゲソ。`recur` の呼び出しが末尾再帰になっていることはコンパイラがチェックするでゲソ。スタックを消費せずにループが書けるのは `recur` だけでゲソ[*14]。

一般の tail call 最適化はサポートしていない[*15] ので、トランポリンを使うしかないでゲソ。トランポリンを楽に書くための記法も用意されているでゲソ [*16]。『プログラミング Clojure』によると、遅延評価やシーケンスなどのトランポリン以外の方法を推奨しているようでゲソ。

4.2.6 Scala

[*14] "Note that recur is the only non-stack-consuming looping construct in Clojure. " – `http://clojure.org/special_forms`

[*15] "Since Clojure uses the Java calling conventions, it cannot, and does not, make the same tail call optimization guarantees.", – `http://clojure.org/functional_programming`

[*16] 実例はこの章の末尾に載せてみたでゲソ

Scala

TCg	http://www.scala-lang.org/
o[W	2.9.1 (2011-08-29)

Scala も Java VM 上での実行を考えて設計された新しい言語でゲソ。

末尾再帰は最適化される模様でゲソ。相互再帰とかは、Clojure 同様トランポリン等を使う必要があるでゲソ [17]。

4.3 Java VM 自体も侵略するでゲソ

それでは次に Java VM の内側も侵略してみるでゲソ！

Da Vinci Machine Project というプロジェクトを知っているでゲソか？ これは OpenJDK を Java 以外の言語からも上手く使えるようにするためのプロジェクトでゲソが、実は Java VM で TailCall を可能にするためのパッチが既に作られているのでゲソ [6]。OpenJDK といえば、Java における事実上の標準で 2011 年 7 月には Java VM のリファレンス実装にも指定されたという、まさに Java VM 界の本丸でゲソ。つまり、このパッチさえあれば Java VM への侵略は完了したも同然じゃなイカ！

```
Da Vinci Machine Project
<http://openjdk.java.net/projects/mlvm/subprojects.html>
```

4.3.1 TailCall 機能付きの Java VM をコンパイルしてみるでゲソ

それではさっそく Java VM で TailCall できるようにしてみるでゲソ。なお、イカの手順は Ubuntu 11.10(x86-32) で試したものなので、他の環境の人は適宜読み替えてほしいでゲソ。(ちなみに、今のところ TailCall の実装は x86-32 版しか存在しないでゲソ…。)

0. まずは、ビルドに必要な環境を整えるでゲソ。とりあえず、イカのパッケージを入れておけばいいんじゃなイカ。

```
$ sudo aptitude build-dep openjdk-7
$ sudo aptitude install openjdk-6-jdk
## libjibx は 1.1 系列でないとイカんようでゲソ。(sourceforge から落とすときは
## 1.1.6a を選ぶでゲソ)
$ sudo aptitude install libjibx1.1-java
### パッケージを入れた場合はリンクが張られないので自力で張るでゲソ…。
$ cd /usr/share/java
$ sudo ln -s jibx-run-1.1.jar jibx-run.jar
$ sudo ln -s jibx-bind-1.1.jar jibx-bind.jar
##  Mercurial の設定も必要でゲソ。hgforest extension を入れるでゲソ。
$ HgForestRoot=~/hgext/   # ここのディレクトリ名は適当でいいでゲソ
$ hg clone https://bitbucket.org/pmezard/hgforest-crew/overview/ \
   ${HgForestRoot}
$ cat >> ~/.hgrc << EOF
    [extensions]
    forest = ${HgForestRoot}/forest.py
```

[17] こちらも実例を章末に載せてみたでゲソ

```
EOF
```

1. 次にパッチ自体とパッチ対象のソースコードをダウンロードするでゲソ。jdk7-b59 というバージョンが必要でゲソ。(ReadMe には b50 と書かれていたりするでゲソが、これは間違いでゲソ)

```
$ mkdir davinci; cd davinci  # ここのディレクトリ名は適当でいいでゲソ！
$ hg fclone -r jdk7-b59 http://hg.openjdk.java.net/jdk7/jdk7 sources
$ hg fclone http://hg.openjdk.java.net/mlvm/mlvm patches
```

2. 次に、パッチディレクトリとソースコードディレクトリの間にリンクをはるでゲソ。

```
$ bash patches/make/link-patch-dirs.sh sources patches
## 一応確認しておくでゲソ。左端に出る数字が同じであれば問題ないでゲソ。
$ ls -il patches/hotspot/series sources/hotspot/.hg/patches/series
```

3. 適用するパッチを選ぶでゲソ。今回はもちろん TailCall パッチを選ぶでゲソよ。ちなみに、buildable と testable はダメなパッチを弾くためのルールでゲソ。つけておいた方が変なものが入らなくていいでゲソ。

```
$ export davinci=$(pwd) guards="buildable testable tailc-lazy tailc"
$ sh patches/make/each-patch-repo.sh hg qselect --reapply $guards
```

4. 実際にパッチを適用するでゲソ。

```
$ sh patches/make/each-patch-repo.sh '(! hg qunapplied | read something) || \
  hg qpush -a'
```

5. いくつか修正しないとダメそうな箇所があるので、適当に直しておくでゲソ。

```
## execvpe という関数名は最近の unistd.h の中身と被っているようなので変えておくでゲソ。
$ sed -i 's/execvpe/azunyan_kawaii/g' sources/jdk/src/solaris/native/java/\
  lang/UNIXProcess_md.c
## ALSA 関係のリンク設定がおかしいようなので直しておくでゲソ。
$ sed -i 's/LDFLAGS/EXTRA_LIBS/' sources/jdk/make/javax/sound/jsoundalsa/\
  Makefile
## awt 関係で使っている定数名がおかしいようなので直しておくでゲソ。
$ sed -i 's/X_ShmAttach/XShmAttach/' sources/jdk/src/solaris/native/sun/\
  awt/awt_GraphicsEnv.c
```

6. それでは、実際に make するでゲソ。

```
## === 環境変数を設定しておくでゲソ ===
## JDK6 があるパスを指定するでゲソ
$ export ALT_BOOTDIR=/usr/lib/jvm/java-1.6.0-openjdk/
## libjibx があるパスを指定するでゲソ
$ export ALT_JIBX_LIBS_PATH=/usr/share/java
## 以下は指定するとコンパイルが並列実行されて速くなるでゲソ（数字は CPU のコア数と
## 相談でゲソ）。
$ export HOTSPOT_BUILD_JOBS=4
$ export PARALLEL_COMPILE_JOBS=4
## 新しい Linux カーネル（3.x）には対応してないのでバージョンチェックを回避するでゲソ
$ export DISABLE_HOTSPOT_OS_VERSION_CHECK=ok
## === ここまでで環境変数の設定は完了でゲソ ===
$ cd sources/
## SAN 値をチェック、ではなくビルド準備が整っているかどうかをチェックするでゲソ
$ make sanity
## さあ、ビルドするでゲソ
$ make
```

7. ビルドは終わったでゲソか？ 完了したら sources/build/linux-i586/bin/に java コマンドや javac コマンドが出来ているはずでゲソ。

4.3.2 動くかどうか試してみるでゲソ

それでは、実際に動かしてみるでゲソ。TailCall パッチには、Java 言語の拡張 (というか Java コンパイラの拡張でゲソね) と、Java VM の拡張の 2 つが入っているでゲソ。

まず Java 言語の方でゲソが、 'goto' という予約語 [*18][*19]を使って末尾呼び出しにしたい箇所を指定できるようになっているでゲソ。

```
public class TailCallExample {
  public static int tailcaller(int x) {
    if (x==0) return x+1;

    return goto tailcaller(x-1);  // この呼び出しが末尾呼び出しになるでゲソ
  }
}
```

どうでゲソ？ まぁ、あまり Java には興味がないから、ここはどうでもいいでゲソね。

次に Java VM についてでゲソが、バイトコード上では 196(=0xc4) という 1 バイトのバイトコード

[*18] Scheme の設計者である Guy Steele 博士も、末尾呼び出しとは引数を渡せる goto のようなものだ、と言ってたでゲソね

[*19] 余談でゲソが、goto は昔から Java の予約語なのに今に至るまで全く意味が定義されていなくて誰にも使えない予約語でゲソ (言語仕様には、C++ と間違って goto と書いてしまった人にコンパイラがエラーを吐くために予約語にした、みたいなことが書かれているでゲソ)。そのおかげで、このパッチは互換性を壊さずに末尾呼び出し機能を導入できているでゲソが、名前空間という貴重なシンボルの海をこんな風に浪費するのはイカがなもんでゲソ？ ほとんど予約語を持たない Scheme の美しさを見習うべきじゃなイカ？

が末尾呼び出しを表すプレフィックスになるでゲソ[*20]。そして、この直後に置かれたメソッド呼び出し命令 (invoke 系命令) が末尾呼び出しになるんでゲソ。これは、.Net とほぼ同じ方式でゲソね。

```
$ javap -c TailCallExample  # バイトコードを逆アセンブルするコマンドでゲソ
...
public static int tailcaller(int);
  Code:
   0:  iload_0
   1:  ifne   8
   4:  iload_0
   5:  iconst_1
   6:  iadd
   7:  ireturn
   8:  iload_0
   9:  iconst_1
   10: isub
   11: bytecode 196  // この直後に置かれた invoke 系の命令が末尾呼び出しになるでゲソ
   12: invokestatic  #2; //Method tailcaller:(I)I
   15: ireturn
}
```

そして、実行する時は普通に java コマンドでいいでゲソ (当たり前でゲソが、さっき自分でビルドした java コマンドを使うんでゲソよ)。ただし、今のところはコマンドラインオプションで-XX:+TailCalls とつけないと TailCall 機能は有効にならないようでゲソ。

```
java -XX:+TailCalls TailCallExample
```

ちなみに、書いたプログラムによっては TailCallException という例外が出ることがあるでゲソ。この場合は-XX:+TailCallsStackCompression というコマンドラインオプションをつければ出なくなるでゲソ[*21]。

4.3.3 Java 以外の言語からも使ってみようじゃなイカ

というわけで、とりあえず Java からは TailCall 機能が使えることは分かったでゲソ。とはいえ、Java VM の機能なら Java 以外の言語からでも使えるはずじゃなイカ。というわけで、とりあえず手近にあった Scala と Clojure からも使えるようにしてみるでゲソ。今回は「普通に末尾呼び出しで書

[*20] ちなみに、この 196 というバイトコードはパッチを当てていない Java VM でも利用されていて、wide 命令 (wide プレフィックス) と呼ばれているのでゲソ。後ろについたロード/ストア命令のオペランド長を 1byte から 2byte に伸ばす効果があるんでゲソが、invoke 系命令には効果がないので、パッチを当てない状態では invoke 系の前についた時はエラーになるでゲソ。

[*21] これは Java VM の protection domain という概念と関連しているでゲソ。Java VM はそれぞれのクラスをどこからロードしたかを覚えていて、これを使って「アプレットのようなインターネットからダウンロードしたクラスにはセキュリティ上重要な機能は使わせない」みたいな成魚を実現してるんでゲソが、このせいで末尾呼び出しでも関数フレームを削除できないことがあるのでゲソ (セキュリティチェック時には、その時点でスタックに積まれている全関数フレームの protection domain の共通部分を確認するのでゲソ。フレームを消してしまうとそのフレームに関する protection domain 情報もなくなってしまうので、このチェックがきちんと行えなくなってしまうんでゲソ)。というわけで、実行時に protection domain が異なるクラス間でメソッドが末尾呼び出しされると、デフォルトでは TailCallException を投げて異常終了してしまうでゲソ。ただし、-XX:+TailCallsStackCompression を指定していると、例外を投げる代わりに末尾呼び出しを諦めて、普通に関数フレームを積んでメソッド呼び出しするようになるんでゲソ。

くと、普通に末尾呼び出し最適化してくれる」状況を目指すでゲソ。

Scala で「普通に末尾呼び出しで書いた」even/odd の相互再帰テストプログラム

```
object MutualTailCallTest {
  def isEven(n : Int) : Boolean = {  if (n == 0) true   else isOdd(n-1)   }
  def isOdd(n : Int)  : Boolean = {  if (n == 0) false  else isEven(n-1)  }

  def main(args : Array[String]) : Unit = {
    var start = System.currentTimeMillis
    for (i <- 1 to 10) {  isEven(1000 * 1000 * 1000)  }
    var end = System.currentTimeMillis
    print((end - start)/1000.0); println("[sec]")
  }
}
```

Clojure で「普通に末尾呼び出しで書いた」even/odd の相互再帰テストプログラム

```
(ns mutualtailcalltest
  (:gen-class))

(declare my-even? my-odd?)
(defn my-even? [^long n]  (if (zero? n) true  (my-odd?  (unchecked-dec n))))
(defn my-odd?  [^long n]  (if (zero? n) false (my-even? (unchecked-dec n))))

(defn -main [& args]
  (let [start (System/currentTimeMillis)]
    (do
      (dotimes [_ 10]
        (my-even? (* 1000 1000 1000)))
      (. System/out print (/ (- (System/currentTimeMillis) start) 1000.0))
      (. System/out println "[sec]"))))
```

といっても、Scala や Clojure のコンパイラに手を入れるのは行儀がよくない気がするでゲソ[*22]。というわけで、今回は scalac や clojurec が出力したクラスファイルの方をいじってみようじゃないカ。

scalac や clojurec の出力したクラスファイルを見たところ、どちらも関数呼び出しとリターンについては素直に Java VM のメソッド呼び出し命令とリターン命令にしているようでゲソ。

scalac によるコンパイル結果

```
public boolean isEven(int);
  Code:
    ...
    11:  iconst_1
```

[*22] 作者注: いや面倒くさかっただけです。

```
   12:  isub
   13:  invokevirtual   #18; //Method isOdd:(I)Z
   16:  ireturn

public boolean isOdd(int);
  Code:
   ...
   11:  iconst_1
   12:  isub
   13:  invokevirtual   #24; //Method isEven:(I)Z
   16:  ireturn
```

clojurec によるコンパイル結果

(正確には、以下は my_even の方だけ。
clojurec では関数 1 つ毎に 1 つのクラスを生成する。それぞれの関数の処理は、
生成されたクラスの invoke() メソッドが呼ばれると実行される模様。
以下では、invokePrim の 29byte 目の clojure/lang/IFn.invoke の呼び出しが
my_odd の呼び出しに対応。)

```
public final java.lang.Object invokePrim(long);
  Code:
   ...
   19:  checkcast       #54; //class clojure/lang/IFn
   22:  lload_1
   23:  invokestatic    #60; //Method clojure/lang/Numbers.unchecked_dec:(J)J
   26:  invokestatic    #64; //Method clojure/lang/Numbers.num:(J)Ljava/lang/..
   29:  invokeinterface #68,  2; //InterfaceMethod clojure/lang/IFn.invoke:(...
   34:  areturn

public java.lang.Object invoke(java.lang.Object);
  Code:
   0:   aload_0
   1:   aload_1
   2:   checkcast       #74; //class java/lang/Number
   5:   invokestatic    #78; //Method clojure/lang/RT.longCast:(Ljava/lang/....
   8:   invokeinterface #80, 3; //InterfaceMethod clojure/lang/IFn$LO.invokePri
   13:  areturn
```

というわけで、先人の英知 [*23] を参考に、イカのような方針で末尾呼び出しを見つけて tailcall プレフィックスをつけるプリプロセッサを作ってみたでゲソ！

[*23] 詳細は、[6] を見るでゲソ！

1. invoke 系の命令 [*24]の直後に return 系の命令 [*25] があれば、それが末尾呼び出しでゲソ。
2. ただし、synchronized 指定されているメソッドの中では、どんな呼び出しも末尾呼び出しにはならないでゲソ。(synchronized 指定がある場合、暗黙のうちにメソッドに入るとロックがとられ、メソッドから出るとアンロックされるでゲソ。ということで、事実上全ての return 命令の直前にロックを解除する monitorexit 命令があるのと同義なので、invoke 系命令の直後に return という実行シーケンスがありえなくなるでゲソ。)
3. それから、例外ハンドリングの対象になっている範囲中の呼び出しも、末尾呼び出しにはならないでゲソ。(例外処理の範囲内 (Java で言う所の try{} の内部) では、例外が起きたときには例外ハンドラに飛ばないといけないので、末尾呼び出しでフレームを削ってしまってはイカんでゲソ。Scheme で dynamic-wind 中で末尾呼び出しにならないのと同じでゲソ。)

今回は Javassist [2] というライブラリを使ってお手軽作成したでゲソ [*26]。とりあえずプレフィックスの挿入部分はこんな感じでゲソ。

```
...
for (final CodeIterator ci = ca.iterator(); ci.hasNext();) {
  final byte[] tailprefix = {(byte)0xc4};

  final int line1 = ci.next();
  final int opcode1 = codes[line1] >= 0 ? codes[line1] : codes[line1]+0x100;
  if (! ci.hasNext()) { break; }
  final int line2 = ci.lookAhead();
  final int opcode2 = codes[line2] >= 0 ? codes[line2] : codes[line2]+0x100;

  if ((opcode1 == Opcode.INVOKEINTERFACE ||
       opcode1 == Opcode.INVOKEVIRTUAL   ||
       opcode1 == Opcode.INVOKESPECIAL   ||
       opcode1 == Opcode.INVOKESTATIC)
      &&
      (opcode2 == Opcode.RETURN  ||
       opcode2 == Opcode.ARETURN ||
       opcode2 == Opcode.IRETURN ||
       opcode2 == Opcode.LRETURN ||
       opcode2 == Opcode.FRETURN ||
       opcode2 == Opcode.DRETURN)) {

      ci.next();
      ci.insert(line1, tailprefix);
  }
...
```

やってることは、バイトコードをイテレートしていって、invoke 系の命令の後ろに return 系の命

[*24] invokevirtual、invokestatic、invokeinterface、invokespecial の 4 つでゲソ。正確には invokedynamic というのもあるでゲソが、今回のソースコードのバージョン (jdk7-b59) ではそこまで対応していないでゲソ...。

[*25] return、areturn、ireturn、lreturn、freturn、dreturn の 6 つでゲソ

[*26] 作者注: というか、お手軽過ぎで上の 1~3 のうち 1 のルールしか実装してません。お手軽と言うか手抜きですね...。

令が続くシーケンスを見つけたらその前にプレフィックスを挿入するのでゲソ。見たまんまでゲソね。

ちなみに、実際に使う場合は -javaagent というオプションでプリプロセサを指定するのでゲソ[*27]。この方式では、Java VM がクラスファイルをロードするときにメモリ上のイメージだけを書き換えるので、ディスク上のクラスファイルは変更しないのでゲソ。勝手にクラスファイルを壊したりはしないので安心でゲソよ。

```
## scalac でコンパイルするでゲソ
$ scalac MutualTailCallTest.scala
## scala コマンドに、ビルドした java コマンドの位置を指定してやるでゲソ
$ export JAVACMD=~/workspace/davinci/sources/build/linux-i586/bin/java
## 同じく、プリプロセッサや他のオプションを指定するでゲソ
$ export JAVA_OPTS="-XX:+TailCalls -XX:+TailCallsStackCompression \
  -Xverify:none -javaagent:tailcallpreprocessor.jar=MutualTailCallTest"
## 実行してみるでゲソ！
$ scala MutualTailCallTest
```

4.3.4 性能を調べてみるでゲソ

それでは、Scala と Clojure がどれだけ上手く相互再帰できるようになったかを確かめてみようじゃないカ。次の表が結果でゲソ[*28]。

言語	通常時	パッチおよびプリプロセス	(参考) 明示的にトランポリン化
Scala 2.9.0 (on Java 7b59)	死亡確認！	4.872[sec]	117.557[sec]
Clojure 1.3.0 (on Java 7b59)	〃	67.291[sec]	407.451[sec]
Java (Java 7b59)	〃	4.762[sec]	-
(参考) OCaml(OCamlopt) 3.12	6.048[sec]		

通常時の欄に書いてある通り、普通に末尾呼び出しで書いて普通に Java VM 上で動かすと、これまではスタックオーバーフローで死んでしまっていたのでゲソ。が、Java VM を侵略してしまえば、死なないどころかかなりの速度で even/odd の相互再帰ができるようになったでゲソ。

参考までに、OCamlopt でネイティブコンパイルした OCaml プログラムとも競争させてみたでゲソ。x86 のマシン語を直接出力できる OCamlopt は関数型言語コンパイラの中でも速い方でゲソが、侵略後の Scala の方が速いでゲソ。これはもう Scala(スカラ) を超えた・・・「スクルト」と呼んでもいいんじゃないカ[*29]！ これで Scala も一人前の関数型言語の仲間入りでゲソ。

[*27] 作者注: 実際にはこのままだと verify が通らないので-Xverify:none も一緒につけています。これはパッチが Java6 以降の新しいクラスファイルフォーマット用の verify 関数にしか対応してないのに、scalac や clojurec が Java5 という 1 つ古いクラスファイルフォーマットで出力してくれるからです (これは file コマンドで確認できます)。scalac については、ProGuard というツールでクラスファイルをバージョンアップすれば-Xverify:none なしでも通ったのですが、clojurec の方は ProGuard でうまくバージョンアップできないみたいで、今のところ-Xverify:none なしだと上手く動いてません...。

[*28] 作者注: 測定環境は次の通りです。・ CPU : Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz、・ OS : Ubuntu 11.10 (x86-32) 。テスト自体は、上のプログラム例で書いた通り、1G 回の even/odd 相互再帰を 10 セット行いました。また、Java VM の JIT コンパイラは server コンパイラ (C2 コンパイラ) を使っています。

[*29] って、スクルトだと防御力の上昇量は逆に減っちゃたりするでゲソが...。ちなみに、これをスクル「ド」と読み間違えた 2 級神 1 種限定萌えな読者には猛省を促すでゲソ！

一方の Clojure。お前に足りない物、それは (ry そして何よりも速さが足りない！ でゲソ。今回のテストプログラムだと、Clojure は関数を全て clojure.lang.IFn という関数オブジェクトにして、関数呼び出しを invoke() というインターフェースメソッド呼び出しにしているようでゲソが、invoke() の引数が全部 Object 型になっているので、整数みたいなプリミティブ型だと Object 型へのキャストと元の型へのキャストが入ってしまって遅いようでゲソ[*30]。

ちなみに、一目見て分かる通りトランポリンの性能がかなり悪く出ているでゲソが、これは even/odd がかなり極端な例ということもあるでゲソ (ほとんど関数呼び出ししかないでゲソからね)。普通のプログラムではもっとオーバーヘッドは少ないはずでゲソ。

4.3.5 今後はどうなるでゲソ？

さて、ここまで読んだ忍耐力あふれる読者諸君は、この TailCall パッチの今後が気になってきた頃じゃなイカ？ 残念ながら TailCall パッチは、最新のソースコード (jdk7-b147) には対応できていないのでゲソ (2011/12/10 現在)。が、2011 年 10 月に開かれた JavaOne 2011 という Java のお祭りでは、次々バージョンの Java に TailCall を取り込むという話が (その他たくさんの話に紛れて一瞬だけ) 出ていたようでゲソ。本当に正式採用になるかどうかはまだ分からないでゲソが、最近の関数型言語の盛り上がり型を見ると可能性は十分あるんじゃなイカ？。というわけで、関数型が Java VM を完全侵略する日はもう間近に迫ってきているのでゲソ。

参考: Scala で「明示的にトランポリンで書いた」even/odd の相互再帰テストプログラム

```
import scala.util.control.TailCalls._

object MutualTailCallTest2 {
  def isEven(n : Int) : TailRec[Boolean] = {
    if (n == 0) done(true)   else tailcall(isOdd(n-1))
  }
  def isOdd(n : Int)  : TailRec[Boolean] = {
    if (n == 0) done(false)  else tailcall(isEven(n-1))
  }

  def main(args : Array[String]) : Unit = {
    var start = System.currentTimeMillis
    for (i <- 1 to 10) {  isEven(1000 * 1000 * 1000).result  }
    var end = System.currentTimeMillis
    print((end - start)/1000.0); println("[sec]")
  }
}
```

参考: Clojure で「明示的にトランポリンで書いた」even/odd の相互再帰テストプログラム

```
(ns mutualtailcalltest2
  (:gen-class))

(declare my-even? my-odd?)
```

[*30] 作者注: 自分のテストプログラムの書き方が悪いだけのような気もしていたり...。一応 type hint はつけてみたんですが...。

```
(defn my-even? [^long n]  (if (zero? n) true  #(my-odd?  (unchecked-dec n))))
(defn my-odd?  [^long n]  (if (zero? n) false #(my-even? (unchecked-dec n))))

(defn -main [& args]
  (let [start (System/currentTimeMillis)]
    (do
      (dotimes [_ 10]
        (trampoline my-even? (* 1000 1000 1000)))
      (. System/out print (/ (- (System/currentTimeMillis) start) 1000.0))
      (. System/out println "[sec]"))))
```

付録：諸君らが愛してくれた `OCamlopt` は負けた！ なぜだッ！？ でゲソ

セ^h オキャム ル「バ、バカな、き、きさまはスカラだろ！？ ち、ちがうのか！？」
スカラ「ちがうな・・・、オレは、超（スーパー）スカラだ！！」（ドヤ顔で）
オキャム ル「超スカラ・・・！？ な、なんだそれは！？」
スカラ「いちいち説明するのもめんどうだ、てめえでかってに想像しろ」

って、「かってに想像しろ」だとこのコーナーいきなり終わってしまうでゲソ (汗。結論から先に言うと、今回の速度差は単にインライン展開の差でゲソ。
OCamlopt は確かに x86 のネイティブコードを出力してくれる強力なコンパイラでゲソが、最適化は (少なくとも今のところは) あんまり頑張っていないでゲソ。今回も even/odd の相互再帰に対するインライン展開はなかったでゲソ。(つまりプログラムで書いた通りに毎回関数呼び出しが出ていたのでゲソね。一応 -inline オプションも付けてみたけどダメだったでゲソ...。)
それに対して、今回使った HotSpot という Java VM が Scala の even/odd に対して施したインライン展開は実に 10 段。単純計算すると末尾呼び出し時のジャンプのオーバーヘッドが 1/10 になっているわけで (いや、これは単純過ぎるでゲソが) さすがに勝てないでゲソ。ちなみに読者の中には、(この作者と同じように) Java VM 上では関数呼び出しが全て virtual 呼びなので、ダイナミックディスパッチが必要だしインライン展開も出来なくて遅い、と思ってた人はいないでゲソか？ これは一般には正しいでゲソが、実際に JIT コンパイルする段階で isEven() や isOdd() という名前のメソッドが 1 つしか存在しなければ、飛び先がそこだというのは確定的に明らかなので、ダイナミックディスパッチも不要だしインライン展開もできてしまうでゲソ。Java VM 言語だから関数呼び出しは遅い、とは限らないのでゲソね。(もちろんこういうやり方をすると、JIT 生成した後で isEven() や isOdd() を実装したサブクラスがダイナミックロードされたら JIT をやり直すという手間も出てしまうでゲソが...。)
最後に余談でゲソが、インライン展開の差とだけ言われると GCC なんかが気になる人もいるんじゃなイカ？ 試しに C で even/odd 相互再帰を書いて gcc -O3 でコンパイルしてみると、、、、くぁ w せ drftgy ふじこ lp(結果は各自見てみるでゲソ。ちなみに今回私が使ってみたのは gcc4.6.1 でゲソ)。こういうえげつない行為をいともたやすく行ってしまう、それが D4C、、、ならぬ GCC でゲソね。とはいえ手元の環境では、-O2 だと単に TCO するだけで引数はスタック渡しなので OCamlopt にもボロ負け、さらに-O1 以下では TCO さえしないのでスタックオーバーフロー死と、安定感が全くないのも GCC でゲソ。所詮は C 言語、関数型言語のような過度な期待をしてはいけないのでゲソ。

参考文献

[1] ABELSON, H., DYBVIG, R., HAYNES, C., ROZAS, G., ADAMS, N., FRIEDMAN, D., KOHLBECKER, E., STEELE, G., BARTLEY, D., HALSTEAD, R., OXLEY, D., SUSSMAN, G., BROOKS, G., HANSON, C., PITMAN, K., AND WAND, M. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11* (1998), 7–105. 10.1023/A:1010051815785.

[2] CHIBA, S. Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM*

OOPSLA'98 Workshop on Reflective Programming in C++ and Java (Oct. 1998).

[3] JONES, S. P., RAMSEY, N., AND REIG, F. C–: A portable assembly language that supports garbage collection. In *IN INTERNATIONAL CONFERENCE ON PRINCIPLES AND PRACTICE OF DECLARATIVE PROGRAMMING* (1999), Springer Verlag, pp. 1–28.

[4] NECULA, G., MCPEAK, S., RAHUL, S., AND WEIMER, W. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, R. Horspool, Ed., vol. 2304 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2002, pp. 209–265.

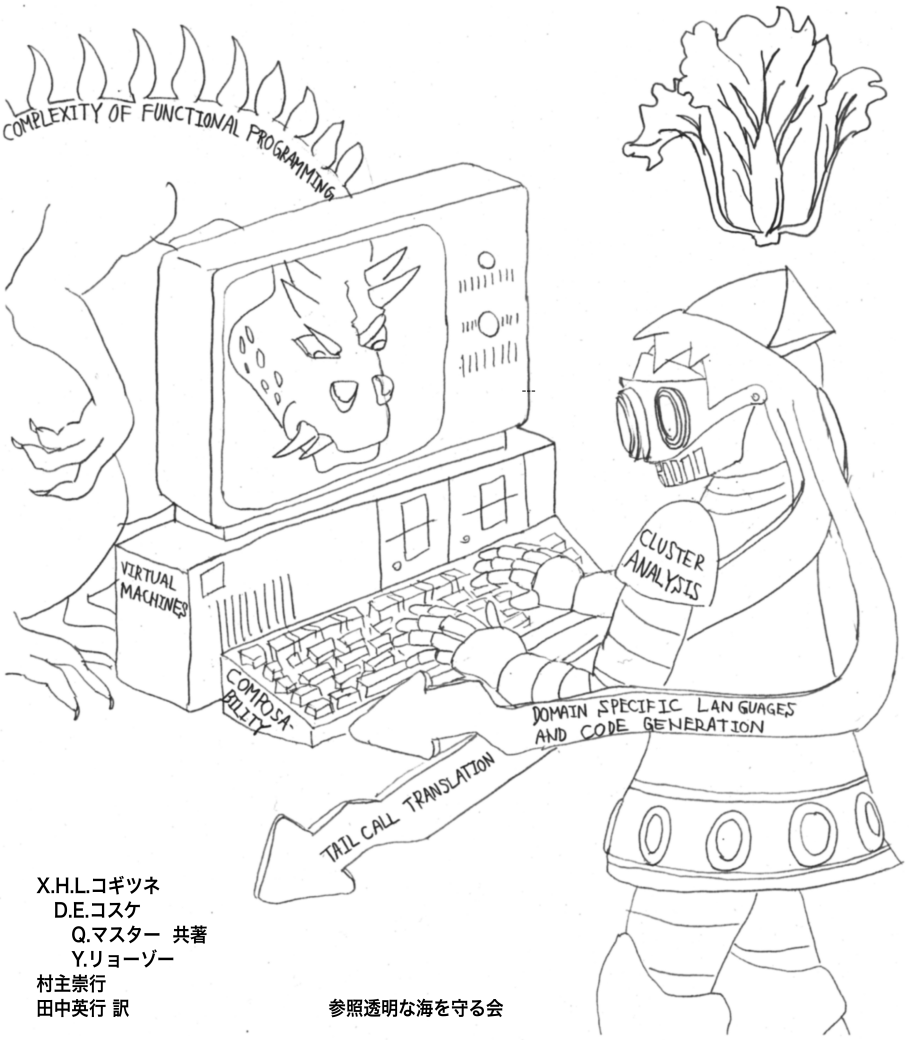
[5] SCHINZ, M., AND ODERSKY, M. Tail call elimination on the Java virtual machine. In *In Proc. ACM SIGPLAN BABEL' 01 Workshop on Multi-Language Infrastructure and Interoperability* (2001), Elsevier, pp. 155–168.

[6] SCHWAIGHOFER, A. Tail call optimization in the Java HotSpot VM. Master's thesis, Johannes Kepler Universität Linz, 2009.

[7] SERPETTE, B. P., AND SERRANO, M. Compiling Scheme to JVM bytecode: a performance study. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2002), ICFP '02, ACM, pp. 259–270.

イカムスメ II

簡約・簡約・簡約・簡約・簡約・簡約



X.H.L.コギツネ
D.E.コスケ
Q.マスター　共著
Y.リョーゾー
村主崇行
田中英行 訳

参照透明な海を守る会

第5章

関数型イカガール第3話

— @tanakh

この作品はフィクションであり、実在する人物、地名、団体とは一切関係ありません。

---

12 月。冬の海に僕はいた。夏の賑わいは見る影もなく、寂しい海にはもちろん他には誰もいない。海の家は夏の終わりと共に閉店し、あのイカの少女も姿を消した。

それでも僕は週末が来るたびここに足を運ぶ。あの時の胸の高鳴りを忘れることができなくてーー

最近の僕を悩ませているのは、圏論だ。正確には圏論そのものではなく、代数学とプログラミングの関わりについてだ。圏論のことを考えているときはとても楽しくて、それだけで時間を潰してしまうことも少なくないのだけども、プログラミングの時にどことなく圏論の影がちらつく Haskell は勉強を始めた当初は手こずったりしたものだった。それが Haskell、あるいは関数型パラダイムのの敷居を上げているのではないか、少なくとも心理的障壁になっているのではないかと思ってしまう。

「そんなの決まっているじゃなイカ！」

突然の声に僕は顔を上げた。そこにはあのイカの少女のくりくりとした大きな目があった。彼女は夕日を背に、僕のノートを覗き込んでいた。いつからそこにいたのだろう。どうやら僕は考え事をすると周りが見えなくなるようだ。しかしそんなことよりも、イカ娘にまた会えたことが嬉しかった。

「それは Composability のためでゲソ」

そんな僕にはお構いなしにイカ娘は続ける。

「Composability はプログラムを自由に組み合わせられる性質でゲソ。プログラムの Composability は、安心してプログラムを書くためにとても重要なことでゲソ。基本的なことじゃなイカ」

Composability という概念は聞いたことがないでもなかったが、それがそんなに重要なものだとは考えていなかったわけで。

「コンビネータ、って聞いたことないでゲソか？ 人間の世界では小学校で習うって聞いたでゲソが」

はて、どこの世界の小学校の話だろうか？

「たけるがドラゴンの表紙の分厚い本を教科書に使ってるでゲソ。確か、ドラゴンなんとか[*1]とか…」

[*1] 「コンパイラ―原理・技法・ツール」通称 “ドラゴンブック”

どういう小学校[*2]なんだ。それはともかく、コンビネータっていうのは自由変数を含まないラムダ式のことだ。狭義には関数を引数に取って、それを組み合わせて新しい関数を返すような高階関数を指すことが多いように思う。パーザコンビネータ[*3]などは典型的な例で、組み合わせ可能な小さなパーザから目的の関数を組み立てるというものだ。

「それで、Composability というわけでゲソ」

Composability が大事だという事は分かった。だけど、それがプログラムの代数的構造を考えることにどう関係があるのだろうか。それに、Composability という言葉を気軽に使ってはいるけど、どういう意味かあまりちゃんとは考えてはいなかった。

「じゃあ例として、パーザコンビネータを考えてみようじゃなイカ」

イカ娘はいつの日かそうしたように、頭から長く伸びた青い触手で鮮やかに僕の手からシャープペンを奪い取り、軽やかに紙の上に滑らせ始めた。

```
newtype Parser a
  = Parser { unParser :: String -> [(a, String)] }

empty :: Parser ()
empty = Parser $ \xs -> [((), xs)]

char :: Char -> Parser Char
char c = Parser f where
  f (x:xs) | c == x = [(x, xs)]
  f _ = []
```

「ここでは簡単のためにパーザの型を `String -> [(a, String)]` にしたでゲソ。まあここではどうでもいいことでゲソね。`empty` は空文字を受理するパーザでゲソが、これの詳しい話は後でするでゲソ。`char` が特定の文字を受理するパーザでゲソね。これも特にひねりのない実装でゲソが、実装の話はどっちにしてもここでは気にしない方がいいでゲソ」

ここまではいい。見慣れたパーザコンビネータの定義だ。

「じゃあ、文字列 “ab” を受理するパーザを書いてくれなイカ？」

ふむふむ。簡単な問題だ。`char 'a'` が “a” を受理するパーザで、`char 'b'` が “b” を受理するパーザだから、これらをシーケンシャルにつないだものになる。`char 'a' >> char 'b'` とでも書けるだろうか。

「その >> はどこから来たんでゲソ？」

>> を使うためにはパーザがモナドのインスタンスになっている必要がある。だけど、ここでの Parser 型はモナドのインスタンスにはされていない。

「Parser がモナドのインスタンスになっている必要はないんじゃなイカ？ やりたいことは、Parser をモナドのインスタンスにすることではなくて、2 つの Parser を直列につなぐことでゲソ。例えばーー」

[*2] Functional Ikamusume Advent Calendar 2011 一日目「たけるの通う小学校に Dragon Book や SystemC 等の専門書籍がてんこもりでヤバイでゲソ!」`http://d.hatena.ne.jp/xiaohuli3/20111201#1322758513` 参照

[*3] Haskell では *Parsec* `http://hackage.haskell.org/package/parsec`、atto-parsec* `http://hackage.haskell.org/package/attoparsec` などが有名

そう言って、再度流れるような動きでペンを走らせる。緩やかにカーブした青い触手は目の前でくねりくねりとうごめき、その怪しい動きは僕を惑わす。

「ちゃんと聞いてるでゲソか？」

ノートには式が完成していた。

```
seq :: Parser a -> Parser b -> Parser (a, b)
seq p q = Parser $ \s -> do
  (a, t) <- unParser p s
  (b, u) <- unParser q t
  return ((a, b), u)
```

「これは 2 つの Parser をつなぐコンビネータでゲソね。実装にはリストモナドが使われているけど、それもやっぱりどうでもいいでゲソ。世の中どうでもいいことが多すぎじゃなイカ？」

それこそどうでもいいんじゃないか、と思いつつ、これを使って文字列 “ab” を受理するパーザを僕は次のように書きなおした。

```
char 'a' `seq` char 'b'
```

「上出来じゃなイカ！」

イカ娘は満足気にフンフンと鼻を鳴らした。あまり大したことをしたつもりはないのだけども。

「じゃあ、今度は文字列 “xyz” を受理するパーザを書いてみてくれなイカ？」

あれ、それはどこが難しいのだろう。2 つのものが 3 つになるだけじゃないのか？

「2 と 3 の間には超えられない壁があるのでゲソ。例えば、この現実の世界と、アニメの世界のように」

二次元の世界に行きたくてもいけないのと、この例題は関係ないのではないか？ 少なくともこのケースは、2 個を 3 個に変えるだけだ。

```
char 'x' `seq` char 'y' `seq` char 'z'
```

「いや、その理屈はおかしいでゲソ」

すっとぼけた顔でイカ娘は言った。

「この式は曖昧でゲソ。関数 seq を自然に演算子として使っているけど、結合性についてはどこでも触れてないじゃなイカ。その式は二通りの解釈があるでゲソ」

そう言って次の 2 つの式を書いた。

```
(char 'x' `seq` char 'y') `seq` char 'z'
char 'x' `seq` (char 'y' `seq` char 'z')
```

はて、この 2 つに違いはあるのだろうか。返り値の方はとりあえず置いておいて、その他に気にする必要があるものだろうか。

「当たり前じゃなイカ。この 2 つは全然違う式でゲソ。次の 2 つは全く違うものじゃなイカ！」

```
(1 - 2) - 3
1 - (2 - 3)
```

それはそうだ、整数の引き算が左結合と右結合で異なる結果になるのは誰でも知っている。だけど、今回の場合は関数だ。あるいは一般化してプログラムだとするべきか。いずれにせよ seq 関数は、プログラムに対する演算子に他ならない。

「seq 関数がプログラムに対する演算子なら、その結合性を気にするのは当然でゲソ」

そりゃそうだ。だけど、今の今まで僕はそのことを気にせず Haskell のプログラムを書いてきた。もちろんモナドを使わずに書いてきたわけではない。パーザコンビネータだってよく使う。でも結合性で問題を起こしたことはない。不可解な動作を起こしたことはない。なぜだろうか。

例えば 3 文字以上のケース、特定の文字列 s を受理するパーザを考えてみる。

```
string :: String -> Parser ()
string s = foldl seq empty $ map char s
```

あるいは次のように書けるだろうか。

```
string :: String -> Parser ()
string s = foldr seq empty $ map char s
```

右結合と左結合が異なるのであれば、これらも違うということになってしまう。どちらを使うべきなのか、fold を書く度に気にするのはめんどうだし、状況に応じて適切なものを使い分けるということがそもそも不可能になってしまう。

「少し意地悪な問題だったでゲソね。本当は、さっきの 2 つの式に違いはないでゲソ。時間がかかるからなんでそうなっているかは示さないでゲソ。気になるんだったら帰ってから式を展開してみるといいでゲソ」

そうなのだ。この 2 つが異なる意味を持ってもらっては困る。そんなことがあっては、おちおちコードを書いていられないじゃないか。

「そうでゲソ。我々は加算が結合律を満たすということを知っているし、減算がそれを満たさないということを知っているんでゲソ。じゃあ他の演算子についてはどうでゲソか？」

数に対する四則演算は息をするようにその性質を意識してコードを書くことができる。それは小さい頃、右も左もわからないまま算数教育としてそのあたりの法則を刷り込まれているからに過ぎない。我々が日々用いる演算は、プログラミング時のそれと比較するとほんの一部分にすぎないのだ。

「そこで、圏論の登場というわけでゲソ」

深く首を立てに振りながら、イカ娘は勿体ぶってそう言った。

「プログラミングにおいて、特に関数プログラミングにおいてでゲソね。プログラムに対する演算というものを考え始めると、その代数的構造のもつ性質を考えることはとっても重要になるんでゲソ。例えば、さっきのようなーー」

ここまで言われたら、さすがの僕でも察しがつく。seq 関数とパーザコンビネータが構成する代数的構造がどのような性質を持っているのか。今回の場合だと seq 関数が結合律を満たすかどうかだ。

「私が書いた seq の定義は結合律を満たしているでゲソ。だけど、それを示すのは大変だし、大変じゃなかったとしても、演算子を見るたびにそんなことを確認するのは面倒じゃなイカ！」

そりゃあそうだ。でも、それをしなければ安心してコンビネータを使うことはできない。

「それで、そのへんにまつわる問題を一気に解決するためのツールとして、プログラミングに圏論、それから群論などの代数学を持ち込んだのでゲソ！」

そう言って、イカ娘は「ドヤ？」とでも言いたそうな顔でこちらを見つめてきた。

「例えば、モノイドというものがあるでゲソ。モノイドは単位元を持つ半群のことでゲソ。半群っていうのはーー」

半群は僕も知っている。次の 2 つの性質を満たす集合 S と二項演算 “・” の対のことだ。

- a, b ∈ S ⇒ a・b ∈ S (演算が閉じている)
- (a・b)・c = a・(b・c) (結合律)

「そのとおりでゲソ！ じゃあ、パーザとそれを組み合わせる二項演算が例えばモノイドになっていたとすると、どうなるでゲソ？」

あっ、モノイドだったら、何も考えずに結合律を満たすと見なせるではないか！

「モナドはモノイドに似たもので、クライスリ圏の射に相当するものでゲソが、これも結合律を満たすものでゲソ。これのお陰で細かいことは気にせずに自由に安心してモナドのプログラムを組み立てることができるんでゲソ！」

僕がモナドをあまり深く考えずに簡単に操ることができたのはそのおかげだったのか。しかし僕はまだピンと来ない。なぜ代数学なのか。

「それは、数学の世界の成果をまるごと取り込むことができるからじゃなイカ？」

イカ娘は腕を組み、目を閉じしばし考えに耽っている。頭から伸びた触手が中空を彷徨う。

「我々が対象としたいプログラムと演算子は、既に歴史ある群論や圏論で研究されている代数的構造である可能性が高いでゲソ。とすれば、それに関する数学的な考察がそっくりそのままプログラミングに適用できるでゲソ。それにーー」

なるほど。確かに敢えて独自の構造を再定義する必要はあまりないだろうし、既存のものに合わせたほうが知識も共有できて利便性も高い。

「モナド、モノイド、その他もろもろの借用物を演算子定義の際のガイドラインにする、つまりモナドはモナド則、モノイドはモノイド則を満たすように、プログラムやライブラリを書くみんなが気をつけることによって、私たちはモナドという型クラスの制約を自分のコードに課すだけで、何も心配せずに、安心して自由にコードを抽象化することができるようになるのでゲソ。これは計り知れないメリットでゲソ！」

ここまで聞いてようやく僕も納得がいった。自分の型を、既によく知られたプロパティを持つクラスのインスタンスにすることによって、細かいことを考えずとも自分の型が優れた代数的性質を持つようにすることができるのだ。その最たるものは、プログラムの Composability だったのだ。

「プログラミングで一番大事なことは Composability でゲソ。小さいプログラムから大きなプログラムが矛盾せずに作れるということは、プログラムの抽象化を自由に行えるということでゲソ。逆にそうじゃなければ、どこかに分割できないブロックが発生するということでゲソ。Composability さえしっかりしていれば、プログラムの妙な挙動で驚くことはなくなるでゲソ。これがホントの驚き最小の原則じゃなイカ！ アッ、と驚くタメゴロー」

フンフンフン、と鼻を鳴らしながら満足気にイカの少女は去っていった。一人残された僕は夕暮れの海を眺め、しばし呆然としていた。

帰りの電車で、僕はノート PC を開きながら、パーザについてもう少し考えていた。パーザコンビネータで本質的なものはもうひとつある。それは選択だ。Parsec の新しいバージョン、Parsec3 を

見てみると、選択の演算子は Alternative クラス[*4]を実装した<|> になっていた。

Parsec3 には他にも、`many` や、`some` 、`manyTill` などのコンビネータが定義されているが、これらのコンビネータは実は<|> だけを用いて定義できることがわかった。これはパーザ固有のものではなかったのだ。Alternative のインターフェースに合わせれば、ただひとつの演算子を定義するだけで、様々な便利な関数がノーコストで使用可能になる。

そこで僕は気がついた。プログラムの抽象化、関数、型、オブジェクト指向におけるカプセル化、モジュール化、云々。これらはすべて Composability を高めるために行われることだったのだ。カプセル化は、モジュール間の依存関係を少なくすることによって、確率的に複数のモジュールが干渉する可能性を下げようとしていることに他ならない。圏論によるアプローチは、数学のフレームワークを利用することによって、依存関係を減らさずに Composability を確保するというある意味スマートな、でもある意味では強引なアプローチだったのだ。

「次は、＊＊＊、＊＊＊」

電車が駅に到着した。考え事をしていると時間が立つのが速いものだ。

「ダァシエリイェス！！」

どこか気怠い車掌のアナウンスを背に、僕は電車を降りた。今まで書いてきたプログラムを、書き直したくて手が疼く。あれもこれも、きれいに書き直せる。そんな気がして、それを早く確かめたかった。そんな想いを胸に、僕は足早に家路についた。

[*4] Control.Applicative に含まれる Functor クラスのサブクラス。標準ライブラリに含まれる。

第6章

映画けいさん！

— @tanakh

この作品は某映画を見たいけどなかなか見に行けない私の妄想の産物、つまりフィクションであり、実在する人物、地名、団体とは一切関係ありません。

「唯先輩、起きてください」
「むにゃむにゃ、あと 1 時間だけ……」
「ダメです。そんな時間ありませんよ」
「むにゃむにゃ、ハスにゃん。もう食べられないよ」
「何寝ぼけてるんですか。着きましたよ、ロンドン」
「……あ、おはよう、ハスにゃん」

私たちは今、ラムダ高計算機プログラミング競技研究部、通称「計算部」の課外活動としてイギリスに来ています。

飛行機の中でずっと私にもたれかかって寝ていたのが、平無駄唯先輩。いつもはとぼけているように見えて、今私たちがここに来れているのも唯先輩のおかげだったりします。

「ロンドンだー、ロンドンだー」
「ヒースローだー！」

あそこで唯先輩と一緒にはしゃいでいるのが、排中律先輩。2 年前に潰れかけだった計算部を復活させた人で、計算部の部長です。普段はいい加減な人ですが、いざというときは…やっぱり頼りないかも。

「おい、澪。何してんだ。早く荷物取ってこいよ」
「うう…飛行機怖い……」
「飛行機はもう降りただろ。おい、しっかりしろー」

律先輩の隣で青い顔をしているのが、入文字山澪先輩。計算部で唯一の常識人で、実力もトップクラス。私のあこがれの先輩です。怖がりなのが玉に瑕かな。

「あらあら、ハスさちゃんは常識人じゃないのかしら？」
「あっ、モナ先輩。なんで私の考えてることを！？」

そしてこれが、型吹モナド先輩。おっとりしていて、すごいお嬢様なんだけど、ちょっと何を考えているのかわからない所があったりします。

「はあ、こんなので大丈夫なのかなあ」

海外に来ても、計算部は相変わらずです。

「ハスにゃん、すごいよ！ 白人がいっぱいいるよ！」

「もう、唯先輩。ここイギリスなんですから、当たり前じゃないですか。って、どこ行ってるんですか！？」
「えっ、こっちじゃないの？」
「乗り継ぎですよ！　ヒースローで降りちゃダメです」
はあ、こんなので大丈夫なのかなあ。

「うわあ、ここがグラスゴーかあ」
「すごい、すごいよハスにゃん！　ヨーロッパだよ！」
「イギリスは毎年来てますけど、今年が一番ワクワクします」
「ちくしょう、お嬢様は違うなあ」
グラスゴーはスコットランド最大の都市で、イギリスで 4 番目の人口を誇ります。イギリス国内では、ロンドンとエディンバラに次いで 3 番目に観客が多いそうです。ハリーポッターが書かれた街として有名なエディンバラからは列車で 1 時間程度。中世の時代から多数の文化が生み出され、産業革命以降は工業都市として栄えたそうですが、今では古風な街並みを残した落ち着いた街です。
私たち計算部は “International Computer Programming Contest”、通称 ICPC の世界大会に出場するためにグラスゴーに来ています。ICPC は毎年いろいろな国で開催されていますが、今年はイギリスのグラスゴーにあるグラスゴー大学が主催なのです。
グラスゴー大学は 15 世紀に設立された名門大学で、英語圏では最古の、とても歴史ある大学だそうです。折しもグラスゴー大学は敬愛するサイモン・ペイトン・ジョーンズ先生が教授として在籍していた所で、私は楽しみのあまり、数日前から満足に寝付けなくてーー。
「おーい、ハスさ、おいてくぞー」
「ハスにゃん、よだれ垂れてる。かわいい」
「むにゃむにゃ、あっ、先輩。待ってくださいよー」
でも、そんなことは先輩方にはお構いなしなようです。そんなこんなで、空港からのタクシーがホテルに到着です。いよいよ私たちの戦いが始まります。

ICPC は世界中から数万人が参加する、数あるプログラミングのコンテストの中でも最も歴史の長く、規模の大きい大会です。制限時間内に正しく動作するプログラムを、なるべくたくさん書いたら勝ちという、単純明快なルールです。国ごとの予選を勝ち抜き、さらに世界 5 つのエリアで行われる予選で上位に入った、わずか 100 チーム弱の強豪たちがここ世界大会の舞台に集うのです。ちなみに 1 チームは 3 人なので、じゃんけんに負けた律先輩とモナ先輩は補欠になりました。
「おお、すごい」
「あっ、看板があるよ。写真撮ろう！　写真撮ろう！」
”International Computer Programming Contest World Finals 20xx” と書かれた看板がホテルから入ってすぐのところにおいてあります。会場はとても大きなホテルで、不覚にもそわそわしてしまいそうです。顎足つきじゃなければ、とてもこんなところに泊まれないだろうなあって。
「ハスさちゃん、真ん中寄って」
「あ、お姉さん写真お願いします」
「おい唯、ここは日本じゃないぞ、ちゃんと英語で頼まないとだなあ。あ、えー、ぷりーず、えーと、ふぉと、ぷりーず」
「写真？　いいわよ。はい、みんなもっと真ん中寄って」
えっ、日本語？

「ありゃ、日本人の方ですか？ ああっ、よく見たらさわちゃん先生じゃないかー！」
「本当だ、さわちゃんだ。変な格好してるから気付かなかったよ」
さわ子先生、部室の押し入れの写真みたいな、本格的に変な格好してる……。ちょっと近寄りがたいような。でも、昔はさわ子先生もプログラミングに青春賭けてたって本当だったんだ。
「もう、失礼ね。本場なんだからそれなりの格好して行くわよ。それより排中さん。ちゃんと英語勉強しておくようにって言わなかったかしら？」
「あははは。それよりさわちゃんなんでここに？ さては私たちに便乗してイギリス旅行とか？」
「何言ってるのよ、私、あなた達のコーチじゃない」
「あっ、そういえばそうだった」
「もう、あなたたちそんなので大丈夫なの？」
先生。私も同感です。

「うわあ、枕が。ハスにゃん、枕がいっぱいあるよ！」
唯先輩たっての希望で、私は唯先輩と同じ部屋になりました。部屋の鍵を開けた途端、唯先輩はベッドに向かってダイブしていきました。もう、子供だなあ。でも、すごい部屋です。広い部屋にキングサイズのベッドが 2 つ。そして、枕がひい、ふう、みい、よお…
「先輩、首痛くないんですか？」
「ふご、ふごご！ は、ハスにゃん助けて！」
先輩は枕を全部重ねてそこに寝ようとしたみたいです。首が変な方向に曲がっているような…。
「ふごごご！ ハスにゃん、はやく…」
「もう、なにしてるんですか」
唯先輩の首に手をかけて、起こしてあげます。あ、唯先輩、良い匂いがする…。
「ありがとうハスにゃん。死ぬかと思ったよお」
「こんなところで死なないでください。唯先輩、うちの頼みの綱なんですから」
唯先輩は、普段はこんな感じなのに、一度プログラミングを始めると別人になるんです。文字通り別人になるわけではなくて、見た目も、口調も、雰囲気もそのままなのに、淡々ととんでもないことをやり遂げてしまうんです。

去年の夏の合宿の時でした。
「gdb?」
「デバッガですよ。唯先輩、使ったことないんですか？」
「デバッガ？ デバッガって何？」
「デバッガっていうのは、プログラムのバグを取るツールですよ。唯先輩、printf デバッグしかしたことないんですか？」
「printf デバッグ？ それより、バグって何？」
「えっ、唯先輩。プログラムが思い通りに動かないときどうしてたんですか？」
「あれえ、私プログラム書いてて思ったとおりに動かなかったことなんてないよ？」
「そんなことあるわけ…。ちょっとソース書いてみてください！」
「じゃあちょっと書いてみるね。ほい」
カチャカチャカチャ…規則正しくタイピング音が響いて、淀みなくコードが出来上がっていきます。私はペアプログラミングするつもりで後ろから先輩がコーディングする様子を見ていました。でも、疑わしいコードを見つけることもなくて。

「できたよハスにゃん」
「え、もうですか？ じゃあサンプル通してみましょうか」
「ええ、めんどくさいよお。サブミットしちゃおうよ」
「そんな、コンパイルも通さずにサブミットだなんて無茶な...って、ええっ？」
躊躇うこともなく唯先輩はオンラインジャッジサイトに今書いたコードを投稿してしまいました。そして、それがそのままアクセプトされ...。

神様は不公平です。いくら練習しても私は唯先輩のようにコードを書くことはできません。そのことを考えて、私は何度計算部をやめようかと思ったかわかりません。でも、結局辞めることができなくて、それでここにいるわけでーー。
「よしっ、と。そろそろ出発するよ、ハスにゃんも早く」
「ああっ、待ってくださいよお。今準備します」
ちょっと考えこんでしまっていたようです。でもそんなことを考えるのは、今は必要ないよね。

「遅いぞ、唯」
「ごめん、澪ちゃん。みんな早いね」
「お前が遅いんだよ」
「律っちゃん。女の子には色いろあるのだよ。」
「なるほど、そうかー。って、私も女の子じゃい！」
先輩たち仲いいなあ。でも、来年にはみんな別々の大学に...。いつも通りにバカをやって、バカなことを言ってる先輩達を見ていると、これが一緒のチームとして参加する最後の大会なんだってことを忘れてしまいそうで。
「みなさーん。そろそろチーム登録行きませんか？」
「お、そうだな。ちょうど列も空いてきたみたいだし」

ICPC 最初のイベントが参加登録です。あ、参加登録ってバカにしちゃいけないのです。参加登録を正しく行わないと、出場権を失ってしまいますし、それに参加者が一堂に会する参加登録は、他の参加者との最初の顔合わせの場でもあるんです。
「うわあ、人がいっぱいいるよ。これみんなプログラマなんだね」
「しかも地区予選を突破してきた強者揃いだぞ」
「わ、私怖くなってきた.....」
「お、おい澪、しっかりしろ―」
「そうだよ澪ちゃん、こういう時は堂々としてないと。ハロー、ウィーアー、ホカゴ tee タイム、フロムジャパン。イエァ、イエァ...」
唯先輩、登録始めちゃったけど、大丈夫かなあ。なんか喋ってるけど、あんまりよく聞き取れないや。私も英語苦手なんだった。
「みんなー。登録できたよ。はい、これ」
唯先輩の手には手提げ袋と、変な怪獣みたいなぬいぐるみ。これは参加賞みたいなものなのかな。それからもう一つ、チームの T シャツ。
「うおお、テンション上がってきた！」
「律っちゃんずるい、私も早く着たいのに」

T シャツは ICPC でのユニフォームのような存在です。私たちのチームの T シャツには、表面に”HTT” というチームのロゴが入っています。コーチのさわ子先生がデザインしたものです。

「無事登録は済んだみたいね。みんなお疲れ様。T シャツよく似合ってるわよ。さすが私の見立てね」

ちょうどさわ子先生がやって来ました。さわ子先生は T シャツを見ると満足気に帰っていきました。

「さわ子先生、何しに来たんだろう...」

「さあ...」

「先生もいそがしいんじゃないか？ 男探しとかで」

ひどい言われようです。

「.....唯先輩、起きてください」

「あと 1 時間だけ...むにゃむにゃ」

「そんな時間ありませんよ。遅れちゃいます」

「むにゃむにゃ、もう食べられないよ」

「唯先輩、また寝ぼけてるんですか？」

「.....あ、おはよう、ハスにゃん」

グラスゴー 2 日目です。時差ボケと緊張と興奮で、私はほとんど眠れませんでした。でも、そういう苦労も唯先輩には無縁のようです。

「はー、よく寝た。ハスにゃん、今日もがんばろうね！」

「本番は明後日ですよ、先輩。それまで体力残しといてくださいよ」

「えー、観光のほうが楽しみだよう。初めての海外なんだよ？」

「なんというか、とても先輩らしいです」

そうだ。この人はこういう人だ。むしろこうでなければいけないんだ。こういうある意味で異常な空間で、いつも通りの自分を保てるというのはとてもすごいことでもあるんです。そういうところも羨ましくて、すぐに舞い上がってしまう自分が何となく情けない感じもして。

「さ、ハスにゃん。そろそろ出発するよ」

「あ、待ってくださいよお」

唯先輩はもう準備を終えていたようです。私はカバンに水と名札を突っ込んで、急いで部屋の入口に行き、

「やってやるです！！」

気合を入れ直しました。私も負けていられません。

ICPC は競技自体は 5 時間程しかありませんが、全体で 5 日ほどの日程が組まれています。全世界から参加者が集うので、時差ボケの緩和の意味があるんだと思います。現に私は睡眠時間が不規則でふらふらです。その間にエクスカーションとして Tech Talk や観光が組み込まれています。この色々なイベントも ICPC の魅力だったりします。

Tech Talk は、主催の IBN による最新の技術紹介など、私たちプログラマにとってはとても刺激的な内容なのですが、当たり前だけど、全部英語で行われます。朦朧とした頭には、この淡々と流れる英語がとてもいい子守唄になるわけでーー。

「ハスにゃん、ハスにゃん。起きて」
「……うーん、ハッ！ ここはどこ！？」
気づくと、周りには誰もいなくなっていました。
「ハスにゃん気持ちよさそうに眠ってたから……。もう Tech Talk 終わっちゃったよ」
「そんなあ…。はあ……」
「あとで内容教えてあげるよ」
「め、面目ないです」
こういう時、唯先輩がいてくれてよかったなあと思います。

「ぷはー、食った食った」
「律、はしたないぞ」
「いやー、失敬失敬」
午後は、グラスゴー市内の観光がありました。それから、ウェルカムレセプションです。
唯先輩は終始はしゃいでいました。澪先輩は英語で話しかけられる度に青ざめていました。律先輩はひたすら食べていました。モナ先輩は完全に周りに溶け込んでいました。私はーー。
「ハスにゃん、こっちこっち」
「えっ、唯先輩」
唯先輩、外国のチームの人達と話してたみたいです。私を強引にそっちに連れていこうとしています。
「これがさっき話してたハスにゃん」
「えっ、えっ」
「とっても可愛いでしょ。私ハスにゃん大好きなんだ」
「Oh… you're so cute!」
わわっ、いきなり大変なところに連れてこられちゃったみたいです。ええい、なんとでもなれです！
「さ、サンキュー」
「What's your favorite programming language? … 」
「あっ、えーと…」

「ふう、疲れたです」
あれからしばらく唯先輩に連れられて、外国チームの人達と話していました。
「ハスにゃん意外と英語喋れたんだね。私びっくりしちゃった」
「もう、そう思うんだったら、無茶させないで下さいよ」
「だってハスにゃん、あんまり楽しそうじゃないみたいだったから。何事も経験だよ、ハスにゃん」
ほんと破天荒な人です。巻き込まれる身にもなってほしいです。でも、ちょっと楽しかったな……。

「唯先輩、起きて下さい」
「むにゃむにゃ、アイキャノットイートエニモワー」
「唯先輩、英語で寝ボケてないで、早く起きて下さい」
「……はぅっ、あ、ハスにゃん。おはよう」

唯先輩は通常運転です。私もホテルの枕にも少しずつ慣れてきました。

今日は、本番前日です。つまり、明日は本番なのです。と言うことは今日は本番前日で、明日の夜にはもう結果が出ているわけで、ここにこうして集まって和やかに談笑している人たちも明日の夜には明暗分かれて悲喜こもごもで、つまり今日は本番前日で……。

「わ、ハスにゃん、どうしたの？」

「あ、先輩。すみません」

緊張して頭の中がぐるぐるです。目眩で足元がぐらつきました。

「おい澪ー、しっかりしろー」

「あ、明日…もう明日……」

「おいー、帰って来いー」

澪先輩も通常運転のようです。

今日は予行練習があります。明日実際に用いる機材のチェック、システムが正しく動作するか、ソフトウェア環境の確認。当日焦らないために、今日しっかりチェックしないといけません。

「あれえ、ログイン出来ないよ」

「唯、ちゃんとパスワード入れたのか？」

「ちゃんと入れたよお。何度も確認したし」

「ちょっとキーボード貸してみろよ。ええと、ユーザ名 team86、パスワード xxxxxxxx と。ありゃ？」

「ほら、入れないでしょ？ 澪ちゃん疑り深い」

「うーむ、なぜ入れない」

いきなりトラブルみたいです。前日でよかったです。

「係の人に聞いてみました」

「おお、さすがモナ！」

「パスワード、ダブルクオート込で入力しないといけないみたいです」

「ええっ、って、なんでやねん！」

たしかに、パスワードが印刷された紙を見てみると、ユーザ名はダブルクオートで囲われていないのにパスワードは囲われています。

「これはひどい」

「いやあ、こういうの有りがちみたいだぞ？ 実際に何年か前の某大会でも同じようなことがあったってどっかのブログの参加記で読んだし」

「へえ、そうなんだあ。こういう大会でも案外ミスがあるもんなんだね。あっ、ログインできたよぉ」

「よし。じゃあ適当に練習問題サブミットするか」

「アイアイサー！」

唯先輩はいつもどおり、流れるような手つきでかたかたコードを入力していきます。なめらかに動く指の動きに私は魅入られていました。規則的で不規則な怪しい動きは、休むことなくコードを生成し続け、私はそれを眺めるうちに、いつしか眠りの淵にーー

「おっしゃー、トップとったどー」

「とったどー」

「おい、リハーサルでトップとっても意味ないだろう」

いつのまにやら練習問題を解き終わっていたようです。

「写真とっとこー！」

「ハスにゃんもおいでよ」
唯先輩と律先輩がはしゃぎながらスコアボードの写真をとっています。はあ、まったく子供みたいなんだから。でも、せっかくだから、私もちょっとだけ。
「待ってくださいよお、先輩！」

夜、ホテルのボールルームで親睦を深めるためのイベントがありました。巨大なチェスや、様々なボードゲーム。技術ネタを話すスペース、それと PC。
「チェックメイト！！」
あ、唯先輩がチェスをやっています。しかも勝ってる。
「あ、ハスにゃん。来てたんだ」
「唯先輩、チェスもできたんですね」
「え、やったことなかったよ。ルールぐらいは知ってたけどね」
「えっ…」
「私一度やってみたかったんだ。あの大きなチェスボードで」
「そう…ですか……」
唯先輩は以前部で将棋のルーチン書いてたから、チェスができても不思議は…って、やっぱり納得できない！
「ん、どうしたのハスにゃん。眠いなら寝てきていいよ」
「はい、そうします。明日も早いですし。先輩も早く寝てくださいね」
「ハスにゃん、おやすみー」
明日は本番です。万全の調子で望まなければ。

「起きて、時間だよ。遅れるよ」
「うーん、むにゃむにゃ、あと 5 分だけ……」
「ダメだよ、もう時間ないよ。起きてハスにゃん」
「むにゃむにゃ、唯先輩、んっ、そこは…」
「寝ぼけてないで起きて、ハスにゃん！」
「……あっ、おはようございます唯先輩」
しまった、この日に限って寝坊しちゃった！

結局緊張して昨日はほとんど眠れませんでした。朝ごはんを食べる時間ももうなさそうです。
「うう、すいません、先輩」
「仕方ないよ、ハスにゃん。それより急いで」
急いで名札と T シャツを身につけて、会場へと向かいました。
「遅い！ 何やってたんだ」
「ごめん、律っちゃん」
「すみません、私が寝坊しちゃって……」
「いいから、入場ゲートに向かって！ そっちよ。あと、電子機器は持ち込めないから、持ってたらここに入れて」
「おい、澪もしっかりしろ」
「あ……ここ、は……あ…こわい……コンパイラこわい…」

「おーい、帰ってこーい」
「しっかりして下さい、澪先輩」
澪先輩に肩を貸して何とか会場入りします。
「頑張ってこい！ 綺麗な色のメダルを期待してるぜ」
「がんばってね、私たちの分も」
そうだ、これは私だけの戦いじゃない。参加できなかった、律先輩とモナ先輩の分も、私達ががんばらなきゃ！

ふう、はあ、ふう、はあ。
「何やってるの、ハスにゃん」
「腹式呼吸です。集中力を高めているんです」
「ひ、ひ、ふうー。ひ、ひ、ふうー」
澪先輩、それ違います。
「ばりばり」
「うわあ、何やってるんですか唯先輩。やめてください」
「何って、問題の封筒を」
「封筒の面を見てください！」
「“Do Not Open Until Told To Do So.” . . . ？ あっ、しまったー」
「とっ、とりあえずしまって下さい！」
唯先輩、やっぱり通常運転です。
「あぁ...あと 5 分...あと 5 分.....」
澪先輩も...。本番開始まで、あと 5 分...。

「10, 9, 8, 7, 6, 5, 4, 3, 2, 1, . . . スタート！」
いよいよ戦いの火蓋が切って落とされました。5 時間の長くて短い戦いが始まったのです。
「唯先輩！ 封筒、開けて下さい！」
「むにゃむにゃ...、はっ。あっ、開けるよっ！！」
封筒の中には綴じられた問題の紙が 3 部。1 人に 1 部というわけです。
「澪ちゃんとハスにゃん、問題読んで！ 一番簡単そうな問題を見つけといて！」
「わかった！」「わかりました！」
うちのチームの戦略はシンプルです。私と澪先輩が問題を読んで、簡単そうな問題から唯先輩に渡していって、唯先輩はひたすらコードを書いてサブミットしていきます。唯先輩は最初の内に、共有コードを打ち込んでおきます。ICPC ではコードの持ち込みが禁止されているからです。
「これ！ 素数を列挙すれば簡単に解ける問題だぞ」
「サンキュー、澪ちゃん」
グッジョブです、澪先輩！

「あれ、通らないよお。たすけてハスにゃん」
どうしたんだろう、唯先輩。今までこんなことはなかったのに。
「どういうことですか、唯先輩。って、TLE ですか？」

唯先輩はなぜだか Haskell を好んで使っています。コードをほとんど間違えないというのも、デバッガを使わないというのも、その恩恵によるところが大きいようです。初めてその様子を見たときはとても驚いたものですが、でも今回はそれが仇になってしまいました。

「ちょっとコード見せて下さい」

私も Haskell はちょっとは読めます。なんとかしないと。最終的には C++ に書き換えることも辞さない構えで。

「あ、これ、ブログで見た問題だ...」

唯先輩の書いた素数列挙のコードは次のようになっていました。

```
primes = 2 : sieve [3, 5 ..] where
  sieve (p:xs) = p : sieve [x | x <- xs, x `rem` p /= 0]
```

よく Haskell できれいに書ける例として登場するコードですが、これはダメです。エラトステネスの篩になってないんです[*1]。ターナーの篩というやつなんです[*2]。

```
primesTo m = 2 : eratos [3, 5 .. m] where
  eratos []     = []
  eratos (p:xs) = p : eratos (xs `minus` [p, p+2*p..m])
```

コードを少し変えて、こうするだけで大分速くなります。

「唯先輩！ これを！」

「あ、ありがとうハスにゃん！」

このコードは速くなるけど、C 等による実装に比べるとまだはるかに遅いです。これがダメだったら、ST モナド[*3]で普通に篩を実装するしかない...。

「通った！」「やりましたね、先輩！」

よかった。杞憂に終わった。私は問題読みに戻ります。

「先輩、これ簡単な DP です」

「ありがと、ハスにゃん！」

先輩に問題と解き方を伝え、託します。しかし、唯先輩の手が動きません。

「唯先輩、どうしたんですか？」

「ハスにゃん.....私 DP 書いたことない」

「えっ...」

練習しといてって言ったのに.....。

「Haskell で DP するには、メモ化ですよ、メモ化」

「ああっ、どうやってメモ化すればいいか、すぐに思いつかないよお」

「メモ化を Haskell でやるには幾つか方法があります。Purely なテーブルを持ちまわる方法、遅延無限テーブルを使う方法、そして、モナドを使う方法です。一番手軽で簡単で高速なのは、モナドを使う方法だと思います！」

「分かった！ ハスにゃん、ペアプロしてくれる？」

「任せて下さい。やってやるです！」

[*1] http://www.haskell.org/haskellwiki/Prime_numbers に詳しい

[*2] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes を参照

[*3] メモリを読み書きするためのモナド

メモ化のモナドとしてはMonadMemo[*4]を用いるのが楽です。

```
class Monad m => MonadMemo k v m | m -> k, m -> v where
  memo :: (k -> m v) -> k -> m v
```

MonadMemoの型パラメータは3つです。キーの型k、値の型v、そして、メモモナド型m。具体的な型としていろいろなコンテナとモナドをバックエンドに使えるのです。

例えば、フィボナッチのメモ化ならこんな感じになります。

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 1)
```

これが元のコードで、

```
fib :: Integer -> Memo Integer Integer Integer
fib 0 = return 0
fib 1 = return 1
fib n = do
  a <- memo fib (n - 1)
  b <- memo fib (n - 2)
  return (a + b)
```

こうなります。とてもシンプルです。Applicativeを使えば、もっとすっきりとした記述になります。

```
fib :: Integer -> Memo Integer Integer Integer
fib 0 = return 0
fib 1 = return 1
fib n = (+) <$> memo fib (n - 1) <*> memo fib (n - 2)
```

MonadMemoの使い方を教えると、唯先輩の指は再び滑らかに動き始めました。

「できた、できたよハスにゃん！」

「やりましたね、先輩！」

「澪ちゃん、ハスにゃん。次の問題は？」

「うーん、ちょっと解き方がわかるのが無いな」

「私もです……」

膠着状態に陥ってしまいました。でも、澪先輩があんなに考えても分からない問題がそんなにたくさんあるものかなあ。

「あっ、この問題、探索でいいんじゃないんでしょうか？」

「うーん、でもちょっと入力値が大きくて自信が持てない」

「澪先輩が分からないということが何よりの状況証拠ですよ！」

「あ、ハスにゃん頭いい」

[*4] http://hackage.haskell.org/package/monad-memo

「おいおい、どうなっても知らないぞ」
「澪ちゃんは他の問題考えといて！ ハスにゃんは私とペアプロ！」
「了解です！」
探索と決まればあとは速いです。唯先輩の淀みないコーディングは瞬く間にコードを完成させました。
「いくよ、ハスにゃん！」
「どんとこいです！！」
勢い良くテストを走らせてみるものの、結果が帰って来ません。
「やっぱりダメだったかなあ……」
「あ、答え出ましたよ。でもちょっと遅いですね」
「うーん、困ったなあ」
「そうだ、並列化ですよ！」
ジャッジマシンは、Corei7 の 4 コアマシンです。4 倍早くなれば、通る可能性は無きにしもあらずです。
「は、ハスにゃん、私並列のコード書いたこと無い……」
「さあ、早く…って、またですか！？」
この人は準備とか練習とか言う物をしないのかなあ。
「もう、仕方がないですね。こんな事もあろうかと、私が並列化ライブラリ書いてきました。使って下さい」
「あ、ありがとう。ハスにゃん！」
私は普段は Haskell を使っていませんが、Haskell で並列コードを書くのは比較的簡単です。特に ICPC では複数テストケースがあるので、ワーカスレッドを立てて、あとはタスクキューを作ればそれらの制御は簡単です。

```
main :: IO ()
main = do
  q <- newChan
  r <- newChan
  n <- readLn

  forkIO $ forever $
    forM_ [1..] $ \ix -> do
      p <- getProblem
      writeChan q (ix, p)

  forM_ [1..4] $ \i -> forkIO $ do
    (ix, p) <- readChan q
    writeChan r (ix, solve p)

  results <- getChanContents r
  forM_ [1..n] $ \ix ->
    putStrLn $ fromJust $ lookup ix results
```

これを使って、唯先輩がごりごりとコードを書きなおしていきます。
「できた！」

「サブミットしましょう！」
しばしの沈黙、そして。
「通った！」
よかった、何とかなった。

その後、残った問題は解き方も分からず、膠着したまま、時間は過ぎて...。
「9, 8, 7, 6, 5, 4, 3, 2, 1, The Contest is Over!!」
私たちは苦戦しつつも、それなりに問題を解くことが出来ました。スコアボードはラスト 1 時間でフリーズされるので、最終的な順位はまだわかりません。問題数的には、メダル圏内に手が届かないとも...。
「ふわあ、疲れた」
唯先輩は机に突っ伏しています。
「.....」
澪先輩は白目を剥いてピクリとも動きません。お疲れ様です。ゆっくり休んで下さい。
「おーい、どうだったかー」
「3 人とも、お疲れ様」
「あ、律先輩、モナ先輩。ありがとうございます」
そういう私ももう疲労がピークです。
「お部屋に帰って、休憩したほうがいいんじゃないかしら」
「モナ先輩...。それじゃあお言葉に甘えて」
ふらふらと部屋に戻り、私は泥のように眠りました。

「おはよう、ハスにゃん」
「おはようございます、唯先輩」
今日は日本に帰る日です。結局私たちはメダルを取ることはできませんでした。時間差で勝てなかったのです。私は思わず泣いてしまいました。先輩たちに有終の美を飾らせてあげられなかったことが悲しくて、悔しくて。
「大丈夫だよ、ハスにゃん」
「.....唯先輩？」
「みんなと一緒に頑張れて、私は楽しかったよ。それにーーー」
「それに？」
「計算部は、勝つことだけが目的じゃないよ。みんなで楽しくプログラミングするのが計算部なんだよ」
「そうよ、ハスさちゃん」
「私もハスさとプログラムするの、楽しかったぞ」
「私は補欠だったけど、旅行できて楽しかったかな」
「み、みなさん.....」
みんな、いい人達だなあ。この人達と同じ部で活動できて良かった。

「そうだ、ハスにゃん、これ」
そう言って、唯先輩が SD カードを差し出しました。
「なんですか？ これ」
「計算部みんなから、ハスにゃんに。ほら私達、今年で引退だから」
「あとで開けてみて。大したものじゃないのだけど」
「あ、ありがとうございます...」
私はまた泣いてしまいました。

帰りの飛行機の中。SD カードが気になって、みんなが眠っている間に、こっそりノート PC で中身を見てしまいました。
SD カードの中には、「ToHsNyan.hs」というファイルがひとつだけありました。

```
main = "keisanbu" + "hsnyan" where
  "keisanbu" + "hsnyan" = forever $ do
    putStrLn "nakama da yo!"
```

これは.....。みなさん、直接言ってくれればいいのに、恥ずかしがり屋さんなんだからーー。
でも、なんだかすごいコードだなあ。これほんとに動くのかな？

```
$ runhaskell ToHsNyan.hs
```

次の瞬間、端末が文字で埋め尽くされました。
「あっ、しまった。止まれ、止まって！」
カチャカチャカチャカチャ。私が Ctrl-C を連打する音が暗い飛行機の中を響き渡ってーー。
「.....ふわあ、ハスにゃん。おはよう」

会員名簿じゃないカ?

@xhl_kogitsune (0, 4 章)
きつねもふもふ杏子かわいい。

@dekosuke (1 章)
LL から Haskell に移住途中ですが、とてもいいですね

@master_q (2 章)
プロニート。コピペプログラマ。お仕事ください! http://www.masterq.net/

@nushio (3 章)
Haskell やって彼女ができました 2。

@yryozo (4 章)
当初予定では Scheme ネタで書くはずが... どうしてこうなった (笑)

@tanakh (5,6 章)
Haskell やって充実した生活を送っております

「簡約(かんやく)！？ λ(らむだ) カ(か) 娘(むすめ) (二期)(かっこにき)」

2011 年 12 月 31 日 コミックマーケット 81 初版発行
2012 年 1 月 6 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 , @dekosuke, @master_q,
@nushio, @yryozo, @tanakh
表紙：dif_engine@Twitter 様
挿絵：@nushio
印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/

参照透明な海を守る会