
簡約入力娘〜巻の七〜

目次

第0章

まえがき

関数型イカ娘とは!?

Q. 関数型イカ娘って何ですか?
A. いい質問ですね！
Q. 七冊目とか、そろそろあなた自身の本当の仕事と向き合えますか？
B. その気になれば痛みなんて完全に消しちゃえるんだ♪

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。

この本は七冊目の関数型イカ娘の本でゲソ。シリーズも後半に突入したので、少し雰囲気を変えていくでゲソ! アニメ 3 期の放映が近いことを祈りつつ、関数型言語で地上を侵略しなイカ!

この本の構成について

この本は関数型とイカ娘のファンブックでゲソ。各著者が好きなことを書いた感じなので各章は独立して読めるでゲソ。以前の「λ カ娘」本がないと分からないこともないでゲソ。ただ、一般的な入門書ではないでゲソ。

第1章

ATS言語stringライブラリ探訪

— @master_q

1.1 探検の前に

ATS 言語 [*1] は依存型 (Dependent Types) と線形型 (Linear Types) をそなえたプログラミング言語で、そのコンパイルプロセスは図 1.1 のように C 言語を経由するんでゲソ。さらにフットプリントが小さく高速なバイナリを吐き出せるそうじゃなイカ。ランタイムがなく GC を無効にすることさえ可能なので、OS への依存度も小さいんでゲソ。実際、mbed マイコンプラットフォーム [*2] や 8-bit AVR を搭載した Arduino [*3] の上で ATS プログラムを動かした実績もあるんでゲソ。

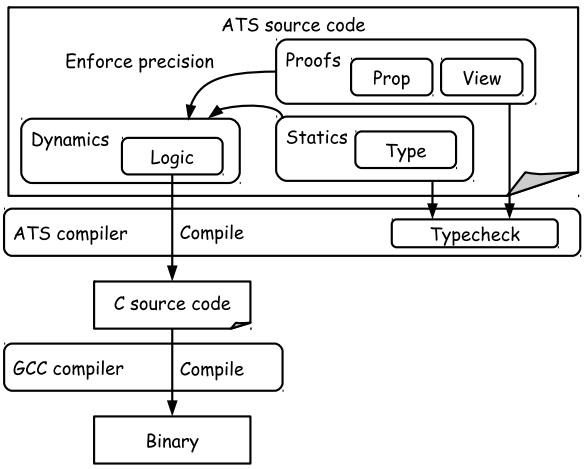


図 1.1: ATS のコンパイルプロセス

「ATS プログラミング入門 [*4]」という良い入門向けドキュメントはあるのでゲソが、これだけで依存型と線形型を駆使した ATS プログラミングができるようになるカ？ というと、それは少し厳しいとワシは思うのでゲソ。というのも依存型と線形型を通常の関数と混ぜて書くプログラミングスタイルは ATS 独自のもので、他のプログラミング言語の経験者にはなかなか習得しづらいんじゃなイカ？ そして、この混ぜて考えるということは 1 つの関数に静的な意味と動的な意味を同時に考えて与えるということなのでゲソ。とにかく馴染みのない概念でゲソ……

そこでこの記事では、他のプログラミング言語でも馴染み深い処理である「文字列処理」に焦点をしぼって、ATS 言語におけるプログラミングの作法を知る探検をしてみようと思うでゲソ。もしこの記事でわからない事柄があったら、まずは先の「ATS プログラミング入門」を (わからないなりに) 読んでみることをおすすめするでゲソ! 今回探検する ATS2 コンパイラのバージョンは ats2-positiats-0.1.0 でゲソ。

[*1] http://www.ats-lang.org/
[*2] https://github.com/fpiot/mbed-ats
[*3] https://github.com/fpiot/arduino-mega2560-ats
[*4] http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/index.html

1.2　2 つの文字列ライブラリ

ATS2 の基盤ライブラリである ATSLIB/prelude [*5] には 2 つの文字列ライブラリがあるんでゲソ。それは ATSLIB/prelude/string [*6] と ATSLIB/prelude/strptr [*7] でゲソ。前者は不変の (immutable な) 文字列を、後者は線形型の文字列を扱うんでゲソ。

まずは簡単なプログラムを書いてみようじゃなイカ。

```
$ vi teststr.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val () = print s1
  val () = print "\n"

  val pos = strchr (s1, 'o')
  val () = println! ("The 'o' is found at ", pos, "th in '", s1, "'")

  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr teststr.dats
--snip--
The 1st translation (fixity) of [teststr.dats] is successfully completed!
The 2nd translation (binding) of [teststr.dats] is successfully completed!
The 3rd translation (type-checking) of [teststr.dats] is successfully completed!
The 4th translation (type/proof-erasing) of [teststr.dats] is successfully compl
eted!
--snip--
$ ./teststr
Hello!
The 'o' is found at 4th in 'Hello!'
s1 := Hello! / s2 := Hello?
```

このプログラムは何をやっているのかというと、だいたいイカのような処理でゲソ。

1. “Hello!” という string 型文字列のリテラルを s1 に束縛
2. s1 を print 関数を使ってコンソールに印字
3. strchr 関数を使って s1 の何番目の文字に'o' が出現するか調べる
4. s1 の指す string 型の文字列を strnptr(n) 型の文字列にコピーして s2 という名前で束縛

[*5] http://www.ats-lang.org/LIBRARY/#ATSLIB_prelude
[*6] http://www.ats-lang.org/LIBRARY/prelude/SATS/DOCUGEN/HTML/string.html
[*7] http://www.ats-lang.org/LIBRARY/prelude/SATS/DOCUGEN/HTML/strptr.html

5. s2 の 5 文字目を'?' に変更
6. s2 の指す strnptr(n) 型の文字列を strptr 型に変換して s3 という名前で束縛
7. s3 の strptr 型文字列を free 関数で解放

この teststr.dats プログラムは単なる ML もどき文法のプログラムに見えるでゲソが、実はちゃんと依存型と線形型を使っている、より安全なプログラムなのでゲソ。既に string ライブラリ側に依存型と線形型による強制が仕込まれているので、ライブラリを使うプログラマが依存型と線形型を意識しなくてもプログラミングができるのでゲソ。

ところで、なぜコンパイル時検査がうれしいのでゲソ？ テストを書くことによっても同様のエラーを実行時に検出できるのではなイカ？ コンパイル時検査と実行時検査とは少し似ているけれど違う手法なのでゲソ。テストは実行時にエラーを検出する手法でゲソ。契約プログラミングも実行時にエラーを検出するしくみでゲソ。一方で、依存型や線形型のような型を使ったコンパイル時検査はコンパイル時に静的なエラーを検出できるのでゲソ。

実行時検査であるテストのことを思い出してみるでゲソ。テストとは、プログラムのコンパイルが終わった後、実際にプログラムを実行することで検査を行なうことでゲソ。テストにはブラックボックステストとホワイトボックステストがあり、前者の方法であればプログラムの詳細を知らなくても実行時検査を行なうことができたでゲソ。しかし検査の網羅率であるテストカバレッジを向上させるためには時にホワイトボックステストを行なう必要があり、さらには結合テストではなくプログラムをモジュールに分割してテストを行なう必要もあったでゲソ。モジュールテストやユニットテストを行なう場合にはモジュールが依存する先をエミュレートするテストスタブを作る必要も生じることがあったでゲソ。これらのテスト戦略はプログラムを作る前に綿密に計画を立てる必要があったじゃなイカ。またテストは例示によって検査を行なうためテストカバレッジを向上させるのにはおのずと限界があったでゲソ。

一方、コンパイル時検査はプログラムの静的な性質そのものを検査できるのでゲソ! 簡単な例では「この関数の引数には NULL が入ってきてはならない」というような静的な性質は依存型によって強制することができるでゲソ。もう 1 つの簡単な例では、「このリソースは必ず hoge_close 関数によって解放されなくてはならない」というようなリソースに対する制約は線形型によって強制することができるんでゲソ。このような静的な性質はコンパイル前には意味論として扱うことができるのでゲソが、コンパイル後の実行時には意味論として見えなくなってしまうことが多いのでゲソ。

もちろん全ての不具合をコンパイル時に型によって検査できる訳ではないでゲソ。例えば外部からの入力がどのようなものであるかはコンパイル時には予測できないでゲソ。また内部でスレッドを使っている場合、スレッドとスレッドがどのように並列/並行実行されうるかもコンパイル時に予測することができないでゲソ。このような場合には静的に検証できるような特別な仕組みを作るか、やはりテストによって実行時検査を行なう必要があるのでゲソ。ここで強調しておきたいのは、実行時検査に限界があるのと同様に、コンパイル時検査で検出できる不具合の種類にも限界があるということでゲソ。

とはいえワシの思うところでは、型によるコンパイル時検査が使える場面では使うべきでゲソ。型による静的な性質の検査はいわば全通りテストのようなものなので、どのような実行時検査よりも網羅率が高くなるはずじゃなイカ。しかもその記述は性質そのものを表現しているので同等のテストと比較してもはるかに簡単でゲソ。そして、静的な検査で網羅できない範囲を動的な性質を扱えるテストでおぎなうのでゲソ。

さて、依存型と線形型によるコンパイル時検査の効果を確かめるために、少しいたずらをしてこのサンプルプログラムに不具合を混入させてみようと思うでゲソ。strnptr(n) 型の文字列 s2 の範囲外に文字を書き込んでみたのがソースコード teststr_e1.dats でゲソ。

```
$ vi teststr_e1.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[6] := '?' // <= Changed.
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e1 teststr_e1.dats
--snip--
The 2nd translation (binding) of [teststr_e1.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e1.dats: 119(line=5, offs=12) -- 124(line=5, offs=17):
 error(3): unsolved constraint: C3NSTRprop(main; S2Eapp(S2Ecst(<); S2EVar(4070
 ->S2Eintinf(6)), S2EVar(4069->S2Eint(6))))
typechecking has failed: there are some unsolved constraints: please inspect
 the above reported error message(s) for information.
```

すばらしいことにコンパイルエラーになったでゲソ。エラーメッセージは「“ 6 < 6 ” という強制が main において解決できない」と言っているでゲソ。束縛名が見えないのでなんだかわけがわからないでゲソが、これは依存型のエラーでゲソ。後に調べるでゲソが、strnptr_set_at_gint 関数に割り当てられている全称量化の条件を “s2[6]” が満たしていないためにエラーになっているんじゃないカ。

次のエラーは線形型のリソース s3 を解放しないままプログラムを終了してしまった例でゲソ。

```
$ vi teststr_e2.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
//  val () = free s3 // <= Changed.
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e2 teststr_e2.dats
--snip--
The 2nd translation (binding) of [teststr_e2.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e2.dats: 59(line=2, offs=22) -- 250(line=9, offs=2):
 error(3): the linear dynamic variable [s3$3422(-1)] needs to be consumed but
 it is preserved with the type [S2Eapp(S2Ecst(strptr_addr_vtype); S2EVar(4075))]
 instead.
```

これもすばらしいことにコンパイル時エラーになったでゲソ! エラーメッセージは「線形の動的な値 s3 は消費 (consume) されるべきなのに、型 strptr_addr_vtype として残ってしまっている」と言っているでゲソ。これも後に調べるでゲソが線形型のリソースは生成したら必ずどこかで消費する必要があるのでゲソ。この線形型の特性によって strptr 型は文字列を格納したメモリ領域というリソースを安全に管理できるのでゲソ。

また一度解放したリソースを再度使おうとしてもエラーになるでゲソ。

```
$ vi teststr_e3.dats
#include "share/atspre_staload.hats"
implement main0 () = {
  val s1 = "Hello!"
  val s2 = string1_copy s1
  val () = s2[5] := '?'
  val s3 = strnptr2strptr s2
  val () = s2[5] := '=' // <= Changed.
  val () = println! ("s1 := ", s1, " / s2 := ", s3)
  val () = free s3
}
$ patscc -DATS_MEMALLOC_LIBC -o teststr_e3 teststr_e3.dats
--snip--
The 2nd translation (binding) of [teststr_e3.dats] is successfully completed!
/home/kiwamu/tmp/teststr_e3.dats: 172(line=7, offs=12) -- 175(line=7, offs=15):
 error(3): the linear dynamic variable [s2$3421(-1)] is no longer available.
```

このエラーメッセージは「線形の動的な値 s2 はもはや有効ではない」と言っているでゲソ。teststr_e3.dats のソースコードでは s2 を解放した後に使おうとしているでゲソ。strnptr2strptr 関数によって s2 は解放され、その直後の行で s2 の 5 文字目に'=' を上書きしようとしていて、この行でエラーが発生しているでゲソ。ちょっと注意してほしいのはここで言うリソースとはメモリリソースに限らない、ということでゲソ。生成されて消費されるものはなんであれリソースと見做すことができる可能性があるのでゲソ。

この記事では teststr.dats プログラムの動作を一つずつ追っていくことで、ATS プログラムの成り立ちを調べてみようと思うでゲソ。

1.3 最初の地図を手に入れよう

まずは ATS2 コンパイラのソースコードからライブラリに対応するソースコードの在処を確かめるでゲソ。

```
$ wget http://downloads.sourceforge.net/project/ats2-lang/ats2-lang/ats2-\
postiats-0.1.0/ATS2-Postiats-0.1.0.tgz
$ tar xf ATS2-Postiats-0.1.0.tgz
$ cd ATS2-Postiats-0.1.0
$ find prelude -name "string.*"
prelude/CATS/string.cats
prelude/DATS/string.dats
prelude/SATS/string.sats
```

```
$ find prelude -name "strptr.*"
prelude/CATS/strptr.cats
prelude/DATS/strptr.dats
prelude/SATS/strptr.sats
```

ふむふむ。なにやら 3 種類のファイル dats, sats, cats があるようでゲソ。このファイルの種類については「ATS プログラミングチュートリアル [*8]」の 2 章に説明があるじゃなイカ。

かいつまんで説明すると、sats は静的な要素のみを含むファイルで C 言語のヘッダファイルに相当すると考えるのが自然でゲソ。もっとも sats ファイルをコンパイルしても C 言語のヘッダファイルになる訳ではないことに注意でゲソ。dats ファイルは sats ファイルに配置できる要素を全て含むことができ、さらに動的な値と関数の定義を含むことができるでゲソ。つまり宣言は sats ファイルに、定義は dats ファイルに書くようにするとわかりやすいソースコード構成になるんじゃなイカ。最後に残った cats ファイルというのは ATS コンパイラとして予約された拡張子ではなく、先の dats と sats ファイルに対応する C 言語ソースコードを表わす慣習としての拡張子でゲソ。

探検する順番としては、最初に sats ファイルを読んでインターフェイスを理解し、次に dats ファイルで実装を理解して、最後に ATS 言語では書けなかったプリミティブについて cats ファイルを補完的に参照すると良さそうじゃなイカ。

また少し天下り的でゲソがイカの位置には prelude の基盤となるソースコードがあるでゲソ。ここも適宜参照するでゲッソ。

```
$ ls prelude/*ats
prelude/basics_dyn.sats  prelude/basics_sta.sats  prelude/macrodef.sats
prelude/basics_gen.sats  prelude/fixity.ats       prelude/params.hats
prelude/basics_pre.sats  prelude/lmacrodef.sats
```

最後にもう一つ、ATS の prelude ライブラリの C 言語側実装はイカのコードに依存しているでゲソ。ここも適宜参照するでゲソ。

```
$ ls ccomp/runtime/
pats_ccomp_basics.h          pats_ccomp_memalloca.h
pats_ccomp_config.h          pats_ccomp_runtime.c
pats_ccomp_exception.h       pats_ccomp_runtime2_dats.c
pats_ccomp_instrset.h        pats_ccomp_runtime_memalloc.c
pats_ccomp_memalloc.h        pats_ccomp_runtime_trywith.c
pats_ccomp_memalloc_gcbdw.h  pats_ccomp_typedefs.h
pats_ccomp_memalloc_libc.h
```

1.4 string 型とは？

最初に teststr.dats で使っていたのは不変の文字列 string 型であったでゲソ。string 型を扱う ATSLIB/prelude/string を読んでみるでゲソ! まずは”string” という型の実体を調べてみようじゃなイカ。……と思ったのでゲソが、prelude/SATS/string.sats には定義がないようでゲソ。探してみると prelude/basics_sta.sats に定義を発見したでゲッソ!

[*8] http://jats-ug.metasepi.org/doc/ATS2/ATS2TUTORIAL/index.html

```
(* File: prelude/basics_sta.sats *)
abstype
string_type = ptr // = char* in C
abstype
string_int_type (n: int) = string_type
//
stadef string0 = string_type
stadef string1 = string_int_type
stadef string = string1 // 2nd-select
stadef string = string0 // 1st-select
typedef String = [n:int] string_int_type (n)
typedef String0 = [n:int | n >= 0] string_int_type (n)
typedef String1 = [n:int | n >= 1] string_int_type (n)
```

まず string 型には string0 型と string1 型に分類できるでゲソ。どちらも動的な実体は ptr つまり単なるポインタなのでゲソが、静的な意味は両者で異なるじゃなイカ。string0 型は単に実体がポインタであるという意味しか持っていないでゲソ。ところが string1 型は静的変数 n に依存した型で、n はどうやら文字列の長さを静的に表現した静的変数のようでゲソ。この静的な変数の使い道はこの型定義からだけでは想像しにくいので、関数インターフェイスを調べるときに一緒に解説しようじゃなイカ。

1.5 string 型の値をコンソールに印字する

この string 型の値、つまり不変の文字列をコンソールに印字するには、teststr.dats ソースコードから明らかなでゲソが、単に string 型の値に print 関数を適用するだけだったゲソ。じゃあ string 型に対応する print 関数の宣言はどこにあるんでゲソ？ prelude/SATS/string.sats にあるじゃなイカ。

```
(* File: prelude/SATS/string.sats *)
fun print_string (x: string): void = "mac#%"
overload print with print_string of 0
```

汎用の print 関数は通常の関数ではなく、様々な関数実体によってオーバーロードされているんでゲソ。今回は string 型の値に対して print 関数を適用したので、print_string 関数が使われるんじゃなイカ。

ではこの print_string 関数の定義はどこにあるか、というと実は ATS 言語では実装されていないんでゲソ。こんな時は sats でも dats でもなく cats ファイルの出番でゲソ。つまり ATS 言語における宣言に対応する定義は C 言語で実装されているんでゲソ。

```
/* File: prelude/CATS/string.cats */
ATSinline()
atsvoid_t0ype
atspre_fprint_string
(
  atstype_ref out, atstype_string x
) {
  int err = 0 ;
```

```
  err += fprintf((FILE*)out, "%s", (char*)x) ;
  return ;
} // end of [atspre_fprint_string]
#define atspre_print_string(x) atspre_fprint_string(stdout, (x))
```

ここまでコードを読んできて、print 関数はそれぞれの階層に分割して実装されていることがわかったでゲソ。ちょっと図にまとめてみなイカ (図 1.2)？ この図から明らかでゲソが、string 型に対する print 関数は prelude/SATS/string.sats ファイルにて宣言され、prelude/CATS/string.cats ファイルにて定義されているんでゲソ。

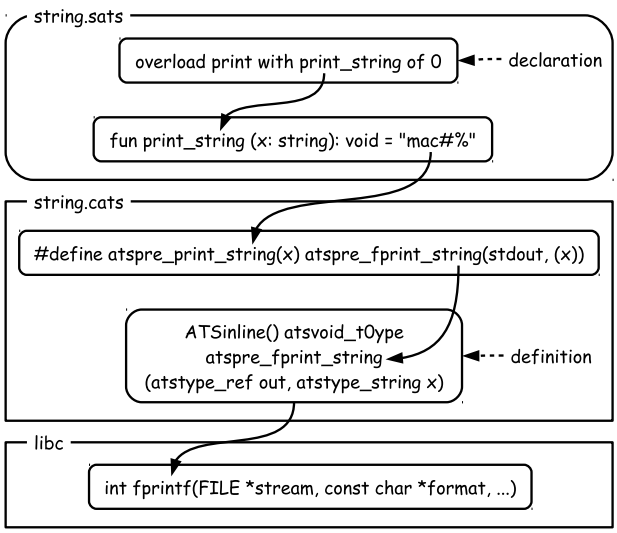


図 1.2: print 関数の実装

つまりこの print 関数は C 言語の実装に ATS によるインターフェイスを割り当てているんじゃなイカ。よくある言語処理系では print 関数に対応する C 言語実装はその言語処理系ランタイムに含まれることが多いでゲソ。ところが、ATS ではこのような機能もランタイムではなく基盤ライブラリの中で実装されているんでゲソ。ATS のコードは評価器が不要で、C 言語の相当する表現にコンパイルされ、さらには線形型を使うことで GC を使わないことも可能でゲソ。このような特徴のために ATS にはその動作に必須なラインタイムというものが存在しないんでゲソ!

1.6 strchr 関数

print 関数は prelude/CATS/string.cats ファイル、つまり C 言語で定義されていたでゲソ。それでは全ての実装は C 言語で書く必要があるのカ？ というとそんな事はなく、ATS 言語の実装を prelude/DATS/string.dats にて行なうことも可能でゲソ。今度はそんな例を見てみようじゃなイカ。teststr.dats では strchr という関数を使っていたでゲソ。

```
(* prelude/SATS/string.sats *)
fun{
} strchr{n:int}
  (str: string (n), c0: char):<> ssizeBtwe (~1, n)
```

この strchr 関数はイカのような libc における同名の関数 [*9] と同じインターフェイスであることがわかるでゲソ。

[*9] http://pubs.opengroup.org/onlinepubs/9699919799/functions/strchr.html

```
/* libc */
#include <string.h>
char *strchr(const char *s, int c);
// Upon completion, strchr() shall return a pointer to the byte,
// or a null pointer if the byte was not found.
```

しかし、よくよく ATS 側の strchr インターフェイスを見てみると”{n:int}” のように依存型の全称量化 (universal quantification) が指定されているでゲソ。この n は種 int の静的な変数と呼ばれ、この変数のスコープは strchr 関数の型宣言の中のみでゲソ。この n は型宣言において二箇所で使われ、それは”string (n)” と”ssizeBtwe (~1, n)” じゃなイカ。

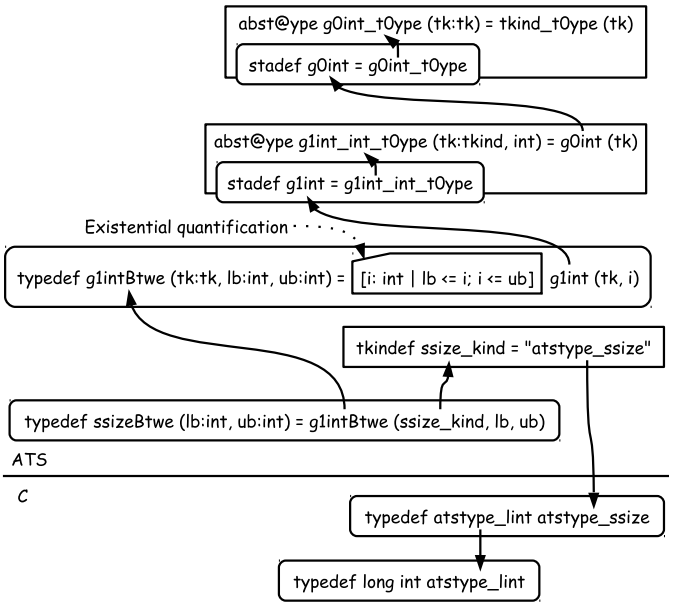


図 1.3: ssizeBtwe 型の構造

前者は静的な変数 n に依存した string 型を意味していて、これはつまるところ「長さが n であるという静的な意味を持つ文字列」となんらかの文字の 2 つの引数を strchr 関数が取るということを意味しているんじゃなイカ。それでは後者つまり返値に対して、静的な変数 n はどんな制約を付けているかは、少し型定義は深いでゲソが prelude/basics_pre.sats, prelude/basics_sta.sats, ccomp/runtime/pats_ccomp_typedefs.h あたりを読むと理解できるでゲソ。

ssizeBtwe 型の構造をまとめてみると図 1.3 が得られるはずでゲソ。ここでの注目すべき 1 つ目のポイントは存在量化 (existential quantification) の部分でゲソ。この存在量化によって ssizeBtwe 型は lb 以上 ub 以下な i に依存した g1int 型になるでゲソ。2 つ目のポイントは先の g1int 型には C 言語の long int を実体とする kind が渡されているということでゲソ。つまりコンパイル後の型は long int になるのでゲソが、その C 言語の型に依存型の静的な意味を付与するのが、g1int 型ということになるんじゃなイカ。

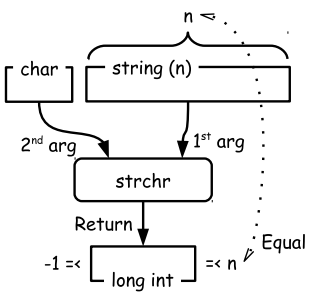


図 1.4: strchr のインターフェイス

この依存型の強制によって strchr 関数にはどのようなインターフェイスが割り当てられたことになるんでゲソ？ それはつまり図 1.4 のようになるんじゃなイカ。strchr は文字列と文字を取り、その文字が表われた文字列の場所を long int 型で返すのでゲソが、この long int の値は -1 から文字列の長さである n の間の範囲に収まることが依存型によって保証されているんでゲソ。しかも strchr 関数内外の実装がこの範囲外の値を取る場合にはコンパイルエラーになるんでゲソ!

ATS の作者である Hongwei の言葉を借りれば、このインターフェイスは「strchr 関数の引数と返値の間に依存型を使って静的な意味を強制している」ことになるんじゃなイカ？ ATS における設計は常にこの「静的な意味の強制」を意識することが重要になるんでゲソ。

今度は strchr 関数の実装を見てみようじゃなイカ。

```
(* File: prelude/SATS/string.sats *)
praxi
lemma_string_param{n:int}(string n): [n >= 0] void
castfn
string2ptr (x: string):<> Ptr1

(* File: prelude/DATS/string.dats *)
implement{
} strchr {n} (str, c0) = let
  prval () = lemma_string_param (str)
  extern fun __strchr (string, int):<> ptr = "mac#atspre_strchr"
  extern fun __sub (ptr, ptr):<> ssizeBtw (0, n) = "mac#atspre_sub_ptr_ptr"
  val p0 = string2ptr(str)
  val p1 = __strchr (str, (char2int0)c0)
in
  if p1 > the_null_ptr then __sub (p1, p0) else i2ssz(~1)
end

/* File: prelude/CATS/string.cats */
#define atspre_strchr strchr

/* File: prelude/CATS/pointer.cats */
ATSinline()
atstype_ssize
atspre_sub_ptr_ptr
  (atstype_ptr p1, atstype_ptr p2) { return (p1 - p2) ; }
```

ロジック本体は prelude/DATS/string.dats で実装されているでゲソが、その中で libc の strchr(3) 関数と、ポインタの引き算のために C 言語で実装された atspre_sub_ptr_ptr() 関数を呼び出しているでゲソ。ATS 側の strchr 関数はテンプレートとして実装されていて、そのテンプレート引数は n でゲソ。この n は関数テンプレートの中で__sub の返値の型を依存型で強制するために使っているでゲソ。このようなテンプレート実装を総称テンプレート実装と呼ぶでゲソ。

しかし不思議じゃなイカ？ ATS の strchr 関数も libc の strchr(3) 関数を使用してしまっているでゲソ。それなのに、普段吐き出されている ATS バイナリは strchr の未定義シンボルを含んでいないでゲソ。そのトリックは ATS2 コンパイラがコンパイル時に未到達コードを削除してくれることにあるんじゃなイカ。ほとんどの ATS プログラムは strchr 関数を使わないので、prelude ライブラリ内の strchr コードはコンパイル時に削除されるんでゲソ。そのため libc の strchr(3) 関数も使わないんでゲソ。一見良いことずくめのこの機能、場合によっては火の入らない ATS コードの定義エラーを隠してしまうことにもなるため、注意が必要でゲソ。

1.7 string 型から strnptr 型へ

さて、少し変わった関数 string1_copy を見てみようじゃなイカ。

```
(* File: prelude/SATS/strptr.sats *)
fun{
} string1_copy
  {n:int} (xs: NSH(string(n))):<!wrt> strnptr (n)
```

インターフェイスを見るかぎり、string1_copy 関数は string(n) 型の文字列を取り、同じ長さの strnptr(n) 型の文字列を返すようでゲソ。この strnptr(n) 型は不変の文字列ではなく線形型の文字列なのでゲソ。ようこそ線形型の海へ! でゲッソ。

新しい型 strnptr(n) が出てきたので、型定義を見てみようじゃなイカ。

```
(* File: prelude/basics_sta.sats *)
absvtype
strnptr_addr_int_vtype (l:addr, n:int) = ptr
stadef strnptr = strnptr_addr_int_vtype
vtypedef strnptr (n:int) = [l:addr] strnptr (l, n)
vtypedef Strnptr0 = [l:addr;n:int] strnptr (l, n)
vtypedef Strnptr1 = [l:addr;n:int | n >= 0] strnptr (l, n)
```

この定義から strnptr(n) 型は「strnptr(l, n) となるようなアドレス l が存在する」と読めるでゲソ。そしてその抽象型の実体は ptr、つまりポインタでゲソ。そしてこの抽象型の定義は abstype ではなく absvtype でなされているでゲソ。この型は抽象観型 (abstract viewtypes) と呼ばれ、線形型の抽象型でゲソ。つまりこの抽象観型 strnptr(l, n) は一度確保されたら、どこかで明示的に消費する必要があることを意味しているでゲソ。線形型の消費を忘れるとコンパイルエラーになるでゲソ。

それじゃあ string1_copy 関数の実装を見てみようじゃなイカ。

```
(* File: prelude/DATS/string.dats *)
implement{}
string1_copy
  {n} (str) = let
  val n = string1_length (str)
  val n1 = succ(n)
  val (pf, pfgc | p) = malloc_gc (n1)
  val _(*p*) = $effmask_wrt (memcpy (p, string2ptr(str), n1))
in
  castvwtp_trans{strnptr(n)}((pf, pfgc | p))
end

implement{}
string1_length
  {n} (str) =
  __strlen (str) where {
  extern fun __strlen (str: string n):<> size_t (n) = "mac#atspre_strlen"
```

```
}
```

一旦、線形型は無視してこの実装が何をしているのか理解してみるでゲソ。string1_copy 関数は引数 str の長さを string1_length 関数を使って測り、それより 1 長いサイズを malloc_gc 関数で確保しているようでゲソ。さらに先に確保したメモリ領域に str の中身を memcpy した後、そのポインタを strnptr(n) 型の値として返しているでゲソ。このとき使ったメモリ操作の関数 malloc_gc, memcpy, __strlen の動的な部分は図 1.5 のように C 言語を使って実装されているでゲソ。

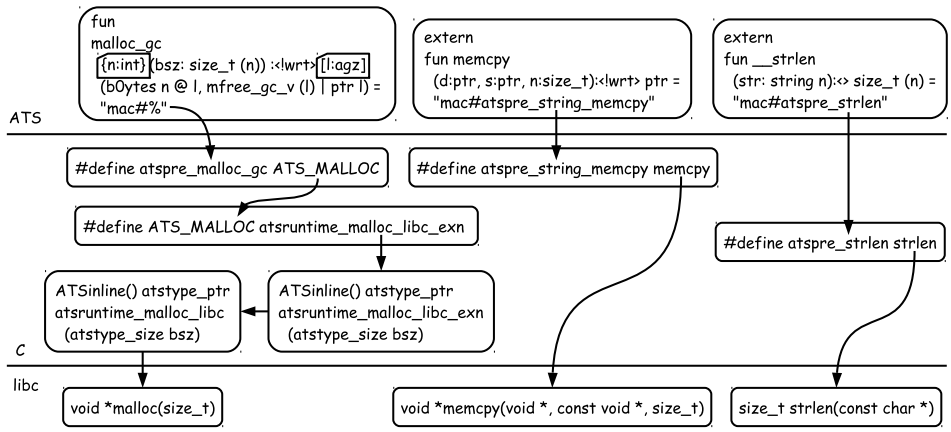


図 1.5: string1_copy 関数の使うメモリ操作関数

これで string1_copy 関数の動的な部分は理解できたのでゲソが、その静的な部分はさっぱりわかっていないじゃなイカ。順に調べていこうじゃなイカ。

1.8 malloc_gc 関数の静的な意味

malloc_gc 関数の型を見てみるでゲソ。

```
(* File: prelude/SATS/memory.sats *)
fun
malloc_gc
  {n:int}
(
  bsz: size_t (n)
) :<!wrt>
  [l:agz]
(
  b0ytes n @ l, mfree_gc_v (l) | ptr l
) = "mac#%"
```

唯一の引数 bsz の型はいいとして、返値の型がよくわらないでゲソ。この縦棒はなんでゲソ？ 実は ATS では縦棒の左右で全く異なる世界を表わすのでゲソ。左側は証明、右側は動的な値なのでゲソ。

この縦棒は返値だけではなく、引数でも使われるでゲソ。例えば、malloc_gc の対になる関数、mfree_gc は次のようなインターフェイスを持ち、引数に証明と動的な値の両方を取るのでゲソ。

```
(* File: prelude/SATS/memory.sats *)
fun
mfree_gc
  {l:addr}{n:int}
(
  pfat: b0ytes n @ l
, pfgc: mfree_gc_v (l) | ptr l
) :<!wrt> void = "mac#%"
```

malloc_gc 関数の返値の型を 1 つずつ調べてみようと思うでゲソ。はじめに、動的な値の型である “ptr l” をまずは調べようじゃなイカ。

```
(* File: prelude/basics_sta.sats *)
tkindef ptr_kind = "atstype_ptrk"
abstype ptr_type = tkind_type (ptr_kind)
stadef ptr = ptr_type // a shorthand
abstype ptr_addr_type (l:addr) = ptr_type
stadef ptr = ptr_addr_type // a shorthand
typedef Ptr = [l:addr] ptr (l)
typedef Ptr0 = [l:addr | l >= null] ptr (l)
typedef Ptr1 = [l:addr | l >  null] ptr (l)

/* File: ccomp/runtime/pats_ccomp_typedefs.h */
typedef void *atstype_ptrk ;
```

addr はおそらく組み込み型で、これは本当にメモリアドレスを表わす型でゲソ。この addr 型の値、つまりどこかのメモリ位置に ptr_addr_type を適用すると ptr_type 型を成し、この型には ptr という短縮名が与えられているでゲソ。ptr_type は抽象型でその動的な実体は C 言語の atstype_ptrk 型、つまり void *ポインタでゲソ。この抽象型は abstype で宣言されているので、線形型ではなく通常の型でゲソ。

次に駐観 (At-views) と呼ばれる線形の証明 “b0ytes n @ l” を調べてみようじゃなイカ。

```
(* File: prelude/SATS/memory.sats *)
typedef b0ytes (n:int) = @[byte?][n]

(* File: prelude/basics_sta.sats *)
tkindef
byte_kind = "atstype_byte"
abst@ype
byte_t0ype = tkind_t0ype (byte_kind)
stadef byte = byte_t0ype

(* File: prelude/basics_pre.sats *)
absview // S2Eat
at_vt0ype_addr_view (vt@ype+, addr)
```

```
stadef @ = at_vt0ype_addr_view // HX: @ is infix
```

@ は抽象観の述語で、vt@ype+ の値を左辺に addr の値を右辺に取って抽象観を成すんでゲソ。vt@ype はフラットなメモリレイアウトを持つ観 (Views) という意味でゲソ。@[*a*][*n*] で表わされる配列はフラットなレイアウトを持つので、@ の左辺になれるでゲソ。この形の線形証明を駐観と呼び、*T*@*L* はアドレス L に T という型の値があるということを表わしているでゲソ。ポインタ p に値 v を書き込むには !*p* := *v* のように書けばよいのでゲソが、このとき対応する駐観が静的な環境に見つからない場合にはコンパイルエラーになるんでゲソ。! と := の定義がないか、prelude ライブラリを探してみたのでゲソが見つからなかったでゲソ。これら 2 つの演算子はおそらくコンパイラプリミティブだと思われるでゲソ。

脱線してしまったでゲソ。malloc_gc 関数はポインタと対応する駐観を同時に返していたでゲソ。ポインタをデリファレンスするためにはこの対応する駐観が必要で、その生成はポインタの生成と同時であるということでゲソ。もっと抽象的に考えると、malloc_gc 関数で確保されるメモリ領域の静的な意味はアドレスに依存した駐観で表わされ、動的な意味はアドレスに依存したポインタで表わされるということになるでゲソ。

残った “mfree_gc_v (l)” は何でゲソ？ と、ソースコードを調べてみたのでゲソが、有意なコードはイカのような抽象観の定義とその観の証明を単に消費する mfree_gc_v_elim 証明関数しか見つからなかったでゲソ。これはワシの想像でゲソが、ATS2 コンパイラは Boehm GC [*10] を使っているのでゲソが、これを別の GC に置き換える計画があるのではなイカ？ そのために、静的な意味を線形型で表現できるようにとりあえず GC で回収すべきメモリの型を定義してあるのではなイカと想像するでゲソ。

```
(* File: prelude/basics_sta.sats *)
absview mfree_gc_addr_view (addr)
stadef mfree_gc_v = mfree_gc_addr_view

(* File: prelude/basics_dyn.sats *)
// HX: returning the pf to GC
praxi
mfree_gc_v_elim
  {l:addr} (pf: mfree_gc_v l):<prf> void
```

1.9 castvwtp_trans 関数とは何か

最後に残った castvwtp_trans 関数を調べてみようじゃなイカ。

```
(* File: prelude/DATS/string.dats *)
macdef castvwtp_trans = $UN.castvwtp0

(* File: prelude/SATS/unsafe.sats *)
sortdef t0p = t@ype and vt0p = viewt@ype
castfn castvwtp0 {to:vt0p}{from:vt0p} (x: INV(from)):<> to
```

[*10] http://www.hboehm.info/gc/

```
(* File: prelude/basics_pre.sats *)
absvt@ype invar_vt0ype_vt0ype (a:vt@ype) = a
vtypedef INV (a:vt@ype) = invar_vt0ype_vt0ype (a)
```

この関数は ATSLIB/prelude/unsafe ライブラリの castvwtp0 関数の単なるラッパーで、castvwtp0 関数は線形型の値を別の線形型にキャストするんでゲソ。つまり線形証明のキャストでゲソ! 先の string1_copy 関数の実装では、はじめに malloc_gc 関数を呼び出して線形証明 pf と pfgc を取得していたでゲソ。その後、string1_copy 関数が返る直前に strnptr(n) という線形証明でキャストしているんじゃなイカ。つまり malloc_gc 関数が返した線形型ではなく、文字列専用の線形型を使ってリソースのライフサイクルを管理しようという訳でゲソ。

1.10 破壊的な文字列変更

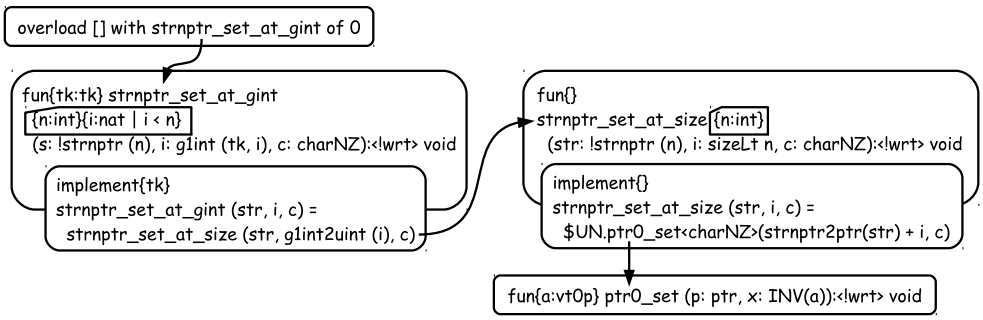


図 1.6: strnptr_set_at_gint 関数近傍

線形文字列の添字アクセスは strnptr_set_at_gint 関数でオーバーロードされているでゲソ。その先の処理をたどっていくと図 1.6 が描けたでゲソ。行き着いたのは prelude/DATS/unsafe.dats の ptr0_set 関数でイカのような実装になっているでゲソ。

```
(* File: prelude/DATS/unsafe.dats *)
implement
{a}(*tmp*)
ptr0_set
  (p, x) = () where {
  val [l:addr]
    p = g1ofg0_ptr(p)
  prval (pf, fpf) = __assert () where {
    extern praxi __assert (): (a? @ l, a @ l -<lin,prf> void)
  } // end of [prval]
  val () = !p := x
  prval () = fpf (pf)
} // end of [ptr0_set]

(* File: prelude/SATS/pointer.sats *)
castfn g1ofg0_ptr (p: ptr):<> Ptr0
```

```
(* File: prelude/basics_sta.sats *)
typedef Ptr0 = [l:addr | l >= null] ptr (l)
```

上記の ptr0_set 関数の実装では__assert という内部の証明関数を使って、a? @ l という証明 pf と、a @ l という証明を消費する証明関数 fpf を無理矢理作っているでゲソ。証明 pf によってポインタ p をデリファレンスすることができ、さらに証明関数 fpf によって証明 pf を消費することができるのでゲソ。つまり ptr0_set 関数はどのようなポインタにも任意の型を書き込むことができるのでゲソ! なるほど確かに unsafe な関数じゃなイカ。

では、strnptr_set_at_gint 関数も安全でない関数になってしまうのカ？ というとそうではないのでゲソ。string_get_at_gint 関数にはイカの型がついていたじゃなイカ。

```
(* File: prelude/SATS/strptr.sats *)
fun{tk:tk}
strnptr_set_at_gint
  {n:int}{i:nat | i < n}
  (s: !strnptr (n), i: g1int (tk, i), c: charNZ):<!wrt> void
```

整数 n と自然数 i が全称量化で導入されているでゲソ。引数 s の型には'!' が付いているので、この線形リソースは消費されないことを意味しているでゲソ。つまり第 1 引数である文字列 s の長さは n で、この文字列は消費されず、第 2 引数である整数 i は n 未満でなければならないでゲソ。

teststr_e1.dats で起きたエラーはこの strnptr_set_at_gint 関数の型として与えられた「第 2 引数である整数 i は第 1 引数の文字列サイズ未満でなければならない」という制約を解決できなかったからでゲソ。teststr_e3.dats で起きたエラーは第 1 引数 s に付与されていた線形リソースが、strnptr_set_at_gint 関数に渡った時点では既に消費されてしまっていたからでゲソ。これまでコンベンションとして扱われていた事柄も、このように依存型と線形型をインターフェイスに適切に与えることで、コンパイル時検査に昇格させることができるんじゃなイカ!

1.11　strnptr 型から strptr 型へ

証明キャスト関数 strnptr2strptr を使って strnptr 型を strptr 型に変換するでゲソ。どちらの型も実体は ptr 型でゲソが、両者は異なる抽象観型でゲソ。そのため、この strnptr2strptr 関数は strnptr 型を消費して、strptr 型を生成して割り当てていることになるでゲソ。

```
(* File: prelude/SATS/strptr.sats *)
castfn
strnptr2strptr
  {l:addr}{n:int} (x: strnptr (l, n)):<> strptr (l)

(* File: prelude/basics_sta.sats *)
absvtype
strptr_addr_vtype (l:addr) = ptr
stadef strptr = strptr_addr_vtype
absvtype
strnptr_addr_int_vtype (l:addr, n:int) = ptr
stadef strnptr = strnptr_addr_int_vtype
```

1.12 文字列の解放

最後に strptr 型を解放するために free 関数を呼び出していたでゲソ。この free 関数を strptr 型の値に適用すると、オーバーロードによって strptr_free 関数と解釈され、結局のところ図 1.7 のように C 言語の free(3) 関数が呼び出されるでゲソ。strnptr(n) 型の解放も型は違えど動的な処理は同じでゲソ。

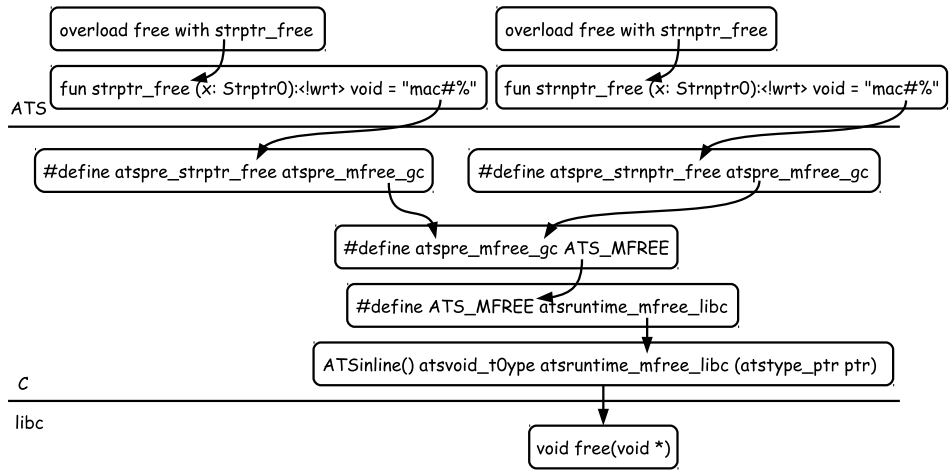


図 1.7: ATS の strptr_free 関数と C 言語の free 関数の関係

もちろん strptr_free 関数にはイカのインターフェイスが割り当てられているので、strnptr2strptr 関数で生成された線形の証明 Strptr0 が消費されるんでゲソ。つまり strptr_free 関数の動的な意味は「libc の free(3) を呼び出してメモリを解放する」であり、静的な意味は「Strptr0 の線形証明を消費する」ということなのでゲソ。

```
(* File: prelude/SATS/strptr.sats *)
fun strptr_free (x: Strptr0):<!wrt> void = "mac#%"

(* File: prelude/basics_sta.sats *)
vtypedef Strptr0 = [l:addr] strptr (l)
stadef strptr = Strptr0
```

1.13 その先へ

今回の探検で、string ライブラリの全てが理解できた訳ではないでゲソ。イカの項目についてはまだ全く理解できていないじゃなイカ。

1. ": <>" の中にあるマークは何か
2. $effmask_wrt とは何か
3. "mac#%" とは何か
4. type, sort, kind について
5. tkindef について

上記リストの項目 1,2 については関数の効果 (Function Effects) [*11] について調べることで仕組みがわかるかもしれないでゲソ。その他の項目は ATS2 コンパイラ本体のソースコードを読む他なさそうでゲソ。特に kind は重要な概念のようで、例えば ATS の抽象型 byte_t0ype の実体として C 言語の型 atstype_byte を指定しているように先のソースコードから読めるでゲソ。この機能のをうまく使えば、C 言語の型をより簡単に扱えるようになるかもしれないじゃなイカ。

またこの記事読んできて、賢明な読者はこの線形証明を消費する関数や線形証明のキャストによって線形型が全くの無力になってしまう可能性があることに気づいていると思うでゲソ。当然、乱用すると線形型で管理しているはずだった動的なリソースのライフサイクルが線形型の静的なライフサイクルと一致しないことになってしまうじゃなイカ。これはワシの感想なのでゲソが ATS 言語の提唱する安全性は、言語そのものによって担保されるものではなく、言語を利用してライブラリやフレームワーク作るプログラマによって担保されるものなのではなイカ？ ということでゲソ。ATS では C 言語と同じように言語専用のヒープがなく、malloc_gc のようなライブラリ関数によって形づくられていたでゲソ。GC を使わない場合にはこれらのライブラリに頼れないということを意味するので、メモリリソースの管理はプログラマが線形型を使って別途専用に作る必要があるでゲソ。この線形型を適切にライブラリに適用する作業はプログラマの能力と良心に依存しているでゲソ。これは依存型についても同様でゲソ。もっと高階な表現をすると、動的なコードの中に静的な意味を適切に見出す作業はプログラマに完全に依存しきっているのでゲソ。これは安全性をうたう ATS という言語に大きな危険性とそして大きな自由を提供していることになるでゲソ。プログラムの低位では証明キャストのような危険な機能を使って安全なインターフェイスを作り出し、プログラムの高位ではその安全なインターフェイスを使ってプログラミングをすることでこの危険性を小さくすることができるかもしれないじゃなイカ？ もちろん NetBSD kernel のような巨大なプログラムにおいては、そのようなプログラミングが原理的に不可能な可能性もあるでゲソ。とにかく今はこの地上で誰一人その解を持っていないのでゲソ。

1.14 ライセンス

ATS2 のソースコードは GPL3 で配布されているでゲソが、原作者 Hongwei の好意で ATS2 コンパイラのソースコードを引用しているこの記事には GPL3 ライセンスは適用されていないでゲソ。

1.15 参考文献

- The ATS Programming Language [*12]
- JATS-UG - Japan ATS User Group [*13]
- ATS プログラミング入門[*14]
- ATS プログラミングチュートリアル[*15]
- ML プログラマ向け ATS 言語ガイド[*16]
- 状態を持つ観 (view) を通じてポインタを扱う安全なプログラミング[*17]
- Bluish Coder ブログ[*18]

[*11] https://github.com/steinwaywhw/ats-docs-and-tips/blob/master/source/features.rst#function-effects
[*12] http://www.ats-lang.org/
[*13] http://jats-ug.metasepi.org/
[*14] http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/index.html
[*15] http://jats-ug.metasepi.org/doc/ATS2/ATS2TUTORIAL/index.html
[*16] https://github.com/jats-ug/translate/blob/master/Web/cs.likai.org/ats/ml-programmers-guide-to-ats.md
[*17] https://github.com/jats-ug/translate/blob/master/Paper/SPPSV-padl05/SPPSV-padl05.md
[*18] http://bluishcoder.co.nz/tags/ats/

第2章

Haskellで状態を扱う

— @dif_engine

物語の概要と目的

この章では、Haskell で「状態」を扱うための基本を学びます。いわゆる命令型言語では、変数に計算途中の値を格納しておき、必要に応じてその変数の値を書き換える（更新する）ことでプログラムを記述することができます。このようなプログラミングスタイルは、数学的なアルゴリズム記述との親和性が高く、多くのアルゴリズムが命令型言語で実装され、実用化されてきました。

一方、Haskell では一度定義した変数を書き換える方法は用意されていません。したがって、状態を積極的に使ったアルゴリズムを Haskell で使おうとするときには何らかの工夫をする必要があります。本稿ではその中で最も基本的なものの一つである「状態を取り回す」ためのハック — 関数型プログラミング界で「状態モナド」と呼ばれているもの — について説明します。

シンプルで汎用性の高いハックが大抵そうであるように、このハックにも理論的な基盤があります。状態モナドのデザインは、圏論の概念、特にモナドと結びついた Kleisli 圏と合わせて考えるとよく理解できます。本稿では Haskell をすでにある程度習得している読者を想定し、このデザインについて説明をします。圏論についてはとくに事前知識を要求しませんが、読者がある程度の数学的な素養、たとえば理工系の大学二年生に求められる程度の理解力を持っていることは想定します。

2.1 プロローグ

2.1.1 その『状態モナド』ってのを使えば参照透明性をぶっ壊せるんだろ？

「——でもさ、Haskell の変数って参照透明でしょ？ つまり一度変数を定義したらその変数を書き換えたり出来ないわけだよね。そんなんじゃ内部状態を持つ関数とか書けそうにないけど」

さて、どう答えたものだろうか。

「不便かどうかは、主観的な話だからスルーかな。『内部状態を持つ関数』について言えば、状態モナドを使えばそういう関数を書けるし、参照透明性がそこまで重大な制約になるとは限らないわ」

「また『モナド』かー。てか Haskell ってなんでもモナドが出てくるのな」

「なんでもモナドというわけじゃないわ。モナドでは不十分な場面では、例えば Arrow なんかが使われることがあるし」

「まあそんなすごい話はともかく、まずモナドからして私はわかってないんだよなー。でさ、その『状態モナド』ってのを使えば参照透明性をぶっ壊せるんだろ？ なんかすごい話だよな」

「参照透明性をぶっ壊す——？ そんな物騒な事はしないわ。Haskell の基本的な性質はまった

く変えないで状態を扱えるようにするのが状態モナドなのよ」

「変数が書き換えられないのに状態の書き換えを実現できるって、どんな魔術なの？」

「いいえ、状態モナドに魔術なんて一欠片も含まれてないわ。だって――。」

続けようとしたが言葉に詰まってしまった。状態モナドは、わかってしまえば――難しいものではないと思う。でも全く簡単――ということもなさそうだ。参照透明性の制約の中で、状態を扱う仕組みを、きちんと説明しようと思ったら少し長い話になりそうだった。

2.1.2 大図書館にて

紅魔館は息苦しい――いつの頃からかそんなふうに思うようになった。

持病の喘息――魔術でも改造しきれなかった体の欠陥――のせいというわけではない。たしかに地下の埃っぽい図書館というのは喘息持ちにとってよい環境とはいえないが、本を読むより楽しいことを知らない私にとってはここは最高の場所だ。レミリアは「パチェ、いつまでもここにいていいのよ」と言ってくれる。他に行くあてのない身としては本当にありがたい。

それでも、ずっとここにいたら私はどうにかなってしまうのではないか――。

もしかしたらそれは、この地下室よりさらに奥深い部屋（そんな部屋があるなんて十年以上知らなかった）から出されてきたあの真っ白な少女に「あなた、300 年前にはここにいなかった子ね。自分の運命をレミリアの玩具にされたくなかったら逃げたほうがいいわよ」と言われたときに芽生えた不安なのかもしれない。

それとも、その少女を私の前に引っ張ってきて「これはねぇ、私の妹なんだけど――私に逆らったからしばらく閉じ込めておいたのよ」と言いながらレミリアが見せた屈託のない笑顔を見たときに感じた恐怖のせいかもしれない。

ここの住人はみなどこか狂ってるのではないか――ずっとここにいたら私も彼らの狂気に飲み込まれてしまうかもしれない。

いいや――本当の不安の源泉は――私自身だ。少女だった頃――今でも外見はそうなのだが――からの百年近くの月日は、日常の何気ないことにも美しさと喜びを感じ取る心の弾力を奪っていってしまった。次第に冷たく固く無感覚になってゆき、いつか私の心は完全に――。

「シケた顔してるね、パチュリー。頭痛とか？」

声の主は魔理沙だった。ここは地下室だが、採光用に空堀がある。多分そっちから入ってきたのだろう。外から持ち込まれた埃と太陽の匂い――というより皮脂が紫外線で分解されたときに発生した高級アルコールの匂い――に、なぜかホッとする。くるくるとした金髪に囲まれた顔には意志の強そうな榛色の目。頬に浮かんだ重たい金色のそばかすが顔に愛嬌を与えている。

「ひさしぶりね、魔理沙」

魔理沙はたまに遊びに来る友達だ。紅魔館の他の住民との関係は知らないけれど、私と話をするときは大抵プログラミングの話になる。

魔術とプログラミングには親近性がある。一般には、魔術というと意味不明の儀式や効果不明の呪いを思い浮かべる人が多いだろう。これは当然のことである。効果がはっきりわかるレベルの魔術を実践するためには、自らの身体が膨大な魔力を蓄積し、放出できる扱えるように改造しなければならない。市井の人が思っているのと違い、不死者へと至る階梯は魔術の目的ではなく手段なのだ。真の魔術師として歩み始めた者にとっての問題は、膨大な力を制御する方法である。計画性の

ない力の暴走は、不死者すら破滅させかねないのだ。人類が生み出した最も精緻な計画法（プログラミング）は、魔術師にとってもまた有益だろうと考え、私は魔術の研究の傍ら、この大図書館のプログラミングの本の中から気に入ったものを読むようにしている。

「それで、今はどんな言語をやってるの？」

「Haskell よ」

「Haskell かー、ちょっとは触ったことあるな。でも何に使うんだ？」

LISP 方言の一つである Scheme が好きな魔理沙——彼女のミニ八卦炉に刻まれた太極の紋章に「eval」「apply」と落書きしているのに気づいたのは私だけかもしれない——が、同じ関数型言語に分類される Haskell のことを、多少知っていても不思議ではない。

「属性魔法の重ねがけのシミュレーションをして、最適なスペルカードを生成しようと思って」説明しておこう。属性魔法というのは、自然界に存在する諸力を、それを司る精霊を通して使う魔術の流派だ。大昔の魔術師たちはこれらの精霊が実在すると考えていた節があるが、現在ではこれらの実在性の一端は我々の認識作用にあると理解されている。世界の諸力をいくつに分類するかと言うのは、一オクターブにはいくつの音が含まれているかという問題と近いかもしれない。西洋の鍵盤楽器と五線譜による教育を受けると一オクターブの音域を十二個に量子化する習慣がついてしまう。一オクターブの音域は周波数を通して物理的に規定されるが、一オクターブの音域に十二の音を感じるのは訓練による認識作用だ。世界に満ちる力を四大精霊に帰着する教育を受けた者は、その教育と訓練によって四大精霊をありありと実感出来るようになる。

これに対し、私は世界の諸力を七つに分ける流儀を独自の研究で編み出した。どの流儀であれ属性魔法では、異なる種類の、あるいは同種類の魔法を重ねがけするようにして複雑な魔術作用を実現することは共通している。属性魔法は、うまく組み合わせれば和音のように調和した魔力を形成し、失敗すれば不協和音のように破壊的な作用をもたらす。したがって、属性魔法においては、目的の状態に合わせて属性魔法を重ねる方法を計画することが研究の課題となる。十分な時間が与えられているときには属性魔法は最も精緻な作用を実現できる魔術体系だが、属性魔法の重ねがけによる副作用は予想するのが難しく、戦闘などの即応性が要求される場面では私は度々敗北を喫している。

「——でもさ、パチュリーの属性魔法と Haskell って相性わるいんじゃないか？　だって、属性魔法って『前の状態』で効果が変わるから、素朴にシミュレーションを書こうとすると、それぞれの魔法を『状態を持った関数』として書くことになるじゃん」

「ええ、そうね」

「——でもさ、Haskell の変数って参照透明でしょ？　つまり一度変数を定義したらその変数を書き換えたり出来ないわけだよね。そんなんじゃ内部状態を持つ関数とか書けそうにないけど」

こうして冒頭の会話に至る。

「——どうかな魔理沙、私の復習も兼ねて、Haskell の参照透明性の制約の中で、状態を扱う仕組みを説明してみたいんだけど」

「そうか、よろしくたのむよ」

「説明の順番としては

- モナドとは何か（p.22）
  - 圏の基礎概念（p.22）
  - モナドと Kleisli 圏（p.30）
- モナドを使って状態を持つ関数を扱う方法（p.35）
  - 状態を持つ関数（p.35）

- 状態モナド（p.37）
- 状態モナドの Kleisli 射としての『状態付き関数』（p.38）

という感じでやってみるわね」

2.2 モナドとは何か

「モナドについては色々な意見があると思うけど、圏論から説明するのが結局はわかりやすいと思うのよね」

「『急がばまわれ』ってか」

「短く言えばそういうことだけど、例えばモナドだけじゃなく Arrow も圏論的な背景を持ってるわけだし、Haskell の概念を理解するために圏論の基礎を学んでおくのは結局は効率の良い投資だと思うわ」

何かを学ぶとき、その分だけ私達の認識は変化する。人生の時間は —— 普通より多少長生きできる私たち魔法使いにとっても —— 結局のところ有限の資源だ。その限られた資源を何に使うか、その選択が私たちの知識や理解を、ひいては未来の私達の姿を決める。

「そういうことなら、せっかくだし圏論から教わることにしようかな」

2.2.1 圏の基礎概念

圏の定義

「じゃあ早速だけど圏の定義を書いてみるね」

圏の定義

圏 (けん；category) $\mathbf{C}$ は以下の **(i)(ii)(iii)(iv)** のデータからなり、**(cat1)(cat2)(cat3)** を満たすものを指す。

(i) **対象 (たいしょう；object)** からなる集合 $\mathrm{Ob}(\mathbf{C})$

(ii) 任意の $X,Y \in \mathrm{Ob}(\mathbf{C})$ に対応する集合 $\mathrm{Hom}(X,Y)$。この集合 $\mathrm{Hom}(X,Y)$ の元を X から Y への**射 (しゃ；arrow)** と呼ぶ。

(iii) 任意の $X,Y,Z \in \mathrm{Ob}(\mathbf{C})$ に対応する**合成 (composition)** と呼ばれる写像

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

(iv) 任意の $X \in \mathrm{Ob}(\mathbf{C})$ に対応する $\mathrm{id}_X \in \mathrm{Hom}(X,X)$

(cat1) [結合法則] $(h\circ g)\circ f = h\circ(g\circ f)$ $\left({}^{\forall}f \in \mathrm{Hom}(W,X), {}^{\forall}g \in \mathrm{Hom}(X,Y), {}^{\forall}h \in \mathrm{Hom}(Y,Z)\right)$.

(cat2) [左単位性] $\mathrm{id}_Y \circ f = f$ $\left({}^{\forall}f \in \mathrm{Hom}(X,Y)\right)$.

(cat3) [右単位性] $f \circ \mathrm{id}_X = f$ $\left({}^{\forall}f \in \mathrm{Hom}(X,Y)\right)$.

「んー……これは関数と合成のあたりを抽象化したもの？」

「そう思ってくれて大丈夫だよ」

「圏ってなんだか難しそうだと思ってたけどそんなでもないような……？」

「圏というのは、何かに合成が考えられて、その合成に結合法則が成り立つというのが本質で、数学の議論のなかをよく探せば実はありふれたものだわ」

「ありふれたものがそんなに役立つのかな？」

「日常生活でありふれたものの代表は空気だけど、ありふれてるからといって役に立たないわけじゃないわ」

「まあ、確かに空気吸わなかったら実際死ぬよね」

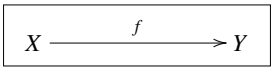
「圏という構造は数学のあらゆるところにあるけど、圏という言葉で照らしたときに意識の中に明確に浮かび上がってくる構造があるのよ」

「なるほど……？」

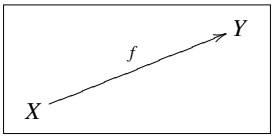
「まあ、でもこの段階ではまだそこまでわからなくていい——というより、そういうことをこれから説明するつもり」

図式

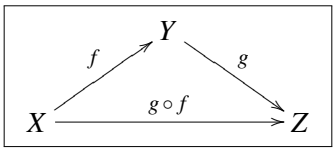
「具体的な圏を扱う前に、図式について触れておくことにしましょう。たとえば対象 X と Y に対して $f \in \mathrm{Hom}(X,Y)$ を——



——こんな風に矢印で描くと、射 f の起点と終点として X と Y を含めることが出来て、これは『対象 X から Y へ至る射 f』と書くより短いわ。それからね、この矢印は別に好きな方向に書いていいことにしましょう。たとえばこんな風に——」



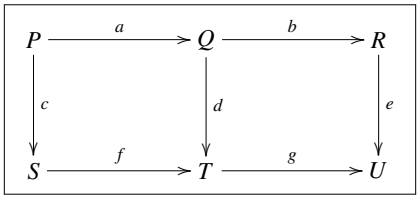
——こんな風に射を表す矢印が斜めでもいいことにすれば射の合成の様子はたとえば



みたいに書けるよね。射の矢印をこういう風に平面的に配置した図のことを**図式 (ずしき；diagram)** って言うんだ。要するに複数の射を並べて書くってだけなんだけど、こうやって自由に配置してみると見通しよく議論できる事が多いんだよ」

「なるほど、絵を書くのは楽しそうだね」

「楽しいというのも大事なポイントだけど、こうやって射を表す矢印の両側に対象を描くことで、合成可能かどうか一目瞭然なので、計算ミスを減らすためにも有効なのよ。例えば——」



「——こんな風に複数の対象が射で結ばれている状況を考えると、射 a と射 b を合成して射 $b \circ a$ を作ることができるし、同様にして $g \circ f$ や $g \circ d$ を作ることもできるけど『$d \circ f$』だとか『$b \circ d$』のよ

うな合成射は作れない事が、図式を目で追えばすぐわかるわ」
「なるほど、複数の射の合成可能性を素早く把握できる記法なんだね」

関手

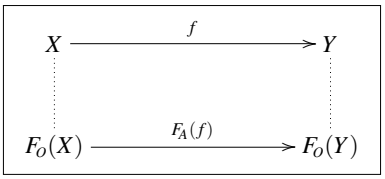
「次は**関手 (かんしゅ；functor)** の話よ。関手というのは、標語的に言えばある圏から別の圏への『射の合成性を保つような』対応のことよ。もっとちゃんと言えば――」

関手の定義 (object-part と arrow-part を分けて記述したもの)
圏 $\mathscr{C}, \mathscr{D}$ が与えられているとする。$\mathscr{C}$ から $\mathscr{D}$ への関手 $F = \langle F_O, F_A \rangle$ は以下の **(i)(ii)** のデータからなり、**(func1)(func2)** を満たすものを指す。

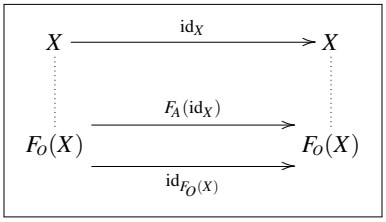
(i) 写像 $F_O\colon \mathrm{Ob}(\mathscr{C}) \longrightarrow \mathrm{Ob}(\mathscr{D})$
(ii) 任意の $X, Y \in \mathrm{Ob}(\mathscr{C})$ に対応する写像 $F_A\colon \mathrm{Hom}(X,Y) \to \mathrm{Hom}\big(F_O(X), F_O(Y)\big)$

(func1) $F_A(\mathrm{id}_X) = \mathrm{id}_{F_O(X)} \quad \Big({}^{\forall}X \in \mathrm{Ob}(\mathscr{C})\Big)$.
(func2) $F_A(g \circ f) = F_A(g) \circ F_A(f) \quad \Big({}^{\forall}X,Y,Z \in \mathrm{Ob}(\mathscr{C}),\ {}^{\forall}f \in \mathrm{Hom}(X,Y),\ {}^{\forall}g \in \mathrm{Hom}(Y,Z)\Big)$.

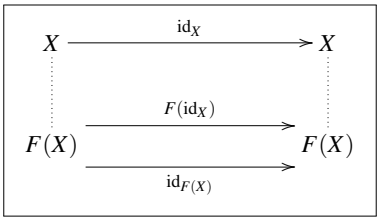
「――こうなるわ」
「えーと、関手の条件の **(ii)** は、圏 $\mathscr{C}$ の射が圏 $\mathscr{D}$ の射に対応するんだから、図式を書いてみるとこんな感じかな？」



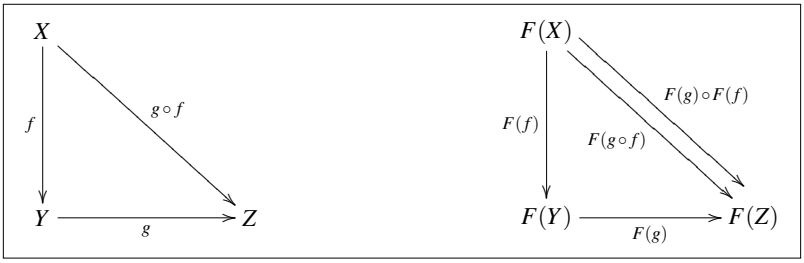
「そして関手の条件 **(func1)** に現れる射を図式として描いてみるとこんなふうになるね」



「圏 $\mathscr{D}$ の対象 $F_O(X)$ から自分自身への射が二つ、つまり $F_A(\mathrm{id}_X)$ と $\mathrm{id}_{F_O(X)}$ が表れているけど、これらが実は等しいというのが関手の条件 **(func1)** の主張だね。さて、対象を表す文字は大文字にして射を表す文字を小文字にするように決めておけば、実は関手 $F = \langle F_O, F_A \rangle$ の『ob-part』F_O と『arrow-part』F_A を区別せずに "F" と書いてしまっても大丈夫だね。今の図式だとこんな風になる――」



「混乱しそうなときだけ F_O や F_A を使うようにすればいいよね」
「関手の条件 **(func2)** に出てくる射を図式にするとこんなふうになるよ」



「右側に出てくる二つの平行な矢印で表された、射 $F(g)\circ F(f)$ と射 $F(g\circ f)$ が実は等しいというのが関手の条件 **(func2)** の主張ってわけだよね」
「結局のところ、関手っていうのは圏 $\mathscr{C}$ から圏 $\mathscr{D}$ へ、合成関係を崩さないように対象と射を移す仕組みだという事になるわ」

自己関手とその冪乗

「関手は圏 $\mathscr{C}$ から $\mathscr{D}$ へ対象と射を移すものなんだけど、特に $\mathscr{C}=\mathscr{D}$ のとき、つまり出発する圏と行き先の圏が同じ場合の関手を**自己関手 (じこかんしゅ；endofunctor)** と言うわ。定義としてはこれだけなんだけど、モナドは自己関手に関係した概念だということもあるのでもう少しだけ詳しく見て行きましょう」
「『モナドは自己関手に関係した概念』か、覚えておこう」
「さて、F を圏 $\mathscr{C}$ の上の自己関手とすると、対象 X に対して――」

$$X, F(X), F(F(X)), F(F(F(X))), F(F(F(F(X)))), \ldots$$

「――という系列があるけど、記法をコンパクトにして――」

$$X, F(X), F^2(X), F^3(X), F^4(X) \ldots$$

「――みたいにすると便利ね」
「つまりこういうことだね」

$$F^n(X) := \overbrace{F\circ\cdots\circ F}^{n\text{ 重の合成写像}}(X)$$

「念の為に付け加えれば、$F^1(X)=F(X)$ で、$F^0(X)=X$ だよね」
「ん？　あー、確かに」

「まったく同様にして、射 f に対しても $F^n(f)$ が考えられるわ」

「自己関手 F の『ob-part』F_O の n 重合成写像 F_O^n と『arrow-part』F_A の n 重合成写像 F_A^n からペア $\langle F_O^n, F_A^n \rangle$ をつくったとき、これってまた関手になってるの？」

「つまり $F^n := \langle F_O^n, F_A^n \rangle$ としたとき、この F^n は果たして関手の条件を満たすか、ということ？」

「うんうん」

「まず関手の条件 **(i)(ii)** はいいかしら」

「条件 **(i)** はまったく自明で、条件 **(ii)** も簡単だね。こうやって対象 X から Y への射 f をどんどん自己関手 F でずらしていけば――」

$$
\begin{array}{ccc}
X & \xrightarrow{\quad f \quad} & Y \\
F^1(X) & \xrightarrow{\quad F(f) \quad} & F^1(Y) \\
F^2(X) & \xrightarrow{\quad F^2(f) \quad} & F^2(Y) \\
F^3(X) & \xrightarrow{\quad F^3(f) \quad} & F^3(Y)
\end{array}
$$

「――こうなるし、ずっと続ければ一般の n についても言えるから結局――」

$$F_A^n \colon \mathrm{Hom}(X, Y) \longrightarrow \mathrm{Hom}\left(F^n(X), F^n(Y)\right).$$

となるのは明らかだよね」

「似たような図式を関手の条件 **(func1)** についても描いてみると面白いよ」

$$
\begin{array}{ccc}
X & \xrightarrow{\quad \mathrm{id}_X \quad} & X \\
F^1(X) & \xrightarrow{\quad F(\mathrm{id}_X) = \mathrm{id}_{F(X)} \quad} & F^1(X) \\
F^2(X) & \xrightarrow{\quad F^2(\mathrm{id}_X) = F\left(\mathrm{id}_{F(X)}\right) = \mathrm{id}_{F^2(X)} \quad} & F^2(X) \\
F^3(X) & \xrightarrow{\quad F^3(\mathrm{id}_X) = F^2\left(\mathrm{id}_{F(X)}\right) = F\left(\mathrm{id}_{F^2(X)}\right) = \mathrm{id}_{F^3(X)} \quad} & F^3(X)
\end{array}
$$

「なるほど、下の方に行ったときの等式の両端を取れば、一般には――」

$$F^n(\mathrm{id}_X) \;=\; \mathrm{id}_{F^n(X)}$$

「――であることがわかるし、このことから関手の条件 **(func1)** が成立することもわかるね。でも、面白いことってなんだろう？」

「いま魔理沙が関手の条件 **(func1)** を導くために捨てた中間の等式が、それはそれとして面白いのよ」

$$F^{n-m}\left(\mathrm{id}_{F^m(X)}\right) \;=\; \mathrm{id}_{F^n(X)} \qquad (m = 0, \ldots, n).$$

「面白い……かなぁ？」
「あ、あとで使うのよ」
「そういうことか（察し）」
「さて、いままでの議論でもなんとなく顔を出している関手 F^0 にはちゃんと**恒等関手 (こうとうかんしゅ；identity functor)** という名前があるわ。記号としてはここでは I を使いましょう。いままでの議論を理解していたら定義は書くまでもないけど、念の為に書いておくわ」

(id-functor1)　$I_O : X \longmapsto X.$
(id-functor2)　$I_A : f \longmapsto f.$

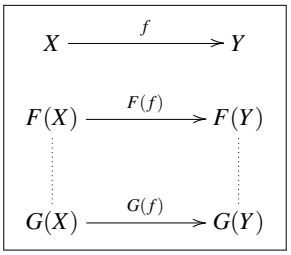
「多項式を考えるときも x に何を代入するかと関係なく $x^0 = 1$ とすると――」

$$c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n = \sum_{k=0}^{n} c_k x^k$$

「――みたいに、定数項だけを特別扱いせずに扱えたりしたな～」
「うん、そうね」
「それにしても、自己関手 F を一個買っただけで F^2 とか F^3 とか無限におまけがついてくるんだから自己関手ってコスパ最強だな……今なら自己関手をあと無限個お付けします！」
魔理沙が突然そう言ってクスクスと笑い出した。よくわからずに私も笑い出した。魔理沙が嬉しいと私も嬉しい。

自然変換と「ジェネリックな射」

「さて、関手というのはある圏 $\mathscr{C}$ から別の圏 $\mathscr{D}$ へ、合成関係を崩さないように対象と射を移す仕組みだったね。今から紹介する**自然変換**は、二つの関手を結んでくれる何か、と言えるし、後で説明するけど『ジェネリックな射』だとも言えるの。さて、圏 $\mathscr{C}$ から $\mathscr{D}$ への関手 F と G があったとして、圏 $\mathscr{C}$ の対象 X,Y に対してこんな図式を書いてみましょう」



「――そして、いま点線で書いたところを上から下に射で結んで4つの対象 $F(X),F(Y),G(X),G(Y)$ を結ぶ可換図式ができるようにしてくれるものを『関手 F から関手 G への**自然変換 (しぜんへんかん；natural transform)**』と呼ぶの。きちんとした定義も書いておくわ」

自然変換の定義

$\mathscr{C}$ と $\mathscr{D}$ を圏とし、F と G を圏 $\mathscr{C}$ から圏 $\mathscr{D}$ への関手とする。このとき、写像

$$\theta : \mathrm{Ob}(\mathscr{C}) \longrightarrow \bigcup_{X,Y \in \mathrm{Ob}(\mathscr{D})} \mathrm{Hom}_{\mathscr{D}}(X,Y)$$

が関手 F から関手 G への**自然変換 (natural transform)** であるとは、次の **(natx1)(natx2)** が成立す

ることである：

(natx1)　$\theta_X \in \mathrm{Hom}\big(F_O(X), G_O(X)\big) \quad \big({}^{\forall}X \in \mathrm{Ob}(\mathscr{C})\big).$

(natx2)　$\theta_Y \circ F_A(f) = G_A(f) \circ \theta_X \qquad \big({}^{\forall}X, Y \in \mathrm{Ob}(\mathscr{C}), {}^{\forall}f \in \mathrm{Hom}_{\mathscr{C}}(X,Y)\big).$

写像 θ が関手 F から関手 G への自然変換であることを記号で

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

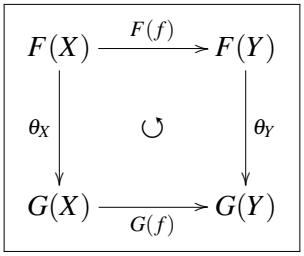
と表す。

「自然変換の条件 **(natx1)** は要するに任意の $X \in \mathrm{Ob}(\mathscr{C})$ について

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

ってことだね」

「うん。そして、条件 **(natx2)** を図式にするとこんな感じになるわ」



「パチュリー、このクルッとした矢印 “↻” は何？」

「これは、図式の中に射で結ばれた二通り以上の順路があるときにどっちの順路で射を合成しても同じものが得られますよ、ということを示す符丁よ。ちなみに、二つの対象を結ぶ合成の順路に沿って射を合成していくと必ず同じになるような図式を**可換図式 (かかんずしき；commutative diagram)** と言うわ」

「なるほど。それはさておき、自然変換の意味ってなんだろう？」

「条件 **(natx1)** から、自然変換は『沢山の射を束ねたもの』だと言えるわ。そういう意味では自然変換を『ジェネリックな射』と捉えることができるわ」

「ジェネリック、とか言うとプログラミングの話に近づいてる気がするね」

「ええ、あとちょっとの辛抱よ」

「条件 **(natx1)** が『ジェネリックな射』という条件だとしたら条件 **(natx2)** はなんだろう？」

「対象 X や Y が何らかの値の集合で、それらを結ぶ射も関数だとするとき、上の図式では $F(f)$ で得た値を θ_Y で “ずらして”$G(Y)$ の値に変換できるわね。しかも、その結果は “先に θ_X でずらしてから”$G(f)$ を適用したものと同じになるのよ」

「それを『自然』と呼ぶ気分はやっぱりピンと来ないけど、関手 F を使って計算した結果をスライドして関手 G で計算した結果に持っていく二つの方法が一致するという意味では、なんかすごく関連が強いことだけはわかる」

「どんな圏にもある身近な自然変換の例は id かな。実際」

(id:natx1)　$\mathrm{id}_X \in \mathrm{Hom}(X, Y) \quad \big({}^{\forall}X \in \mathrm{Ob}(\mathscr{C})\big).$

(id:natx2)　$\mathrm{id}_Y \circ f = f \circ \mathrm{id}_X \quad \big({}^{\forall}X, Y \in \mathrm{Ob}(\mathscr{C}), {}^{\forall}f \in \mathrm{Hom}(X,Y)\big).$

「んー……これって圏の定義の id に関係する公理を述べ直しただけのような？」

「そうだよ（便乗）。そうなんだけど、恒等関手と自然変換の定義を念頭に見直すと

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

であることが読み取れるでしょ」

「えーと、対象 X は恒等関手 I で移された $I_O(X)$ だとみなせて、射 f は恒等関手 I で移された $I_A(f)$ だと見なせるから……なるほど、確かに id は恒等関手 I から I 自身への自然変換になってる。これってあれだね、第一話から出てる冴えない仲間が実はすごい強い奴だったみたいな流れだよね、『id、おまえ……自然変換だったのかよッ！』的な？」

「ごめん、それよくわからない」

「それはともかく、ちょっと思ったんだけど『ジェネリックな射』のなかにはこんなのも考えられるよね――。」

$$\boxed{X \xrightarrow{\theta_X} G(X) \quad \left(\forall X \in \mathrm{Ob}(\mathscr{C})\right).}$$

「うん、考えられるね」

「だけど、いまみたいなやつだと自然変換の定義での関手 F に相当するものがないからこういう『ジェネリックな射』は自然変換じゃ表せないの？」

「この場合、自然変換 $\theta\colon I \xrightarrow{\bullet} G$ を考えればいいんだよ」

「なるほど、恒等関手 I にはそんな使い道があるのか！」

「さて、自己関手 F があったとして、さらにこんな自然変換があったとしましょう：」

$$\boxed{\tau\colon F^m \xrightarrow{\bullet} F^m.}$$

「m とか n は？」

「いまは決めないでおくわ」

「なるほど、m や n に依存しない一般論ってことね」

「この自然変換は『ジェネリックな射』なので自己関手 F で移せるわ。それを “$F \circ \tau$” と書くことにするわ。定義はこう」：

$$\boxed{\left(F \circ \tau\right)_X := F_A\left(\tau_X\right).}$$

「むむ？」

「自己関手 F を『さきにかぶせておく』こともできるわね。それを “$\tau \circ F$” と書くことにするわ。定義はこう」：

$$\boxed{\left(\tau \circ F\right)_X := \tau_{\mathrm{F(X)}}}$$

「むむむ……これはもしかして……」

「$\tau \circ F$ も $F \circ \tau$ も自然変換になるわ」：

$$\boxed{\begin{aligned} \tau \circ F &\colon F^{m+1} \xrightarrow{\bullet} F^{n+1}, \\ F \circ \tau &\colon F^{m+1} \xrightarrow{\bullet} F^{n+1}. \end{aligned}}$$

「やはり……！　これって証明難しい？」

「いいえ、簡単よ。だから証明は略すわ」

「なんかパチュリー、数学の先生みたい」
「ついでに触れておくと自然変換 $\mathrm{id}\colon I \xrightarrow{\bullet} I$ と自己関手 F からは二つの自然変換：

$$\mathrm{id} \circ F\colon F \xrightarrow{\bullet} F,$$
$$F \circ \mathrm{id}\colon F \xrightarrow{\bullet} F.$$

が作れるけど、$\mathrm{id} \circ F = F \circ \mathrm{id}$ であることも簡単にわかるわ」
「……これも簡単だから証明略かな？」
「その通り！ さて、いよいよお待ちかねのモナドの説明に入れるわ」

2.2.2 モナドと Kleisli 圏

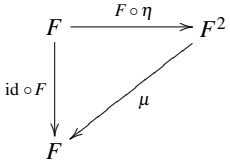
モナドの定義

モナドの定義

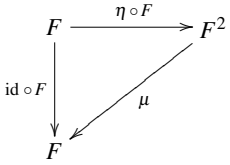
$\mathscr{C}$ を圏とする。$\mathscr{C}$ 上のモナドとは以下の **(i)(ii)(iii)** のデータからなり、**(mon1)(mon2)(mon3)** を満たすものを指す。

(i) 圏 $\mathscr{C}$ 上の自己関手 F,
(ii) 自然変換 $\eta\colon I \xrightarrow{\bullet} F$,
(iii) 自然変換 $\mu\colon F^2 \xrightarrow{\bullet} F$.

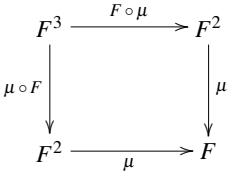
(mon1) 次のジェネリックな図式は可換である：



(mon2) 次のジェネリックな図式は可換である：

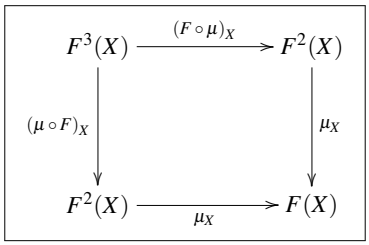


(mon3) 次のジェネリックな図式は可換である：



「ええと『ジェネリックな図式』ってなに？」
「自然変換を『ジェネリックな射』と呼んだ延長線上で勝手にそう言ってみたんだけど、普通の

図式にするためには任意の――つまりなんでもいいから一つの――対象 X を持ってきて補ってやればいいわ」:



「実際には、こうやって対象 X を補わなくても『自己関手を “対象” とし、自己関手から自己関手への自然変換を “射” とする圏』での図式になってるんだけど、今回はあまり深入りせずに済ませましょう」

「まあそういうのがあるってことだけ頭の片隅に入れとくよ」

「こうやって表示してみるとモナドの条件 **(mon1)(mon2)(mon3)** は、ある種の調和を表してることが見えてくるわ」

「見えないんだけど……」

半群の一般化としてのモナド

「魔理沙は**半群 (はんぐん；semigroup)** って知ってる？」

「んー、情報の代数の本に載ってたから多分知ってる」

「うろ覚えなのね。まあいいわ。上に挙げたモナドの条件 **(mon1)(mon2)(mon3)** は、半群の公理を図式にまとめたものと形が同じになるの。たとえば半群 M の二つの演算をこう書いてみる」:

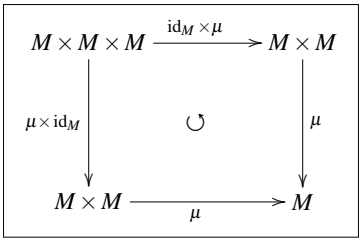
$$\eta : 1 \to M,$$
$$\mu : M \times M \to M.$$

「μ はアルファベットの m に相当するから multiplication から連想して乗法の二項演算の記号だな。ところで η の定義域の “1” ってなんだろう？」

「単元集合 $\{*\}$ だよ。つまりここでは半群 M の単位元をゼロ項演算とみなしてるわけ」

「ゼロ項演算？？」

「まあいいわ。代数の復習までしてると収拾がつかないから、あとで調べてちょうだい。さて、あえて可換図式を使って結合法則を表してみるとこんな風になるわ」:



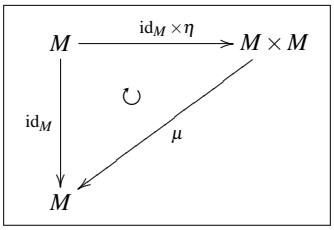
「――ただし、例えばここに出てきた $\mathrm{id}_M\times\mu$ は次のようなものよ」

$$\begin{aligned}\mathrm{id}_M\times\mu : M\times M\times M &\longrightarrow M\times M\\ \langle x,y,z\rangle &\longmapsto \langle x,\mu(y,z)\rangle\end{aligned}$$

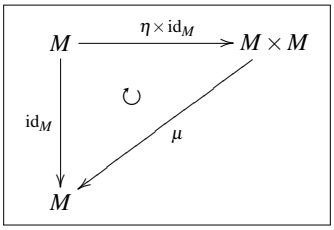
「$\mu\times\mathrm{id}_M$ の方もおんなじようにして書けるね」

$$\begin{aligned} \mu \times \mathrm{id}_M \colon M \times M \times M &\longrightarrow M \times M \\ \langle x, y, z \rangle &\longmapsto \langle \mu(x, y), z \rangle \end{aligned}$$

「同じようにして、半群の右単位条件はこんな可換図式になるわ」:



「そして、半群の左単位条件はこんな可換図式になるわ」:



「なるほど、書き換えの法則はちょっと良くわからないところが残ってるけど、確かにモナドの条件 **(mon1)(mon2)(mon3)** は半群の公理と似てるね」

「つまり、モナドというのは半群を一般化した概念だとも言えるの。だからこう[*1]言われてるわ」:

端的に言えば、X のモナドは X の自己関手からなる圏におけるモノイドに他ならず、自己関手の合成に置き換えられる積 $\times$ と恒等自己関手により定まる単位元を持つ.

「これってもしかして有名な『モナドは単なる自己関手の圏におけるモノイド対象だよ。何か問題でも？』というコピペの元ネタ？」

「たぶんね」

モナド条件のまとめ

「まあそれはともかく、モナドの条件をもっと噛み砕いて行くとこうなるわ」:

[*1] 『圏論の基礎』p.184 より。ただし、起こり得る読者の混乱を避けるため原文（訳文）の「函手」はすべて「関手」と書き換えました。

(M1) $F(f)\circ\eta_X = \eta_Y\circ f \quad \left({}^{\forall}X,Y\in \mathrm{Ob}(\mathscr{C}), {}^{\forall}f\in \mathrm{Hom}(X,Y)\right).$

(M2) $F(f)\circ\mu_X = \mu_Y\circ F^2(f) \quad \left({}^{\forall}X,Y\in \mathrm{Ob}(\mathscr{C}), {}^{\forall}f\in \mathrm{Hom}(X,Y)\right).$

(M3) $\mu_X\circ F(\eta_X) = \mathrm{id}_{F(X)} \quad \left({}^{\forall}X\in \mathrm{Ob}(\mathscr{C})\right).$

(M4) $\mu_X\circ \eta_{F(X)} = \mathrm{id}_{F(X)} \quad \left({}^{\forall}X\in \mathrm{Ob}(\mathscr{C})\right).$

(M5) $\mu_X\circ F(\mu_X) = \mu_X\circ \mu_{F(X)} \quad \left({}^{\forall}X\in \mathrm{Ob}(\mathscr{C})\right).$

「な、なるほど……？？」

「順番に、**(M1)** は自然変換 η の条件 **(natx2)** を書き下したもので、**(M2)** は自然変換 μ の条件 **(natx2)** を書き下したもので、**(M3)** はモナドの条件 **(mon1)** を書き下したもので、**(M4)** はモナドの条件 **(mon2)** を書き下したもので、**(M5)** はモナドの条件 **(mon3)** を書き下したものになってるわ」

「なるほど」

Kleisli 圏

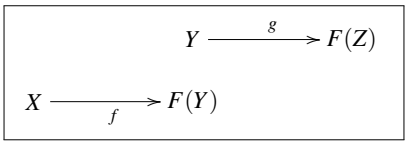
「モナドが半群の一般化だという話は、プログラミング言語への応用からすると脇道だと思う。モナドが役に立つ理由はなんといっても **Kleisli 圏**の存在だと思うわ」

「『くらいすり』は人の名前？」

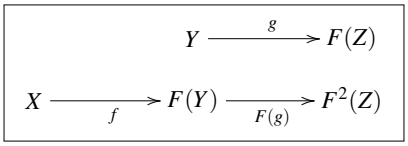
「そうね[*2]。簡単に言えば、Kleisli 圏は、圏 $\mathscr{C}$ 上の自己関手 F のなかのこんな形の射を “合成” するような圏だと言えるわ」:

$$X \xrightarrow{\ f\ } F(Y).$$

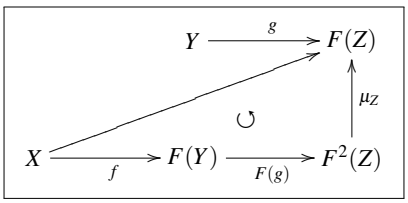
「それって、普通に無理だよね。だって、図式を書いてみると



となって、矢印の始点と終点が合わないから合成できないし。待って、なんか思いついた。F は関手だから今の図式の g を F でずりっと下ろしてやると ——



となって、あー……これを合成しても $X\longrightarrow F^2(Z)$ にしかならないけど関手 F がモナドならば自然変換 μ を使ってこんな風にできる ——



[*2] Heinrich Kleisli (October 19, 1930 April 5, 2011)

こうすれば、圏 $\mathscr{H}$ の射 $f\colon X \to F(Y)$ と射 $g\colon Y \to F(Z)$ の『合成』として

$$\boxed{\mu_Z \circ F(g) \circ f\colon X \to F(Z)}$$

が使えそうだね」

「いま魔理沙が書いた "$\mu_Z \circ F(g) \circ f$" が『Kleisli 射の合成』と呼ばれるものよ。つぎに問題になるのは、これが射としての資格があるかということね。実際には F がモナドならばうまく行って、それは『Kleisli 圏』と呼ばれてるわ。書き出してみましょう」:

Kleisli 圏

圏 $\mathscr{C}$ が与えられているとする。更に、$\mathscr{C}$ 上のモナド F が与えられているとする。次のデータ **(i)(ii)(iii)(iv)** について、**(kleisli1)(kleisli2)(kleisli3)** が成立する：

(i) $\mathrm{Ob}(\mathscr{C}^F) := \mathrm{Ob}(\mathscr{C})$.

(ii) 任意の $X, Y \in \mathrm{Ob}(\mathscr{C}^F)$ に対して

$$\overline{\mathrm{Hom}}(X,Y) := \mathrm{Hom}\,(X, F(Y))\,.$$

(iii) 任意の $X, Y, Z \in \mathrm{Ob}(\mathscr{C}^F)$ に対応する写像

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

(iv) 任意の $X \in \mathrm{Ob}(\mathscr{C}^F)$ に対応する

$$\eta_X \in \overline{\mathrm{Hom}}(X,X).$$

(kleisli1) $(h\ \bar{\circ}\ g)\ \bar{\circ}\ f = h\ \bar{\circ}\ (g\ \bar{\circ}\ f)\ \left(^{\forall} f \in \overline{\mathrm{Hom}}(W,X), {}^{\forall} g \in \overline{\mathrm{Hom}}(X,Y), {}^{\forall} h \in \overline{\mathrm{Hom}}(Y,Z)\right)$.

(kleisli2) $\eta_Y\ \bar{\circ}\ f = f\ \left(^{\forall} f \in \overline{\mathrm{Hom}}(X,Y)\right)$.

(kleisli3) $f\ \bar{\circ}\ \eta_X = f\ \left(^{\forall} f \in \overline{\mathrm{Hom}}(X,Y)\right)$.

記号を適宜置き換えることにより、上の構造は "$\bar{\circ}$" を合成写像とする圏だとみなせる。これを『圏 $\mathscr{C}$ 上のモナド F が作る Kleisli 圏』と呼び、$\mathscr{C}^F$ で表すことにする。

「パチュリー、条件 **(i)(ii)(iii)(iv)** は自己関手 F がモナドであることからすぐに出てくるね。でもって、**(kleisli1)(kleisli2)(kleisli3)** がこれから示さないといけないことだね」

「モナドの条件を書き下した **(M1)(M2)(M3)(M4)(M5)** を使えばできるよ」

「どれ、やってみるか」

Proof of **(kleisli1)**

$$\begin{aligned} h\ \bar{\circ}\ (g\ \bar{\circ}\ f) &= \mu_Z \circ F(h) \circ (g\ \bar{\circ}\ f) \\ &= \mu_Z \circ F(h) \circ (\mu_Y \circ F(g) \circ f) \\ &= \mu_Z \circ (F(h) \circ \mu_Y) \circ F(g) \circ f \end{aligned}$$

ここで条件 **(M2)** を射 $h\colon Y \to F(Z)$ に適用すると $F(h) \circ \mu_Y = \mu_{F(Z)} \circ F^2(h)$ となるので

$$\begin{aligned} &= \mu_Z \circ \left(\mu_{F(Z)} \circ F^2(h)\right) \circ F(g) \circ f \\ &= \left(\mu_Z \circ \mu_{F(Z)}\right) \circ F^2(h) \circ F(g) \circ f \end{aligned}$$

(M5) を適用すると

$$\begin{aligned}
&= (\mu_Z \circ F(\mu_Z)) \circ F^2(h) \circ F(g) \circ f \\
&= \mu_Z \circ \big(F(\mu_Z) \circ F^2(h) \circ F(g)\big) \circ f \\
&= \mu_Z \circ F\left(\mu_Z \circ F(h) \circ g\right) \circ f \\
&= (\mu_Z \circ F(h) \circ g) \ \bar{\circ}\ f = (h \ \bar{\circ}\ g) \ \bar{\circ}\ f
\end{aligned}$$

□

「これで Kleisli 射の合成の結合法則が示せたね」

「次に行ってみよう」

Proof of **(kleisli2)**

$$\begin{aligned}
\eta_Y \ \bar{\circ}\ f &= \mu_Y \circ F(\eta_Y) \circ f \\
&= (\mu_Y \circ F(\eta_Y)) \circ f
\end{aligned}$$

(M3) により

$$= \big(\mathrm{id}_{F(Y)}\big) \circ f = f.$$

□

「これで η が Kleisli 射の合成について左単位元を与えてくれる事が示せたね」

「最後に、η が Kleisli 射の合成について右単位元を与えてくれる事を示そう」

Proof of **(kleisli3)**

$$\begin{aligned}
f \ \bar{\circ}\ \eta_X &= \mu_Y \circ F(f) \circ \eta_X \\
&= \mu_Y \circ (F(f) \circ \eta_X)
\end{aligned}$$

(M1) において $Y := F(Y)$ と置き換えたものを適用すれば

$$\begin{aligned}
&= \mu_Y \circ \big(\eta_{F(Y)} \circ f\big) \\
&= \big(\mu_Y \circ \eta_{F(Y)}\big) \circ f
\end{aligned}$$

(M4) により

$$= \big(\mathrm{id}_{F(Y)}\big) \circ f = f$$

□

「さてこれで圏 $\mathscr{C}$ 上のモナド F が作る Kleisli 圏 $\mathscr{C}^F$ が確固たる存在だと確認できたね、魔理沙」

「うん。最初は圏っていうのはつまらない抽象化のように思えたけど、なんか案外奥が深い……？」

「その辺は見方によるかもね。さて、せっかくだから話を整理するために Kleisli 圏 $\mathscr{C}^F$ と元の圏 $\mathscr{C}$ の言葉の対比表を作ってみましょう」:

2.3 モナドを使って状態を持つ関数を扱う方法

2.3.1 状態を持つ関数

「以上の説明で、Haskell で状態を扱う方法 —— 参照透明性の枠の中で —— をする準備が整ったわ」

「やっとか！」

「型 X から型 Y への、型 S の『内部状態』変数を持つ関数は、たとえば C++ ではこんなふうに書けるわ」:

圏 $\mathscr{C}$ 上のモナド F が作る Kleisli 圏 $\mathscr{C}^F$	圏 $\mathscr{C}$ における実体
対象 X	$X \in \mathrm{Ob}(\mathscr{C})$
射 $f\colon X \to Y$	$f\colon X \to F(Y)$
対象 X に対応する恒等射 id_X	$\eta_X\colon X \to F(X)$
$f\colon X \to Y$ と $g\colon Y \to Z$ の合成射 $g \circ f$	射 $\mu_Z \circ F_A(g) \circ f$
対象 X と Y が定める hom 集合 $\mathrm{Hom}(X, Y)$	$\mathrm{Hom}\big(X, F(Y)\big)$

表 2.1: 圏 $\mathscr{C}$ 上のモナド F が作る Kleisli 圏 $\mathscr{C}^F$ の構成要素とそれらの圏 $\mathscr{C}$ における実体

```
Y  stateful_func(X x){
    static S state = s_0; //static 変数 state は関数の内部状態を表現している。
    S next_s = new_state(x, state);
    Y ret = return_val(x, state); //戻り値を計算する
    state = next_s; //状態の更新をする。
    return ret;
}
```

「静的変数の初期化のところは、プログラムが実行されるまえに一度だけ呼ばれるんだっけ」

「そうだね」

「だから、プログラムが起動した直後の、まだ関数 `stateful_func` が一度も呼ばれてないときには関数内部の静的変数 `state` の値は `s_0` になってるんだな」

「そして、`return` 文の直前で変数 `state` を `next_s` で更新してるから、関数 `stateful_func` が呼び出されるたびに『内部状態』を表す変数 `state` が更新されるのよ。そして、こういうことができるのは C++ では当たり前に破壊的代入ができるからね」

「そうだな」

「複数の状態変数を持つ関数であっても、要するに状態変数全部を構造体にまとめちゃうとかすれば、今の形式で扱えるわ」

「なるほど、確かにそうだ」

「いま挙げた『内部状態』を持つ関数は、本質的には『与えられた状態と型 X の引数』というペアから『次の状態と型 Y の戻り値』を計算していると言えるわ」

「んー……なるほど、『与えられた状態と型 X の引数』から次の状態を計算してる関数が `new_state` で、『与えられた状態と型 X の引数』から戻り値を計算してる関数が `return_val` というわけか」

「これらの関数を部品として使って、こんな感じの関数を考えられるわ――」

$$f\colon X \times S \longrightarrow Y \times S,$$

ただし

$$f(\langle x, s\rangle) = \langle \mathrm{return\_val}(x,s),\, \mathrm{next\_state}(x,s)\rangle\,.$$

「これはいまパチュリーが言った『与えられた状態と型 X の引数』というペアから『次の状態と型 Y の戻り値』を計算するという話をそのまま関数にしてみたんだね」

「うん。そしてもしここに出てきた関数 return_val と next_state が両方とも参照透明な関数ならば、この形をそっくりそのまま Haskell に持ってくることができるよね」:

```
f :: (X , S) -> (Y , S)
f (x, s) = ( return_val x s, next_state x s )
```

「まあそうだね」

「そんなわけで『Haskell の参照透明性の枠の中で状態を扱う』方法の第一の答えがこれね」

「なんかすごく……がっかり感が半端ない。えーと……モナドはどこ行ったの？」

「第二の答えはモナドを使うんだけど、第一の答えのがっかりなところを言葉にしてみると何だろう？」

「うーん、C++ なんかで書いた『内部状態を持つ関数』に比べて、関数の形が素直じゃないような」

「そんなわけで、さっきの f にカリー化を施します」:

$$f' : X \longrightarrow \mathrm{Hom}(S, Y \times S)$$

Haskell に持ってくるとこうなるね：

```
f' :: X -> (S -> (Y , S))
f' x = \s -> ( return_val x s, next_state x s )
```

「ねえパチュリー、いま大変な事に気づいたんだけど」

「何かしら……？」

「ええと、なんか Haskell と圏の関係の説明をおもいっきり省かれた気がする……！」

「それは諦めて頂戴」

「ぇー」

「シメキリとか色々あるのよ。圏についてはそれなりにじっくり説明したし、Haskell をちょっと読み書きできればなんとかなると思うわ」

「なんとかなるのかな……」

「不安みたいだから、Haskell の圏 $\mathscr{H}$ について、説明なしでまとめの表だけ書いておこうか」

「はるかに良いです」

圏 $\mathscr{H}$ の言葉	Haskell の言葉
対象 X	型 `X`
対象 X と Y が定める hom 集合 $\mathrm{Hom}(X, Y)$	型 `(X -> Y)`
対象 X と Y からなる積対象 $X \times Y$	型 `(X,Y)`
射 $f : X \to Y$	関数 `f :: X -> Y`
射 $f : X \to \mathrm{Hom}(Y, Z)$	関数 `f :: X -> (Y -> Z)`
対象 X に対応する恒等射 id_X	関数 `id :: X -> X`
$f : X \to Y$ と $g : Y \to Z$ の合成射 $g \circ f$	合成関数 `g.f :: X -> Z`

表 2.2: Haskell の圏 $\mathscr{H}$ と Haskell の言葉の対応

2.3.2 状態モナド

「結論から言えば今の議論で出てきたタイプの関数：

$$f' : X \longrightarrow \mathrm{Hom}(S, Y \times S)$$

は Kleisli 射なのよ」

「うわあああああああガタッ、(椅子から落ちる音)」

「素直に驚いてくれたことはちょっと嬉しいな。さて、ちゃんと説明すると状態を表す型を S をパラメータとする圏 $\mathscr{H}$ 上の自己関手 $\mathrm{State}^S = \langle \mathrm{State}^S_O, \mathrm{State}^S_A \rangle$ を

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

ただし

$$\mathrm{State}^S_A(f)(\varphi) := \lambda s \mapsto \langle f(\varphi_1(s)), \varphi_2(s) \rangle .$$

ここで、$\varphi(s) = \langle \varphi_1(s), \varphi_2(s) \rangle$ とした。

——こんなふうに定義できるんだ」

「ぱっと見たとき、射を移す写像 State^S_A が複雑に見えるけどど $f \colon X \to Y$ を使う縛りで考えれば、まー自然にこの形にたどり着くね」

「そんな風に、関手 $F = \langle F_O, F_A \rangle$ のうちの『arrow-part』F_A は『ob-part』F_O だけから自然に決まる事も多いので、数学者向けの議論では F_O だけ与えて話を進めることも多いわ」

「なにそれ怖い w」

「さて、すでに軽く触れたけど関手 State^S はさらにモナドになってるんだ」

「二つの自然変換 η と μ が必要だな」

「自然変換 η はこんな風に定義されるわ」:

$$\begin{aligned} &\eta_X \colon X \longrightarrow \mathrm{State}^S_O(X) = \mathrm{Hom}(S, X \times S) \\ &\eta_X(x_0) = \lambda s \mapsto \langle x_0, s \rangle \end{aligned}$$

「そして自然変換 μ はこんな形になるわ」:

$$\begin{aligned} &\mu_X \colon \mathrm{State}^S_O\left(\mathrm{State}^S_O(X)\right) \longrightarrow \mathrm{State}^S_O(X) \\ &\mu_X(\varphi) = \lambda s \mapsto \varphi_0(s_0), \end{aligned}$$

ただし

$$\begin{aligned} &\varphi_0 := \left(\pi_1 \circ \varphi\right)(s), \\ &s_0 := \left(\pi_2 \circ \varphi\right)(s). \end{aligned}$$

「こうして State^S がモナドであり、状態を扱う事ができるので『状態モナド』と呼ばれるのよ」

「ごめん、μ の定義に出てくる π_1 とか π_2 はなんだろう？」

「Haskell では `fst`、`snd` とそれぞれ呼ばれてるものと同じに思っていいよ」

「なるほど。ところで、こいつらがちゃんとモナドになることは……」

「ちょっと面倒だけど難しくないから、読者への演習問題とします」

「読者？ ワッザ！？」

「気にしないで。いずれにせよ、関手の性質 **(func1)(func2)** とモナドの性質 **(M1)(M2)(M3)(M4)(M5)** を示せば終わりでしょ」

「宿題かぁ」

2.3.3 状態モナドの Kleisli 射としての『状態付き関数』

「さて、色々端折ってしまったけど、最後に Haskell で状態付きの関数をモナドの Kleisli 圏を通

して扱ったことを、プログラミングの立場から考えてみましょう。話をひっくり返すようだけど、状態モナドを使ったからといって『内部状態』を真に実現したわけじゃないわ」

「うん。本当に書き換え可能な内部状態を実現したら Haskell の参照透明性をぶっ壊すことになるもんね。むしろ、状態モナドは、『更新可能な内部状態を参照透明性の枠の中でシミュレートしてる』ぐらいの捉え方がいいかなと思った」

「それで合ってると思うわ。そして、それにくわえて、Kleisli 射としての合成を通してあたかも普通の関数のように合成できるのも、ある程度使いやすさに貢献してると思うわ」

「今まではどっちかというと圏論の記号で議論してきたけど、Kleisli 射の合成の “∘” に相当するものって Haskell にもちゃんとあるの？」

「もちろんあるわ――」:

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
```

「なるほどー」

「状態モナドの Kleisli 射の形で書くと『状態そのものから分離した』関数の挙動を扱えるので、私は自分の属性魔法の合成魔法を最適化するために状態モナドを利用してシミュレーションをしてるんだ」

「なるほど、面白そうだなぁ」

その後、咲夜さんが差し入れてくれたお茶を飲みながら魔理沙と私は話し込んだ。

2.4 エピローグ

数学やプログラミング言語の時間が終わり、そしてその後のお茶の時間も終わった。楽しいひとときはこれで終わり。

流石に私も疲れた。魔理沙が帰るのを見送るとき、ふとたまらなく寂しい気持ちになった。魔理沙が帰っちゃったらまた紅魔館に私は取り残されてしまう。レミリアも咲夜さんもみんないい人たちなんだけど――。

そんな私の想いをが露骨に顔に出てしまっていたのだろうか、魔理沙が聞いてきた。

「なあ、パチュリーはずっと図書館で暮らしていて平気なの？ つまらなくなったりしない？」

いつもなら――いつもの私なら「本を読むのが一番楽しいの」とでも答えたかもしれない。

でも、私の心の状態を一歩前に、少しだけ変えて見よう――。

「百年ちかく本ばかり読んでたので、最近はちょっと飽きてきたわ。ねえ魔理沙、帰っちゃう前に、ほんの五分だけでいいから私もあなたの箒の後ろに乗せてくれないかしら？ ――紅魔館を上から見てみたいの」

参考文献

本記事の執筆にあたり、上海アリス幻樂団様の東方 Project シリーズからキャラクターや設定を拝借いたしました。素晴らしい作品を発表されている原作者様に敬意を表明いたします。勝手なキャラクターや設定の拝借および改変については何卒ご寛恕下さいますよう願い申し上げます。また、執筆にあたって以下の文献を参考にいたしました。

2.4.1 参考にした教科書および専門書

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

は多大な影響を受けています。

[2] 種村季弘「薔薇十字の魔法（新訂版）」 青土社 (1986)
魔術についてのそれっぽい文章を作るために参考にさせて頂きました。

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

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

[5] S. Mac Lane, “Categories for the Working Mathematician”, Springer-Verlag (1971)(2nd ed : 1998)
よく読まれている圏論の教科書です。何の説明もなく「CWM」と言う略称で呼ばれることもあります。すでに何らかの分野における数学の専門知識を持っている読者を想定読者としているため難しいところも多いのですが、圏論について真剣な興味があるなら持っていて損はありません。本記事における Kleisli 圏の扱いは、この本における「モナドの Kleisli 圏」の項目を参考にしています。ただし、この本での Kleisli 圏の記述は非常にあっさりとしており、証明も概略が与えられているにとどまります。

[6] S. Mac Lane 「圏論の基礎」 三好 博之・高木 理 共訳, シュプリンガー・ジャパン (2005), 丸善 CWM 第二版 (1998) の日本語訳です。

[7] —— 「Haskell/圏論」
`http://ja.wikibooks.org/wiki/Haskell/圏論`
Haskell と圏論の関係について基本的な事が書かれています。モナドについても解説されています。本記事における条件 **(M1)**-**(M5)** は、このページの unit と join によるモナド則をお借りしたものです。

[8] —— (Stack Overflow)
`http://stackoverflow.com/questions/3870088/`
有名なコピペ『モナドは単なる自己関手の圏におけるモノイド対象だよ。何か問題でも？』が CWM に由来するという説はここに載っていました。

[9] Eugenio Moggi “Computational lambda-calculus and monads” (1988)
計算の圏論的意味論がモナドの理論に基いて述べられています。本記事のメインテーマである状態モナドもこの論文で取り上げられています。

第3章

【新編】加速しなイカ？【叛逆の物語】

— @nushio

希望を願い、呪いを受け止め戦い続ける者たちがいる。それが科学少女。
就職を拒んだ代償として戦いの運命を重ねた魂。その末路は、消滅による救済。
この世界から消え去ることで、絶望の因果から解脱する。
いつか訪れる終末の日、“技術的特異点”を待ちながら、私たちは戦い続ける。
剽窃と捏造ばかりを繰り返す、この救いようのない世界で。
あの、懐かしい笑顔と、再びめぐり合うことを夢見て。

3.1 プロローグ

見滝原中学校　教室

「…女子のみなさんは、くれぐれもジャバじゃなきゃアイティーじゃない、とか抜かす男とは交際しないように！」

「そして男子の皆さんは、絶対に特定の言語にケチをつけるような大人にならないこと！」

生徒達の苦笑。早乙女先生の失恋話はもはや生徒たちにとって慣れっこらしい。私にとっても——そう、数えるのを諦めるほどに。黒板には意味をなさないコードの断片が手描きフォントで貼り付けられている。シャフトちゃんと仕事しろ。

「コホン♡それでは、今日の授業は先週の宿題から。好きなプログラミング言語を選んでその歴史についての調べ学習でしたね。暁美ほむらさん」

「はい」

ここで当てられるのは中沢くんではなく私。それは運命石（シュタインズゲート）の選択——ではなく、私が意図的に引き当てた世界線なのだ。英語の授業がおかしな言語の授業へ置き換わっていることもそう。物語を操る**モナド**の力に辿り着くために。物語の外に逝ってしまった彼女にもう一度めぐり逢うために。私は落ち着いたペースで教室前方へ歩を進める。ノート PC などは携えず手ぶら。地域モデル校先行導入の教育支援情報クラウドシステムにより、電子黒板が自動的に私のデスクトップに切り替わる。早乙女先生と場所を替わった私はプレゼンテーションを始めた。

3.2 夢のようなライブラリがあった、ような…

「かつて、CPU の周波数が時間の指数関数的に向上を続け、ただ互換性のある CPU を買い換えつづけるだけで、あらゆるソフトウェアの性能が向上していったフリーランチとよばれる時代がありました。しかし、西暦 2005 年ごろには CPU の単一コアの周波数の上昇傾向が終わり、以降は誰もがいつまで待っても到着しないランチを狩りに行くために並列プログラミングという重労働を強いられる時代となりました。CPU も Bulldozer や Haswell など、どんどんベクトル化／マルチコア化

が進み、GPU なら 2010 年代には既に何万というスレッドを並列に動作させる環境が数万円で入手可能になりました。さらに将来いかに異様なハードウェアが現れても、プログラムを書いていかざるを得ない。ならばプログラムを操る程度の能力を持った言語が是非とも必要である、と唱え、関数型言語の勢力がさかんに高性能計算に特化した言語 (DSL) を作りはじめました。しかし、彼らばかりではありませんでした。いまこそ C++ 言語の力を見せつけてやろうと、闇の言語の軍勢もまた立ち上がったのです」

「Halide というライブラリ `http://halide-lang.org` は、C++ 言語の高速配列演算ライブラリであって、GPU をはじめとするさまざまな計算モデルの上でのステンシル計算ができます。ステンシル計算とは、配列変数を更新していくタイプのアルゴリズムで、配列の各要素を、それぞれ近傍の要素の値のみに基づき、同じルールで更新するものをいいます。具体的なアプリケーションでは、Halide が念頭に置いている画像処理、また配列ベースの流体計算、MHD、等々です。

レポジトリ `https://github.com/halide/Halide` から最新のコードはチェックアウトできます。チェックアウトしたら、そのフォルダで

```
> make
```

するだけです。インキュベー、じゃなかったインストールする場合には適当にインクルードパス/ライブラリパスの通ったところにファイルを置きます。C++ のライブラリはいつだって、初見は意味不明不明なものです。それでもさっぱり読めないテンプレート型エラーに泣きながら動かしてみたり、Doxygen のハイパーリンクでいつも同じ所に飛ばされたりしているうちに、おぼろげながらライブラリの構造が見えてくるものです。Doxygen はとても便利で生産性の向上に貢献することがわかります。」

■Halide とは？ 「注意点として。まず執筆時点では Halide は LLVM 3.3 以降を必要とします。OS のディストリビューションによっては、パッケージマネジャーが管理している LLVM のバージョンが低いために LLVM の野良ビルドが必要な場合があります。おなじく執筆時点の情報ですが、github にある Halide は公式サイトのバージョンと比べてだいぶ進んでおり、境界条件の扱いをはじめとする様々な機能が盛んに開発中であることに注意が必要です。この解説は Halide の開発者らによる論文 [3, 4] を基にしているので、そちらも参照してください」

私はすらすらと説明していく。それも当然、私はこの授業を何度も「予習」しているのだもの。

「さて、Halide は、ただのコードジェネレータではなく、あくまでも C++ プログラマにプログラム自動生成がもたらす速度を提供したいという発想で作られています。C++ にビルトインのコード生成メカニズムであるテンプレートメタプログラミングも使用していません。純粋な C++ のオブジェクト指向と演算子オーバーロードで C++ 内に埋め込まれた独自言語を実現しているのが大きな特徴です。テンプレートプログラミングにとらわれず、C++ プログラムの任意の箇所から GPU 計算を呼び出せます。実行中計算結果が要求されると、その場で LLVM を利用して CPU もしくは GPU 向けプログラムを生成し、コンパイルし、自身にリンクすることで結果を得るという、素晴らしい技術を用いています。同様な目的で開発された Paraiso [2] とは、この表のような機能差があります」

	Paraiso	Halide
基本言語	Haskell	C++
コード生成対象	x86, CUDA	x86/SSE, ARM, Native Client, OpenCL, CUDA, ...
扱える次元	n 次元	n 次元
最適化の種類	ループ融合、同期	並列度、局所性、計算節約 の間のトレードオフ
自動チューニング機能	あり	なし (ただし論文中で言及あり)

3.3 見かけはとっても簡潔だなって

■Halide の文法 「プログラミング言語 Halide は、 普及を意識し、広く使われている言語である C++ のライブラリとして実装されています。したがって Halide でプログラムするということは C++ のプログラムを書くことになります。基本的な型として、*n* 次元配列を表す `Func` と、配列添字変数を表す `Var` があり、これらを組み合わせてステンシル計算を記述します。Halide のサンプルプログラムを示します」

私はプレゼンテーションを表示していた画面を分割してコンソールを出し、サンプルコードを `cat` する。

```
Func input = initial_condition.realize(NX,NY);
Var i,j;
Func cell2(i,j) = (input(i-1,j)+input(i,j)+input(i+1,j))/3;
Func output(i,j) = (input(i,j-1)+input(i,j)+input(i,j+1))/3;
```

「このような構文を実現するため、Halide では C++ の演算子オーバーロードを多用しています。例えば、複数引数関数のように見えるものは括弧演算子 `operator()` をオーバーロードすることで実現しています」

■Halide のプログラムの例 「Halide ではステンシル計算をとても直感的な形で記述することができます。たとえば、以下のような拡散方程式の離散化解法は」

$$C_{\text{in}} = \text{initial condition} \tag{3.1}$$

$$C_2[i,j] = \frac{1}{3}\left(C_{\text{in}}[i-1,j] + C_{\text{in}}[i,j] + C_{\text{in}}[i+1,j]\right) \tag{3.2}$$

$$C_{\text{out}}[i,j] = \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right) \tag{3.3}$$

「ほぼそのまま以下のような Halide 記述に翻訳できます。ループを書くことなく、2 次元配列へのアクセスや代入を表現できていることがわかります」

```
Var i,j;
Func input = initial_condition.realize(NX,NY);
Func cell2(i,j) = (input(i-1,j)+input(i,j)+input(i+1,j))/3;
Func output(i,j) = (input(i,j-1)+input(i,j)+input(i,j+1))/3;
```

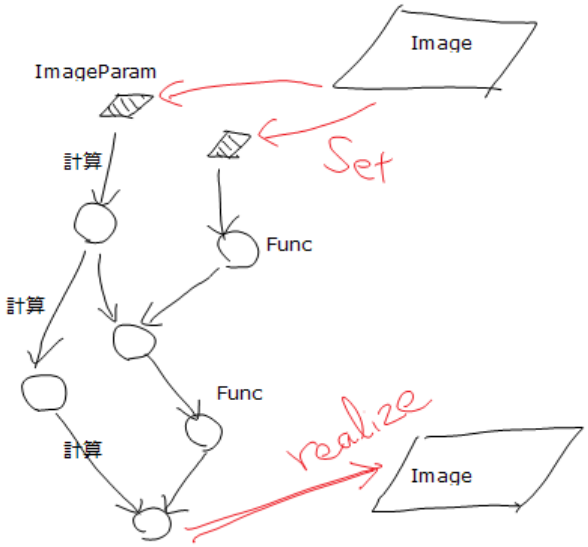
■Halide のコード生成と実行 「Halide で、配列変数の表現としてもっとも頻繁に登場する `Func` 型は、『その配列を計算するための手続き（データフローグラフ)』を保持しているにすぎません。これに対し `Image` 型は実際にメモリ上に確保された多次元配列です。

`realize()` 関数を呼び出して `Func` 型を `Image` 型に変換したとき、コード生成・最適化・実際の計算が行われるという仕様です。(プログラムに変更がなければ、いったん生成されたコードは使いまわされます)」

```
inPar.set(input);
```

```
output=cell3.realize(NX,NY);
```

■Halide のデータフローグラフのイメージ



- Image:実際にメモリ上に確保された配列
- Func:配列を操作するプログラムを構成するデータフローグラフの要素
- ImageParam:Image へのポインタ、データフローグラフの入力点

「このように、C++ 言語のデータとして構文木を持つ Halide のアプローチは、テンプレートメタプログラミングなど C++ の機能を利用したコード生成とくらべ、コード生成部を Halide 開発者がすべて実装しなければならないという欠点があります。しかし、Halide 開発者たちは、コード生成部に LLVM フレームワークを利用することで実装コストを大きく抑える一方、テンプレートメタプログラミングよりもはるかに自由度の高いコード生成を手に入れることを選択したのです」

■Halide におけるステンシル計算の実装例 「Halide で実装されたステンシル計算の例を示します。Halide では、いったん生成した Func 計算にたいしメソッド呼び出しにより具体的な順序や並列化度を指定することで、計算アルゴリズムを変換することができます。これらのメソッドはスケジュール (schedule) と呼ばれています。これらスケジュールには色々な種類があり、またいくつも組み合わせて施すことができます。スケジュールの多様な組み合わせ可能性が、Halide の誇る多様な自由度のコード生成を実現しています」

```
Var x,y;              // 配列添字を表す変数
Image input, output; // 計算結果を格納する配列変数
ImageParam inPar;     // Halide プログラムの入力
Func cell2(x,y) = a * inPar(x,y) + b * inPar(x+1,y); // Halide プログラム
Func cell3(x,y) = c * cell2(x,y) + d * cell2(x,y+1); // Halide プログラム

// スケジュールを指定
cell3.split(y, yo, yi, 16).parallel(yo).vectorize(x,4);
cell2.store_at(cell3,yo).compute_at(cell3,yi).vectorize(x,4);

for (int t=0; t<=MAX_T; ++t) {
  inPar.set(input);             // プログラムに入力を設定
  output=cell3.realize(NX,NY);  // プログラムを実行
```

```
    std::swap(input, output);      // 出力を次のタイムステップの入力へ交換
  }
```

■Halide のコード生成 「生成されたコードはその場で主プログラムにリンクされて実行される (Just in Time compile, JIT) のほか、C++ 言語ソースやアセンブリ、LLVM バイトコードをファイルに生成させ、のちほど利用する (Ahead of Time compile, AOT) こともできます」

```
compile_to_assembly
compile_to_bitcode
compile_to_c
compile_to_file
compile_to_function_pointers
compile_to_header
compile_to_lowered_stmt
compile_to_native
compile_to_object
compile_to_src
```

■Halide/LLVM 経由での融合加乗算の命令 「例えば、Halide から.bc ファイルを生成した上で、Bulldozer 向けアセンブリを生成させることで、fma(融合加乗算) 命令を含むアセンブリを生成できます」

```
$ clang -O1 -march=bdver1 -ffp-contract=fast blur.bc -S

$ less blur.s
...
vfmaddss %xmm3, (%r11,%rdi,4), %xmm1, %xmm3
vfmaddss %xmm3, (%r11,%r10,4), %xmm1, %xmm3
vfmaddss %xmm4, (%r11,%rax,4), %xmm1, %xmm4
vfmaddss %xmm4, (%r11,%r14,4), %xmm1, %xmm4
vmulss %xmm0, %xmm4, %xmm4
...
```

「fma は 1 命令で $f(x,y,z) = x \times y + z$ の形式の乗算と加算を同時に行う命令です。GPU はむろんのこと、執筆時点では Bulldozer,Haswell をはじめ最新式の CPU にも fma 命令が備わっています。これらハードウェアにおいて、fma 命令は、それなくしてはどう足掻いてもピーク性能の半分しか出すことができない、とても重要な命令です。Halide が fma 命令を出力できることは、現在及び将来のハードウェアにおける Halide プログラムの性能に寄与すると考えられます」

3.4 並列度も、演算重複も、メモリ負荷も、大事だよ

■Halide の最適化空間 「Halide の設計では、ステンシル計算の速度を向上させるための要因として、並列度を上げること、同一値の再演算を避けること、無駄なメモリアクセスを省くことの三種類を考えています。いっぽう、Halide は、ステンシル計算に対する様々な変換をサポートしていますが、これらは『計算の粒度』と『メモリアクセスの粒度』の二軸を操作することに分類されます。大雑把にいって、計算の粒度とはある値が計算されてから次に使われるまでの時間間隔、メモリアクセスの粒度とはあるアドレスが次にアクセスされるまでの時間間隔のことです。粒度が細かい (粗い) とは、時間間隔は短い (長い) ことを意味します。

Halide では、ステンシル計算の実行速度最適化問題が難しい本質的理由は、二軸を操作しながら、三種類の要因を最適化する問題であることにある、としています。三種類の要因はどれも実行速度と正の相関があるはずですが、操作できる軸にたいし、要因の数のほうが多いため、三種類の要因すべてを向上させる自明なアプローチが存在しない、拘束条件つき最適化問題になっています。具体的には以下のようなトレードオフがあります」

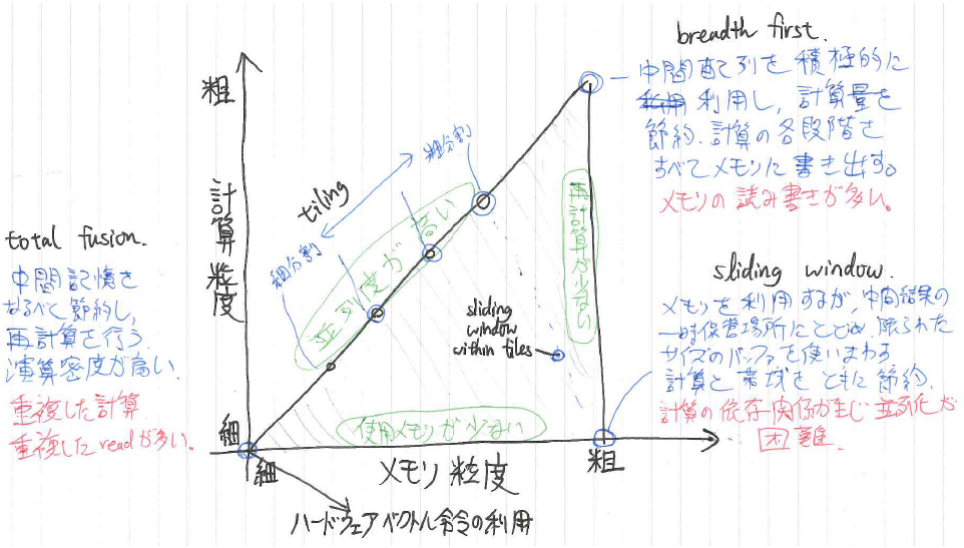


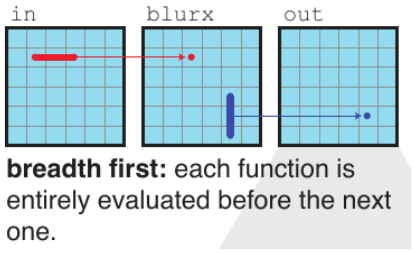
図 3.1: Halide の最適化問題空間の捉えかた

メモリ粒度	計算粒度	並列度向上	演算重複回避	メモリ負荷軽減
粗い	粗い	○	○	×
細かい	細かい	○	×	○
粗い	細かい	×	○	○
細かい	粗い	このような組み合わせは存在しない		

「といっても、さっぱりわかりませんね。彼らが言う計算とメモリの粒度とは何か？ それが三要因のトレードオフにどう関係するのか？ 以下、拡散方程式を例に見ていきましょう」

■Halide の最適化空間:粗メモリ、粗計算

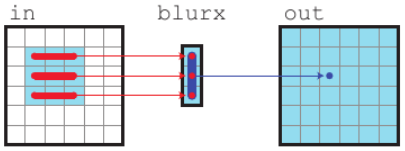
$$C_{\text{in}} = \text{initial condition}$$
$$C_{\text{blurx}}[i,j] = \frac{1}{3}\left(C_{\text{in}}[i-1,j] + C_{\text{in}}[i,j] + C_{\text{in}}[i+1,j]\right)$$
$$C_{\text{out}}[i,j] = \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right)$$



[4]

「$C_{\text{blurx}}[i,j]$ を全て計算し、メモリに置いてから、それを利用して $C_{\text{out}}[i,j]$ を計算するスケジュールです。これは大きなメモリを確保し、中間結果をすべて書き出しながら計算を行うため、一度計算した値を再計算することはありません。また、大きなメモリへの書き込みは自由に並列化できます。しかし、**メモリアクセス頻度が高く、必要メモリ容量も多く**なってしまう欠点があります」

■Halide の最適化空間:細メモリ、細計算

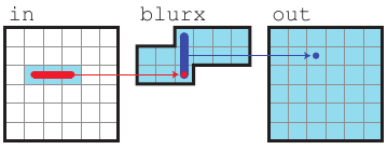


total fusion: values are computed on the fly each time that they are needed.

[4]

「中間記憶を最低限しか使わず、$C_{\text{out}}[i,j]$ の各要素を計算するたびに、必要な $C_{\text{blurx}}[i,j]$ の要素を全て再計算するスケジュールです。これなら、メモリ容量は節約でき、計算順序を工夫しキャッシュをうまく使えばメモリアクセス頻度も抑えられます。また、最終結果配列への書き込みは並列に行うことができます。しかしメモリをできるかぎり節約しながら計算を行うため、過去に計算した値を覚えていることができず**再計算が多くなってしまう**欠点があります」

■Halide の最適化空間:粗メモリ、細計算



sliding window: values are computed when needed then stored until not useful anymore.

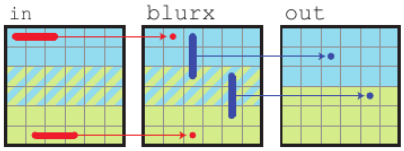
[4]

「$C_{\text{out}}[i,j]$ の各要素を計算するたびに、新たな $C_{\text{blurx}}[i,j]$ の要素を計算することから、計算の粒度は細かく、いっぽうで、計算領域全体をカヴァーできる大きなバッファを確保しておき、再計算を防ぐスケジュールです。結果、計算粒度が細かいことからキャッシュがよく効き、メモリへの読み書きも演算も最低限で済み、いいとこづくめに聞こえます。しかし、計算をメモリにとっておいて後から使うことから、計算の順序に依存関係ができてしまうため、**並列化できない**という欠点があります。

なお、実際には最適化により、確保されるバッファのサイズは中間結果をぴったり保持できるだけ (この例だと 3 行分) になります。そのバッファをスライドさせながら使いまわすようなコードが生成されます。」

■Halide の最適化空間:細メモリ、粗計算　「計算の粒度はメモリの粒度より粗くできないため、このような組み合わせのスケジュールはありえません」

■粗メモリ粗計算と細メモリ細計算の中間



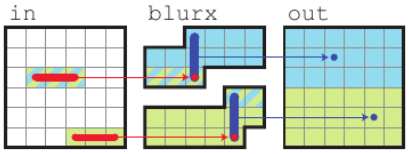
tiles: overlapping regions are processed in parallel, functions are evaluated one after another.

[4]

「配列を 1 次元または 2 次元のタイルに分割し、タイル 1 個分を記憶できるだけのバッファを用意し、その中ではすべての中間結果を保持する。タイルバッファを使いまわすスケジュールです。タ

イルの大きさにより粗メモリ粗計算と細メモリ細計算の中間にあたる様々な粒度を実現できます。タイルバッファのサイズがキャッシュに収まるときには高性能が期待できます」

■すべての中間



sliding windows within tiles:
tiles are evaluated in parallel
using sliding windows.

[4]

「配列を 1 次元または 2 次元のタイルに分割し、タイル 1 個分を記憶できるだけのバッファを用意し、その中では最低限のの中間結果を保持するバッファを使いまわす、タイルごとの計算は並列に行うというスケジュールです」

■Halide の最適化空間:再掲 「以上の解説をふまえて図 3.1 を鑑賞していただくと、Halide の哲学が理解いただけるかと思います」

■Halide がサポートするプログラム変換の種類 「以上のようなステンシル計算の実装空間を探索するため、Halide はさまざまなプログラム変換をサポートしています」

- 分割:1..(M*N) までのループを 1..M と 1..N の二重ループに分割する
- 融合:1..M と 1..N の 2 つのループを 1 つにまとめてしまう
- スレッド並列化:各添字を並列に計算
- アンロール:固定長のループを全部展開してループを消す
- ベクトル化:固定長のループをベクトル命令に置き換える
- タイル化:2 次元のブロックを用意しそれを更新
- 入れ子ループの入れ替え
- 特殊化:条件変数が真の場合と偽の場合でループを分けループの中身を簡単化

これらのスケジュールは Func オブジェクトのメソッドとして実装されています。また、これらのメソッドはいずれも更新された Func オブジェクトへの参照を返すようになっているため、複数のメソッドを連続して呼び出す構文が可能です。

詳しくは、Halide のドキュメント http://halide-lang.org/docs/ から Func オブジェクトのドキュメントも参照してください。

■プログラム変換:ループ分割

```
Func &split(Var old, Var outer, Var inner, Expr factor);
```

配列変数添字 old のループアクセスを、outer と inner を添字とする二重ループに分割します。ただし、内側ループは [0 .. factor-1] の範囲を動きます。

■プログラム変換:ループ融合

```
Func &fuse(Var inner, Var outer, Var fused);
```

inner と outer の二重ループを、fused を変数とする一重ループに融合します。融合された変数の動く範囲の大きさは、もとの二つの変数の範囲の大きさの積になります。

■プログラム変換:並列化

```
Func &parallel(Var var);
```

変数 var を並列化。var の動く範囲の数だけのスレッドを立て、var のそれぞれの値を並列に処理します。

■プログラム変換:アンロール

```
Func &unroll(Var var);
Func &unroll(Var var, int factor);
```

変数 var をアンロール。var のループを無くして展開し、var の値のすべての場合を明示的に処理するプログラムを生成します。1 引数版を使うには、var の動く範囲が既知の定数である必要があります。2 引数版は、var のループをサイズ factor の内側ループと外側ループに split した上で内側をアンロールします。

■プログラム変換:ベクトル化

```
Func &vectorize(Var var);
Func &vectorize(Var var, int factor);
```

変数 var をベクトル化。var のループを無くして var の値のすべての場合を一度のベクトル命令で処理します。1 引数版を使うには、var の動く範囲が既知の定数である必要があります。2 引数版は、var のループをサイズ factor の内側ループと外側ループに split した上で内側をベクトル化します。

■プログラム変換:タイル分割

```
Func &tile(Var x, Var y,
           Var xo, Var yo,
           Var xi, Var yi,
           Expr xfactor, Expr yfactor);
```

二つの添字 x と y の二重ループアクセスを、サイズ xfactor × yfactor のタイルに分割します。1 つのタイル内をアクセスする添字 xi と yi、およびタイルたちを参照する添字 xo と yo の四重ループに変換します。

3.5 本当のピーク性能と向き合えますか？

■ベンチマーク対象と環境 「以下のようなマシンで、拡散程式ソルバをスケジュールや問題サイズを変えて実行し、性能の変化を調べました。ノート PC だと一晩かかった LLVM のビルドが数分で終わるすごいマシンです[*1]」

マシン名	armagnac0
CPU	4 x AMD Opteron(TM) Processor 6376
freq	1.4GHz (Turbo Boost 時:2.6GHz)
cores	64
Memory	256GB

■ベンチマーク対象と環境 「シミュレーション対象アルゴリズム:

[*1] special thanks to K=N=Ninja for preparation of the machine and the useful advices.

$$C_{\mathrm{in}} = \text{initial condition}$$
$$C_{\mathrm{blurx}}[i,j] = \frac{1}{3}\left(C_{\mathrm{in}}[i-1,j] + C_{\mathrm{in}}[i,j] + C_{\mathrm{in}}[i+1,j]\right)$$
$$C_{\mathrm{out}}[i,j] = \frac{1}{3}\left(C_2[i,j-1] + C_2[i,j] + C_2[i,j+1]\right)$$

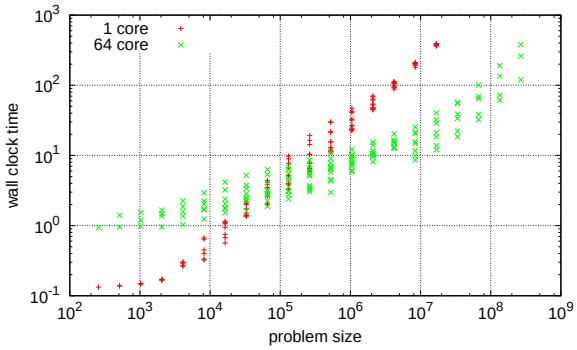
に対し問題サイズ NX,NY をそれぞれ $[16..2^{15}]$ まで変化させ、また `1 core` と `64 core` の二通りのスケジュールを用いて、コード生成時間を含む実行時間を測定しました。測定結果を表 3.1 に示します」

- `1 core` :とくにコード変換なし
- `64 core`:Halide のチュートリアルについてた以下のスケジュールを施したもの

```
cell3.split(y, yo, yi, 16).parallel(yo).vectorize(x,4);
cell2.store_at(cell3,yo).compute_at(cell3,yi).vectorize(x,4);
```

私がプレゼンをここまで進めたとき、教室が一瞬暗くなり、また明るくなった。

実行時間



実行効率

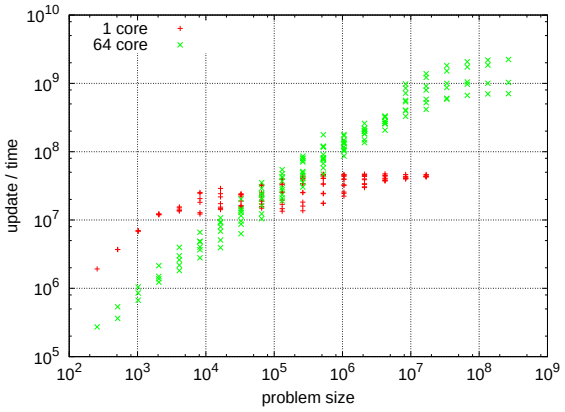


表 3.1: 問題サイズに対するコード実効時間と実効効率の変化。`1 core` と `64 core` のスケジュールの違いについては本文を参照。

3.6 見滝原中学校　Ninja 結界

そこには腰を落とし、片足を水平近くまで伸ばして急制動の姿勢をとる一人の少女の姿が！

「ドーモ、ホムラ=サン。シ=マ=カゼ=Ninja デス。スピードなら誰にも負けませんよ？」

“death-devil-wind” を意味する古代パワーワードを名乗る彼女はあからさまに Ninja！ Ninja の登場である！ と見る間に少女が突入してきた方向からガラス、土砂片の奔流！ 一瞬前まで整然とプレゼンが進行していた教室はたちまち空気、椅子、机、肉片のブレンダーめいたタツマキジゴクと化した！ シ=マ=カゼ=Ninja がトレインしてきた超鋭角の衝撃波面が教室という境界条件で複雑に反射して乱流化したものである。まさにデス・デビル・ウインド、死の魔の風だ！

視界が晴れたときには生徒の半分は絶命！ 残り半数近くも重篤な NRS により失禁。とある黄髪の生徒などは首と胴体がサヨナラしたうえで失禁するなど、あまりにもアワレな姿を晒していた。シ=マ=カゼ=Ninja と名乗る少女はセーラーフクめいた大きな襟を身につけている。だが、襟以外の部分といえば極ミニスカートしかなく、あまり豊満でない胴体前部装甲に至っては、わずかに二本のサスペンダーめいたヒモが覆うばかり！ 襟の無いセーラーフクはワイシャツだが、セーラーフクの無い襟はワイセツだ!!

「この**時**を…待っていたッ!!」

赤縁の眼鏡をかなぐり捨て、三つ編みにした長い黒髪をほどき、私も一瞬のうちにやはりセーラーフクを模した超常の衣装を纏う。私の衣装の方は長袖にスカートの中まであるタイツで、猥褻が一切無い。ほむほむ $\neq$ 変態。いいね？「ドーモ、シ=マ=カゼ=サン。ホムラです」

アイサツが済むや、シ=マ=カゼ=Ninja は巨大なハサミめいて大ぶりの半月形の護拳のついた片刃剣を構え、さっそく Ninja 特有のインタビューをしかけてきた。

「ホムラ=サン。そのマシンには Opteron$^{(TM)}$ 6376 が四枚刺さってるんですよね？ だとしたらピーク単精度浮動小数演算性能はいくらですか？」

「Opteron$^{(TM)}$ 6376 は 16 コアで、1 コアごとに 128bit fma 演算器がついているわ。単精度は 32bit であり、fma は 2 浮動小数演算 (flop) ことを考慮すると 128bit 演算器でクロックあたり 8flop こなせるから $8\text{flop/core/clock} \times 16\text{core/cpu} \times 2.6\text{GHz} \times 4\text{cpu} = 1331.2\text{GFlop/s}$ になるわね」

「はっやーい！ じゃあホムラ=サン。あなたのプログラムの実効性能はいくらかしら？」

「グラフから読み取れる最大性能は 2×10^9update/s で、今のプログラムは 1update あたり単精度 6 演算だから、12GFlop/s ってところかしらね」

「おっそーい！ ピーク性能の 1% しか出ていないようじゃ、Ninja の私には到底追いつけないね！ 私なんかマルチスレッドと FMA4 命令を駆使してピーク性能の 96.9% [*2] は出しちゃうんだから！ じゃあ、いっくよー?!」

ゴウランガ！ シ=マ=カゼ=Ninja の姿が 2 人、4 人、8 人！、いや 64 人！ あまりにも高速過ぎる動きにより残像が質量を持って見える、これぞ Ninja 伝統のブンシン=ジツである！ しかも爪先立ちした分身像のそれぞれは失神し倒れ伏した生徒の首筋を正確に捉えている！ 返答いかんでは残りの生徒の命も無いぞ、という無言のメッセージ状況を作り出して、何を問おうというのか?!

「おちついてホムラちゃん！ ここは Ninja 結界よ！ Ninja の肉体は物理法則を無効化する代わりに論理ダメージを物理ダメージに変換する薄膜に包まれているわ！ 相手の主張の痛いところを突いて相手に論理ダメージを与えるのよ！ たとえば…」

「それには及ばないわ」金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！ 金属打撃音！

シ=マ=カゼ=Ninja の動きが止まる。ただでさえ布面積の少ないコスチュームはハチノスめいてズ

[*2] using `https://github.com/Mysticial/Flops`

タズタの断片となり宙を舞う！

「はうっ…この私がやられるなんてっ」

アニメーエーショーン上の倫理規制により、シ=マ=カゼ=Ninja はあまり豊満ではないボディの特定二箇所が、宙を舞う布とカメラとを結ぶ延長線上に位置するよう運動せざるをえない！ 二軸三次元拘束最適化問題！ これは Ninja 同士のイクサにおいてあまりにも大きなディスアドバンテージだ！

「残念だけど、私、スピードキャラには相性がいいのよ。時間を止めている間に機関銃弾を設置すれば、相手がわざわざ相対運動エネルギーを上乗せしてくれるんだもの。それに `Flops` ベンチマークは確かにピーク性能を測る上では便利だけれども、fma 命令をひたすら発行しているだけで、実用的なアプリケーションと同列に比較できるものとはとても言えないわね」

「ぐっ…!!!」

ほむほむの実用性攻撃！ 拡散方程式アプリの方もシンプルすぎて実用性が疑わしいことは棚にあげた論理攻撃だ！ 空中を舞う布片の動きに合わせてスローモーで宙を舞いながら、シ=マ=カゼ=Ninja のプライドもぬののふく (元) と同様ズタズタだ！

「それにダウンロードしてきたプログラムを実行しているだけでは、Ninja どころかただのスクリプトキディ以下の存在よ」

もはや記事のなんかとは一切関係が無い人格攻撃！ シ=マ=カゼ=(自称)Ninja は致命的な論理ダメージにより作り出されたクレーターの中でヤムチャ=アティチュードに追い込まれてしまう！ ちなみに大事なところは地形破片やカメラアングルにより奥ゆかしく隠されているので青少年の倫理規定もオッケーだ！

「さあ、ハイクを詠みなさい」

「うう…速いだけじゃ、駄目なのね、インガオホー」

シ=マ=カゼ=(自称)Ninja はいつのまにか何者かが体内に設置したバクダンにより爆発四散！ そして時は動きだし、大量の空薬莢が地面に落ちる音をたてる。

風がカラテ残滓を運びさった後には、魂を代償に生き延びる力を契約した者たち——5 人の魔法少女の姿だけが立っていた。

主を失った Ninja 結界がかしぎ、ゆらぎ、はがれ、閉じてゆく。

「はいはいー、おしごとしゅーりょー。今回も楽勝だったわー。ていうか、あんたもさー。ピーク性能の 1% 以下とかさすがに無くね？」

「単純な拡散方程式では帯域幅に対する演算性能が少なすぎたのね。複数の時間ステップを融合 (inter-timestep fusion) させればあるいはもう少し性能の向上が図れたのかもしれないけれど、残念ながらそこまで行かなかったわ」

その後、魔法少女 5 人の力を合わせた儀式魔法（マジカルステージ）により教室は何事もなかったかのように復旧した。え、マミさんはアバってたじゃないかって？ 何を言ってるんだい。あれは抜け殻だよ。ソウルジェム化していなければ即死だったんだから、感謝してほしいな ／人◕‿◕人＼

こうして見滝原中学校の平和と魔法少女の秘密は守られたのである。だが、儀式魔法とは、その強大さゆえに、それを行使するたびに世界線が少しづつずれていく[*3]ことを知っている者は——知っていて利用している者は——ただひとり。

多世界解釈（リーディングシュタイナー）を具有する彼女は、常識的な光景へと揺り戻す教室の中で教壇に立っていた。

「発表を続けます」

[*3] 訳: 無茶を言って締切り伸ばしてもらったのでストーリーの整合性を無視して執筆を続けます。もうちょっとだけ続くんじゃ。

3.7 Halide って、ほんと簡単

「Halide 論文では、ステンシル計算を定義した上で様々なスケジュールを適用することで、生成プログラムの性能を様々にいじくれる、と主張しています。しかし、誤ったスケジュール (配列の範囲外アクセスなど) を設定してしまうとコード生成タイミングで実行時エラーになってしまい原因究明が困難であったことから、自由なスケジュール設定ができていませんでした。

ところが、Halide が（ここ数日のバージョンアップで）無効なスケジュールを検出すると「有効なスケジュールの一覧」を出力するようになったことから、実行時エラーで落ちないスケジュールを設定するのが容易になりました。

そこで、Halide のスケジュール空間を自動探索するプログラムを書いて走らせてみました」

■Inter-timestep fusion 「拡散方程式のプログラムは計算に比べてとてもメモリ負荷が大きいです。理想的な場合でも 1 load + 1 store あたり 10 演算なので、単精度として 0.8B/F。1 演算あたり、0.8 バイトのデータが転送されてこなければなりません。実際には、無駄なメモリアクセスがある場合はより高い B/F が要求されますが、いっぽうで無駄な演算がある場合にはこれより低い B/F で済む可能性もあります。

これに対し、当該マシンの演算帯域比は、

- 演算: 1331.2GFlop/s
- 帯域: 約 50GB/s × 4 socket = 200GB/s

で、約 0.15 にすぎません。拡散方程式の数値解法を単純に実装して最適化しようとした場合、メモリ帯域幅がボトルネックになりがちであることがわかります。

アルゴリズム固有の限界 0.8B/F を超えて転送量を圧縮する方法として、複数の時間刻みを融合する方法が考えられます。n ステップを融合して一つの更新操作としてプログラムできれば、理想的な演算帯域比は $0.8/n$ B/F になり、マシンの演算帯域比に近づけられるものと思われます」

■Halide における自動性能チューニング 「Halide の生成するコードを制御するパラメータには以下のような種類が考えられます [*4]」

1. **User option**: Inter-timestep fusion のような、プログラムの構造そのもの、配列変数の数を変えてしまうような変換 (Halide の schedule の範囲に収まらないプログラム変換)
2. **Loop option** ループ分割、融合、タイリングといったループの個数にまつわる選択肢 (整数パラメータの個数を増減させる)
3. **Timing option** 計算タイミング、ストアタイミングといった整数パラメータの個数を変えない選択肢
4. **Int option**: ベクトル長、アンロール長といった整数パラメータの選択肢

「Halide においてユーザが記述する C++ プログラムは実際の数値計算プログラムの生成・実行ドライバに過ぎません。したがって、コード生成の選択肢はすべて C++ の普通の値として表現できます。Halide が仮にテンプレートライブラリのような、C++ のコンパイル時を利用してコード生成するライブラリだったとしたら？ User option を変化させる Inter-timestep fusion のような変更や、Loop option を変化させるような変更を、実行時の値に基づいて行うことは困難だったでしょう。したがってこれらの option はコンパイル時定数である必要があったでしょう。また Int option であるアンロール長、ベクトル長などもコンパイル時定数にしておいた方がよさそうです。実行時にできることは、あらかじめ生成しておいた幾つかのパターンの中から一つを選択することがせいぜいで

[*4] この分類は Halide の用語ではなく著者独自のものです

しょう。

これに対し、Halide はユーザーの書いたプログラムの実行時に数値計算コードを生成しますから、ユーザープログラムから見たコード生成記述の自由度は格段に上がります。Halide を用いたプログラム実行速度最適化アプリケーションは、Halide ユーザーにとっては C++ の普通の多引数関数の探索問題のように書くことができるのです。たとえば、N_FUSION 段の時間ステップを融合するこうどなぷろぐらむを生成する操作も、Func の配列に対するループとして、以下のように簡潔に書けます」

```
  /* 数値計算アルゴリズムの記述 */
  Halide::Func cell2[N_FUSION], cell3[N_FUSION]; // 可変長配列を利用
  for(int i_f = 0; i_f < N_FUSION ; ++i_f) {
    if (i_f==0) {
      cell2[i_f](x,y)=a * inPar(x,y) + b * inPar(x+1,y) + b * inPar(x-1,y) ;
    } else {
      cell2[i_f](x,y)=a * cell3[i_f-1](x,y) + b *  cell3[i_f-1](x+1,y)
                                            + b * cell3[i_f-1](x-1,y);
    }
    cell3[i_f](x,y)= a * cell2[i_f](x,y) + b * cell2[i_f](x,y+1)
                                            + b * cell2[i_f](x,y-1) ;
  }
```

「このプログラムに対し、例えば以下のようにスケジュールを指定します」

```
  /* スケジュールの記述 */
for(int i_f = N_FUSION-1; i_f >=0 ; --i_f) {
  if (i_f==N_FUSION-1){
    cell3[i_f].tile(x,y, xo,yo, xi, yi, N_TILE_X, N_TILE_Y).fuse(xo,yo,nid)
              .parallel(nid).vectorize(xi,N_VECTOR).unroll(xi,N_UNROLL);
  }
  else{
    cell3[i_f].store_at(cell3[N_FUSION-1],nid).compute_at(cell3[N_FUSION-1],yi)
              .vectorize(x,N_VECTOR).unroll(x,N_UNROLL);
  }
  cell2[i_f].store_at(cell3[N_FUSION-1],nid).compute_at(cell3[N_FUSION-1],yi)
            .vectorize(x,N_VECTOR).unroll(x,N_UNROLL);
}
```

「このプログラムは、先ほどから例に出ている拡散方程式のステンシル計算を N_FUSION ステップ分融合させたものになっています。Halide でこのような多段のステンシル計算を記述するにあたっては、図 3.2 のようなイメージが助けになります」

「まず、数値計算アルゴリズムの記述にあたっては、図 3.2 左側のように、データの流れにそって冒頭からプログラムを書いていきます。一番先頭の Func だけは、メモリ上のデータ Image へのポインタでありデータフローの入り口である InputParam から、初期条件を入力するため、特別扱いが必要です」

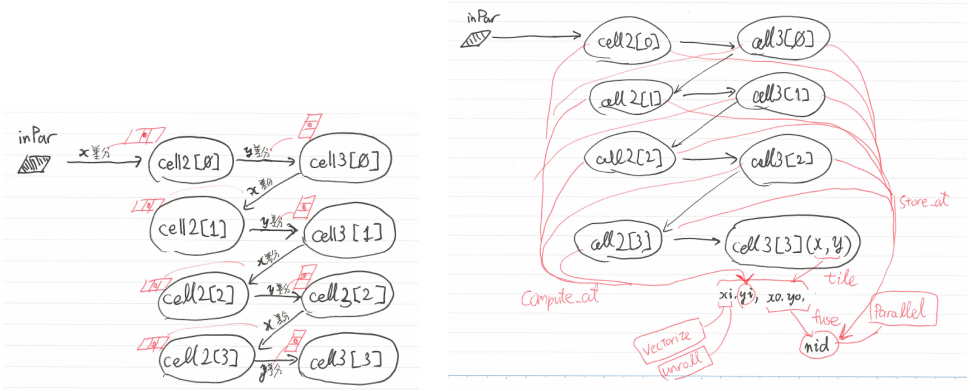


図 3.2: Halide における複雑なステンシル計算の書き方。N_FUSION=4 の場合。

「つぎに、スケジュールの指定にあたっては、アルゴリスム記述とは逆に、データフローに逆らって記述することを Halide チュートリアルでは推奨しています。まずステンシル計算の最後段にあたる cell3[N_FUSION-1] の Loop option を指定することで、全体のループ構造が決定します。図 3.2 左側では、cell3_[3] に対して tile,fuse と言った指定をすることで、デフォルトでは x, y の二重ループで行われる計算を (内側ループ変数から順に)xi, yi, nid の三重ループとし、うち nid に関してはマルチスレッド化、xi についてはベクトル化とアンロールといったスケジュールを指定しています。

最後段以外の変数に関しては、基本的に loop option は指定せず、timing option を指定することで、最後段 cell3_[3] を計算するループの中に順次組み入れていくことになります。timing option では、ループ添字変数をひとつ指定し、その変数がインクリメントされるごとに、計算・ストアをまとめて行います。例えば、cell2[2].store_at(cell3[3],nid) というメソッド呼び出しでは、nid がインクリメントされるまでの間データを保持しつづられるだけのバッファを作ることを指定しています。nid はタイル分割のときに作られた変数で、また parallel 化されていますから、これは、各スレッドごとに、タイル 1 枚の大きさに等しいバッファを作って cell2[2] の一部を記憶することを意味します。また cell2[2].compute_at(cell3[3],yi) というメソッド呼び出しでは、yi がインクリメントされるごとに、手に入る計算結果は再利用しながら、cell2[2] を計算することを指定しています。

この timing option に指定するループ添字変数は、Halide の探索空間である計算とメモリの粒度を操っています。ループの内側の添字を指定するほど、粒度が細かくなるわけです。また、計算の粒度がメモリの粒度より粗くできない理由も理解できます。粒度の逆転は、バッファが確保されるループよりも外側で計算をしてそのバッファに書き込むことを意味します。そのようなプログラムはコンパイルを通らないか、範囲外アクセスエラーを引き起こすでしょう。

さきほど、loop option は最後段の Func 変数にだけ指定する、と書きましたが、途中段の Func に対して compute_root() スケジュールを指定することで、その段の計算を、最も外側のループのさらに外側で行うよう指定できます (最も粗い計算粒度)。そうした場合、compute_root() 指定された変数には独自の loop option を指定できます」

3.8　もう、Halide にも頼らない

「あらためて、以下のようなループ構造に対し」

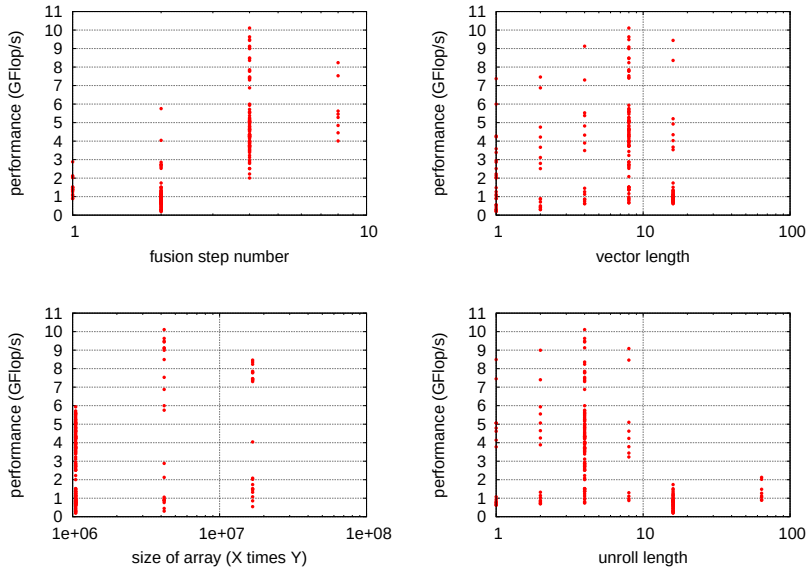


表 3.2: テーブル分割スケジュールによる性能空間探索結果。

```
cell3[i_f].tile(x,y, xo,yo, xi, yi, N_TILE_X, N_TILE_Y) // タイル分割
          .fuse(xo,yo,nid) // タイルを識別する外側添字は x,y まとめてナンバリング
          .parallel(nid)   // そのタイルごとに並列化
          .vectorize(xi,N_VECTOR) // 再内側添字 xi はまず N_VECTOR 個づつベクトル化
          .unroll(xi,N_UNROLL);   // さらに N_UNROLL 個づつアンロール
```

「以下のようなパラメータを振って、パラメータの関数としての実行性能を計測します」

```
int N_FUSION = 4;    // [1..32]  User option   : Inter-timestep fusion の段数
int CELL2_CHOICE=4; // [0..8]   Timing option : cell2(y 差分) のタイミング
int CELL3_CHOICE=4; // [0..17]  Timing option : cell3(x 差分) のタイミング
int N_VECTOR=8;      // [1..8]   Int option    : マシンベクトル長
int N_UNROLL=4;      // [1..32]  Int option    : アンロール段数
int N_TILE_X=64;     // [1..512] Int option    : タイルバッファのサイズ (x)
int N_TILE_Y=64;     // [1..512] Int option    : タイルバッファのサイズ (y)
```

「結果を表 3.2 に示します。アンロール長は 4、ベクトル長は 8、inter-timestep fusion の深さは 4 が最適、などがわかりましたが、従来の性能を大きく改善することはできませんでした」

「性能が向上しない原因として、バッファサイズが並列タスク分割に使うタイルサイズと同じサイズに固定されていて変更できないことが考えられます。そこで、タイル分割を二重にし、外側のタイルごとにスレッド並列化し、内側のタイルのサイズのバッファを作る新たな Loop option を考えます」

「このループ構造のもとで、外側と内側のタイルのサイズを変えて測定した実効性能の分布を表 3.3 に示します。50Gflops 近い性能は出ていますが、残念ながらこれでもマシンピーク性能の 4% にすぎません」

「Halide の生成したアセンブリを見てみたところ、だいたい浮動小数演算命令 1 つに対し整数演算命令が 6 つもの割合で存在しました。整数演算ではおもにアドレス計算を行っているものと思われます。Halide の、スケジュールに基づくループ変換機能は確かに強力でしたが、Halide の生成す

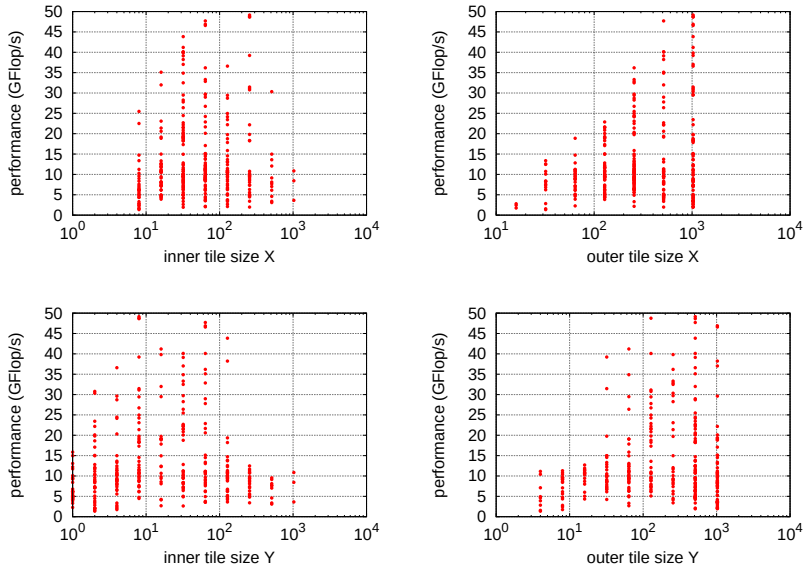


表 3.3: 二重テーブル分割スケジュールによる性能空間探索結果。内側と外側のテーブルのサイズの関数としての実効性能を示す。

るアセンブリは、一般の数値計算アプリケーションを普通のコンパイラでコンパイルして生成されるアセンブリと比較しても、浮動小数演算命令密度が低すぎ、見るからに非効率です」

「ステンシル計算のホットスポットを浮動小数命令で埋め尽くさない限り、ピーク性能に近い性能は出ません。さらにピーク性能の 8 割、9 割を極めようと思ったら、浮動小数演算命令のレイテンシに注意を払う必要があります。fma 命令のレイテンシは典型的には 4,8,12 サイクルなどです。`Flops` ベンチマークなどの演算性能を出すことに特化したコードでは fma 命令を敷き詰めた上で、各命令の結果を 12 サイクル以上後まで利用しないようにしています」

「Ninja 性能を出せるプログラムの形というのは、このようにきわめて限られており、一般のアプリケーションのホットスポットを、そのような形に変換するのは困難で、多くの場合には不可能です」

ひとときの間、教室を満たしていたギラギラとした Ninja アトモスフィアは、空調（エアコンディショナー）によって、人類の生存にとって快適な涼しい空気にすっかり置換されてしまっていた。

自動チューニングによる Ninja 性能の達成——そんな途方もない願いが叶うとしたら、Halide や Paraiso のように、ステンシル計算を抽象的に捉え、無数の最適化の可能性を作り出すほかに希望はない。しかし、彼女の前にはまだまだ無数の乗り越えるべき世界線が横たわっているのであった。

3.9　見滝原駅前喫茶店

「ねえ、テンコウセイ＝サン。わたしたちは魔法少女ですよね。魔法少女は Ninja と戦いますか。おかしいと思いませんか、あなた？」

「そんなことはないわ。私たちの世界観において魔法はじつのところ異星人の科学がもたらしたものだし、Ninja とはコンパイラ最適化では及びもつかない、ピーク性能に実際近い実効性能を叩き出す手動最適化のタツジンを指す学術用語なの [1,5]。高度に発達した科学が Ninja すら凌駕したとき、科学は魔法をもうわまわる何かに進化するのではないのかしら」

「科学と魔法が交差するとき——技術的特異点——か。それは、軌道エレベータも勝手に建つかもしれないなあ」

「今回は実際に Ninja が登場する世界線までたどりついたおかげで、Ninja 根源との距離が分かったのは一定の収穫ね。Halide で Ninja 性能を出すところまで行かなかったのが残念だわ」

「大丈夫だよほむらちゃん。絶対何とかなるよ」

「その通り！ 今回のテーマは『残念』だからな。むしろ残念が好都合さ。Rocky, 食うかい？」

「そんなことより、この原稿は C81 の原稿を~~パクリ~~平行世界という設定なのだけど、C81 のソースの末尾にこんなコメントを見つけたのよ。この遺言泣けるわね」

キュウベえと契った少女たちは紙片を覗き込む。

```
% Paraiso で性能が出なかったら継続エンドに分岐
% section{最後に残った道しるべじゃなイカ？ }
% Paraiso で性能が出たらこの True エンドに分岐！
% section{Haskell は、私の最高の友達でゲソ！ }
```

「ああー、これは泣けるねー。さやかちゃん泣いちゃうわー」

「たしかに、これは遺言だね。今回も継続エンドに分岐しそう」

「いつか、著者が Ninja 真実にたどり着けるよう、私も祈っているわ」

なぜなら、著者のゴールは、ストーリー必然的に私のゴールでもあるのだから。

ネタ元

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

- 5pb・ニトロプラス「シュタインズ・ゲート」
- 安部真弘著・原作「侵略！ イカ娘」
- ブラッドレー・ボンド、フィリップ・ニンジャ・モーゼズ「ニンジャスレイヤー」
- @dif_engine 蓮物語、殺物語
- 角川ゲームス/DMM.com「艦隊これくしょん」
- 鎌池和馬原作・錦織博監督「劇場版 とある魔術の禁書目録 —エンデュミオンの奇蹟—」
- Magica Quartet 原作・新房昭之監督「魔法少女まどか☆マギカ」
- 東堂いづみ・東映アニメーション「おジャ魔女どれみ」
- 戸塚たくす「オーシャンまなぶ」
- TRIGGER/中島かずき原作・今石洋之監督「キルラキル」

また、執筆にあたって以下の文献を参考にいたしました。

参考文献

[1] Kim, C., Satish, N., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. Closing the ninja performance gap through traditional programming and compiler technology. Tech. rep., Technical report, Intel Labs, 2011.

[2] Muranushi, T. Paraiso: an automated tuning framework for explicit solvers of partial differential equations. *Computational Science & Discovery 5*, 1 (2012), 015003.

[3] Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S. P., and Durand, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*, 4 (2012), 32.

[4] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., and Amarasinghe, S. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices 48*, 6 (2013), 519–530.

[5] Satish, N., Kim, C., Chhugani, J., Saito, H., Krishnaiyer, R., Smelyanskiy, M., Girkar, M., and Dubey, P. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *ACM SIGARCH Computer Architecture News* (2012), vol. 40, IEEE Computer Society, pp. 440–451.

※この物語はフィクションであり、実在の人物・団体・事件・製品および艦船とは関係がありません。

第4章

ご注文は依存型ですか？

— @tanakh

こころぴょんぴょん待ち？
考えるふりして
もうちょっと！ 近づいちゃえ。

```
main :: IO ()
main = mapM_ (putStrLn . ah) [1..105]

ah :: Int -> String
ah n = fromMaybe (show n) $ f <> g <> h where
  f = if n `mod` 3 == 0 then Just "あぁ＾〜〜、" else Nothing
  g = if n `mod` 5 == 0 then Just "こころが" else Nothing
  h = if n `mod` 7 == 0 then Just "ぴょんぴょんするんじゃ〜" else Nothing
```

簡単には教えないっ！
こんなに好きなことは〜〜〜〜〜（テンテテン！）
内緒なのぉ〜〜〜〜！

```
1
2
あぁ＾〜〜、
4
こころが
あぁ＾〜〜、
ぴょんぴょんするんじゃ〜
8
あぁ＾〜〜、
こころが
11
あぁ＾〜〜、
13
ぴょんぴょんするんじゃ〜
あぁ＾〜〜、こころが
16
```

```
17
あぁ＾〜〜、
19
こころが
あぁ＾〜〜、ぴょんぴょんするんじゃ〜
22
23
あぁ＾〜〜、
こころが
26
あぁ＾〜〜、
ぴょんぴょんするんじゃ〜
29
あぁ＾〜〜、こころが
31
32
あぁ＾〜〜、
34
こころがぴょんぴょんするんじゃ〜
あぁ＾〜〜、
37
38
あぁ＾〜〜、
こころが
41
あぁ＾〜〜、ぴょんぴょんするんじゃ〜
43
44
あぁ＾〜〜、こころが
46
47
あぁ＾〜〜、
ぴょんぴょんするんじゃ〜
こころが
あぁ＾〜〜、
52
53
あぁ＾〜〜、
こころが
ぴょんぴょんするんじゃ〜
あぁ＾〜〜、
58
59
あぁ＾〜〜、こころが
```

61
62
あぁ＾〜〜、ぴょんぴょんするんじゃ〜
64
こころが
あぁ＾〜〜、
67
68
あぁ＾〜〜、
こころがぴょんぴょんするんじゃ〜
71
あぁ＾〜〜、
73
74
あぁ＾〜〜、こころが
76
ぴょんぴょんするんじゃ〜
あぁ＾〜〜、
79
こころが
あぁ＾〜〜、
82
83
あぁ＾〜〜、ぴょんぴょんするんじゃ〜
こころが
86
あぁ＾〜〜、
88
89
あぁ＾〜〜、こころが
ぴょんぴょんするんじゃ〜
92
あぁ＾〜〜、
94
こころが
あぁ＾〜〜、
97
ぴょんぴょんするんじゃ〜
あぁ＾〜〜、
こころが
101
あぁ＾〜〜、
103
104

あぁ＾〜〜、こころがぴょんぴょんするんじゃ〜

ふわふわどきどき内緒ですよ
初めが肝心詰んだ詰んだ
ふわふわどきどき内緒だって
いたずら笑顔でぴょん！　ぴょん！

（GHC の）バージョン上げた途端
見知らぬ世界（Type Level）へと
そんなのないよ　ア・リ・エ・ナ・イ

```
{-# LANGUAGE DataKinds              #-}
{-# LANGUAGE FlexibleInstances      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds              #-}
{-# LANGUAGE UndecidableInstances   #-}
{-# LANGUAGE ScopedTypeVariables    #-}
{-# LANGUAGE TypeFamilies           #-}
{-# LANGUAGE TypeOperators          #-}

import           Control.Monad
import           Data.Maybe
import           Data.Monoid
import           Data.Proxy
import           Data.Type.Equality
import           GHC.TypeLits
```

それがあり得るかも　Type Level の異次元
たぃ↓ぷふぁ↑みりぃ→　覗・い・た・ら！

```
type family If (c :: Bool) (t :: a) (e :: a) ::a where
  If True t e = t
  If False t e = e

type family Reverse (acc :: [t]) (n :: [t]) :: [t] where
  Reverse acc '[] = acc
  Reverse acc (x ': xs) = Reverse (x ': acc) xs

type family Map (f :: a -> b) (xs :: [a]) :: [b] where
  Map f '[] = '[]
  Map f (x ': xs) = f x ': Map f xs

type family FromMaybe (a :: t) (b :: Maybe t) :: t where
```

```
  FromMaybe a Nothing = a
  FromMaybe a (Just b) = b

type family Mappend (a :: Maybe [Symbol]) (b :: Maybe [Symbol])
    :: Maybe [Symbol] where
  Mappend a Nothing = a
  Mappend Nothing b = b
  Mappend (Just a) (Just b) = Just (Concat a b)

type family Concat (a :: [t]) (b :: [t]) :: [t] where
  Concat '[] ys = ys
  Concat (x ': xs) ys = x ': Concat xs ys
```

型レが型レを見つめてました
なんで?なんで？ ふたつある？ (うそ！)
困りますね (きっと) おんなじ趣旨 (だから)

```
type family Div (a :: Nat) (b :: Nat) :: Nat where
  Div a b = Div' 0 a b

type family Div' (dep :: Nat) (a :: Nat) (b :: Nat) :: Nat where
  Div' 100 a b = 0
  Div' dep a b =
    If (a + 1 <=? b) 0 (1 + Div' (dep + 1) (a - b) b)

type family Mod (a :: Nat) (b :: Nat) :: Nat where
  Mod a b = Mod' 0 a b

type family Mod' (dep :: Nat) (a :: Nat) (b :: Nat) :: Nat where
  Mod' 100 a b = a
  Mod' dep a b =
    If (a + 1 <=? b) a (Mod' (dep + 1) (a - b) b)
```

何を (見つめるの？ 型でしょ！)
型だけ見せるよ〜〜〜
(これは夢　型の夢　コンパイルして　お・し・ま・い！)

```
type family ShowNat (n :: Nat) :: [Symbol] where
  ShowNat n = Map ShowDigit (ToDigits n)

type family ShowDigit (n :: Nat) :: Symbol where
  ShowDigit 0 = "0"
  ShowDigit 1 = "1"
```

```
  ShowDigit 2 = "2"
  ShowDigit 3 = "3"
  ShowDigit 4 = "4"
  ShowDigit 5 = "5"
  ShowDigit 6 = "6"
  ShowDigit 7 = "7"
  ShowDigit 8 = "8"
  ShowDigit 9 = "9"

type family ToDigits (n :: Nat) :: [Nat] where
  ToDigits n = Reverse '[] (ToDigits' n)

type family ToDigits' (n :: Nat) :: [Nat] where
  ToDigits' 0 = '[]
  ToDigits' n = (n - (Div n 10) * 10) ': ToDigits' (Div n 10)
```

型もぴょんぴょん（出力）可能！
楽しさ求めて　もうちょっとはじけちゃえ　(ぴょんぴょんと！)

```
class KnownSymbols s where
  symbolVals :: Proxy s -> String

instance KnownSymbols '[] where
  symbolVals _ = ""

instance (KnownSymbol x, KnownSymbols xs) => KnownSymbols (x ': xs) where
  symbolVals _ =
    symbolVal  (Proxy :: Proxy x) ++
    symbolVals (Proxy :: Proxy xs)
```

一緒なら素敵だーい！
型に言わせたいから（言いなさい）

```
type family Ah (n :: Nat) :: [Symbol] where
  Ah n = FromMaybe (ShowNat n) (Mappend (F n) (Mappend (G n) (H n)))

type family F (n :: Nat) :: Maybe [Symbol] where
  F n = If (Mod n 3 == 0) (Just '["あぁ＾〜〜、"]) Nothing

type family G (n :: Nat) :: Maybe [Symbol] where
  G n = If (Mod n 5 == 0) (Just '["こころが"]) Nothing

type family H (n :: Nat) :: Maybe [Symbol] where
```

```
  H n = If (Mod n 7 == 0) (Just '["ぴょんぴょんするんじゃ〜"]) Nothing
```

こころぴょんぴょん待ち?
考えるふりしてもうちょっと近づいちゃえ (ぴょんぴょんと)

```
main :: IO ()
main = putStr $ symbolVals (Proxy :: Proxy Main)

type Main = Main' 1

type family Main' (i :: Nat) :: [Symbol] where
  Main' 105 = '[]
  Main' i = Concat (Concat (Ah i) '["\n"]) (Main' (i + 1))
```

簡単には（コンパイル結果を）教えないっ
こんなに遅いことは (遅いってことは…わわわ！)
内緒なの〜〜〜〜！

```
> time ghc kokoropyonpyon.hs
[1 of 1] Compiling Main             ( kokoropyonpyon.hs, kokoropyonpyon.o )
^C

real    10m45.233s
user    10m2.803s
sys     0m9.204s
```

ハスケル「いっそ依存型になりてぇ」

会員名簿じゃないカ?

@master_q　(1 章)

ATS たん、モフ! モフ!

@dif_engine　(2 章)

生涯パチュマリ一筋

@nushio　(3 章)

Haskell やって三次元嫁ができました。

@tanakh　(4 章)

あぁ ^ 〜〜、こころがぴょんぴょんするんじゃ〜


```
.S(S(KS)K)(S(S(KS)K)(S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))).S(S(KS)K)(S(K(M(S(S(KS)K)I)))(S(S(KS)K)(S(S(KS)K)I(M(S(S(KS)K)I)))))).S(S(KS)K)(S(S(KS)K)(S(S(KS)K)I(S(S(KS)K)(M(S(S(KS)K)I)))))).
(K(S)(K(SK)S)S·((((I(K(SK)S)S)(K(SK)S)S)(I(K(SK)S)K)(M(S(K(SK)S)S)(K(SK)S)S)(K(SK)S)S·(((((I)
```


「簡約(かんやく)　λ(らむだ)カ娘(かむすめ)　巻の七(かんのなな)」

2014 年 8 月 17 日 コミックマーケット 86 初版発行

発行：参照透明な海を守る会

http://www.paraiso-lang.org/ikmsm/

ikmsm-authors@googlegroups.com

著者：@master_q, @dif_engine, @nushio, @tanakh

表紙：馬三郎 叉吉 様 http://www.pixiv.net/member.php?id=3313233

印刷：丸正インキ有限会社 コミック事業部 様 http://www.marusho-ink.com/

参照透明な海を守る会