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. 十冊目なんて平気なんですか？ 怖く……ないんですか？
A. 平気ってことはないし、辛かったりもするけど、同人誌を出し続けることで助かってる面もあるわけだし。やりがいはあるよね

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

この本は、コミックマーケット 80 での「簡約! λ カ娘」、コミックマーケット 81 での「簡約!? λ カ娘 (二期)」、さらにコミックマーケット 82 での「簡約!? λ カ娘 (算)」、に続く、さらにさらにコミックマーケット 83 での「簡約!? λ カ娘 4」、に続く、もーさらにさらにコミックマーケット 84 での「簡約!? λ カ娘 Go!」、に続く、そしてコミックマーケット 85 での「簡約!? λ カ娘 Rock!」、コミックマーケット 86 での「簡約 λ カ娘 巻の七」、コミックマーケット 88 での「簡約!? λ カ娘 8」、に続く、コミックマーケット 90 での「簡約!? λ カ娘 9」、に続く、十冊目の関数型イカ娘の本でゲソ。コミック連載終了に安部真弘氏に「おつかれさまでした!」の気持ちをこめて、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。

第1章

モナドとひも

— @myuon_myon

概要

String Diagram を用いて操作を図示することの雰囲気を感じ取ってもらうことが目標です。

1.1 としょかん

「話がしたい。13 時？」

彼女からのメッセージを見たときは思わず笑ってしまいそうになった。相変わらず簡素なメッセージで、そっけなく見えるけどこれでも彼女なりに言葉を尽くしているらしいことは私もそれなりに付き合いが長くなってからわかった。本人曰く句読点を打つ手間は誠意と敬意の表れであり、これは一種の敬語であり、それを使う手間を惜しまない相手にしか使わない、そういう特別なものらしい。

その時は句読点が敬語のわけないでしょ、とは言ったけれど、そういうことではない、みたいな返しをもらった気がする。

私は了解の意味のスタンプを押していよいよ観念して布団から起き上がった。

一日ぐうたらお布団から動かない計画はあえなく破綻したけれど、逆に破綻した方が健康には間違いなくいいことくらいは分かっている。分かっているのだけど、こうして誰かに呼び出されでもしなければお布団から抜け出せないほどに私の体は自堕落生活をエンジョイし始めていた。夏休みとは、自由な時間とは、無責任とはかくも人を荒廃へと追いやるのだということを学べただけでも私は大学へと来たかいがあったというものではないだろうか。

「最近 Haskell を始めた」

「へぇ、またどうして？」

「モナドの勉強をしていて、その流れで」

私がずぞぞと音を立てて残り少ないアイスティーを吸うと、彼女は私のカップを見て露骨に不快そうな顔をした。

図書館というのは私が知る限り雑音や会話する声を忌み嫌う人たちが集う場所だと思っていたのだけれど、大学の図書館には気がつくとおしゃべり空間ことラーニングコモンズという、主に言語コミュニケーションにより議論ができるような場所が用意されていた。

クーラーが効いていて、快適な椅子と机が用意されていて、なによりホワイトボードがあるので私も彼女もここをいたく気に入っていた。夏休み開始直後ということもあって比較的空いていたので、椅子になるクッションが用意されている 4 人がけのテーブルを私達は遠慮なく陣取ることにした。

「モナドは知ってるよね？」

「return と bind を備えた型クラスってことぐらいはね」

彼女は嬉しそうに頷いて、すぐに立ち上がって黒いマーカーを手に取った。

「それらがモナド則を満たすものをモナドと呼ぶ。モナドは……直観的にはモノイドのような構造をもつもの、と思ってもいい。2 つのプログラム P1 と P2 があって都合の良い型であれば、我々は P1 を実行してから P2 を実行する、ということができる。この、続けて実行する、という操作でプログラムはモノイドのように扱える」

マーカーはたくさん用意されていたが、それらは当然ランダムに使われ、一部は捨てられて新品が導入されを繰り返したせいで残りのインクがまちまちになっている。彼女はしゃべりながら何本かのマーカーを試し、その中で一番新品に近そうなものを選んだ。私はそんな彼女をふふふと見つめていた。

「モナド則は次の 3 つからなるね」

```
- (f >>= g) >>= h == f >>= (\x -> g x >>= h)
- return x >>= k == k x
- m >>= return == m
```

「うーん、見たことはある、気がする」

正直な所見たことがあるかどうかすら記憶にはないのだけれど。

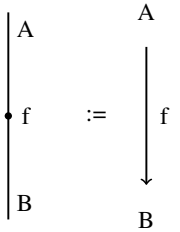
「そのモナド則っていうの、あんまりわかったような気がしないけど、知らなくてもとりあえず Haskell でプログラムは書けるよ」

「そういうと思ったから今日は話をしにきたの」

彼女は楽しそうに言って、マーカーをひょいひょいと回して持ち直した。

「絵を描きましょう」

「右のような、A から B への関数を左のような紐と点で表すことにしましょう」



「へぇ」

「これを使って、関数を図に起こすことを考える」

彼女が私を見つめた。私が何かを言うのを待っているようだった。

「つまり、モナドを図にするってこと？」

「そう」

そして、先ほどと同じような図を書いていく。

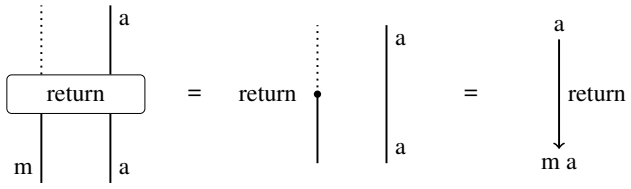


図 1.1: return

「return の定義を思い出して。a -> m a というのは、1 a -> m a と思えば 2 本の紐からなるものだと思うことができるね」

「左の紐は 1 -> m で、右の紐は a -> a の型を持つ。ここで、1 はしばしば省略されるので、点線で書いてみた」

3 つのイコールで繋がれた図を見た。

第2章

矢澤にこ先輩と一緒にモナドモナド！

— @public_ai000ya

2.1 Haskell プロのニコニーよ

こんにちは、みんなのアイドル、ニコニーよ
μ's のみんなからは　Haskell オタクとかよく言われてるわ
今日はモナド上のモナドこと MMonad を理解することを目指すわよ
にこも MMonad 周りは詳しくないから、一緒にやっていきましょ
にこも頑張って解説するわ

ええ、よろしくね！
MMonad は mmorph パッケージ[*1]に定義されているわ
ここでは便宜上、圏論におけるモナドを『モナド』、Haskell における型クラス Monad を『Monad』、同じく圏論におけるモナド上のモナドを『モナドモナド』[*2]、Haskell における型クラス MMonad を『MMonad』として話していくわね
例えばこんな感じね……『Haskell の Monad インスタンスはちょうど圏 Hask のモナドになるわ』

2.1.1 ショック死しないで欲しいニコ

本稿についての注意点があるニコ！
まず本稿は、土台となる知識をセルフコンテインしていないわ
つまり、さらっと圏論の知識や Haskell の知識が出てきたりするけど、それについて本稿での解説はないの……多岐に渡り過ぎちゃうからさ
でもまあ……想定される知識は、関手, 型クラスくらいよ
ごめんねっ！ はい、にっこにっこにー♪

2.2 モナドモナドってなによ？

MMonad って何かっていうと……モナドの圏の上に構築されるモナドよ
最初は圏論での意味で確認した方が簡単だから、そっちから見ていきましょうか
まず何かが対象の普通の圏 C があるわね、ここで一般的なモナドを考えましょう。そう、ここのモナドは圏 C の自己関手よ
そして圏 C のモナドが対象、そのモナドの自然変換のうち、ある条件を満たすものが射の圏を考えられるわ。その圏を圏 Monads と呼びましょう
この『ある条件を満たす、モナド間の自然変換』をモナドモフィズム（monad-morphism）とい

[*1] https://www.stackage.org/haddock/lts-7.7/mmorph-1.0.6/Control-Monad-Morph.html
[*2] 一般的には『モナドモナド』なんて呼び方はしないらしいわ。モナドモナドってすごい名前ね……ぷーくすくす ww

うの
そこで普通のモナドと同じように……圏 Monads 上のモナドも考えられるわね
それがつまりモナドモナドよ！（ドドーン！！）
……
アバウトがわかったところで、じゃあ始めていきましょう！

2.3 解説の流れ

じゃあ本題の MMonad について、始めましょう
以下の流れで解説していくわね

- 各章
  1. モナドモフィズムってなによ！
  2. MFunctor はモナドファンクタにこ
  3. MonadTrans は特性が弱まった MMonad にこよ
  4. モナモナする MMonad

Haskell のモナドモフィズムは、ある法則を満たす関数として。MFunctor, MonadTrans, MMonad は型クラスとして表現されるわ

2.4 モナドモフィズムってなによ！

さて、Haskell 上のモナドモフィズムとは何かを定義していくわね
Haskell の単相型が対象、単相的な関数が射の圏を圏 Hask とするわ
圏 Hask の定義よ。割とよく見る一般的な圏 Hask と同じものね

圏	対象	射
Hask	単相型	単相的な関数

Hask の対象の例	Hask の射の例
`‘Int‘ , ‘Char‘`	`f :: Int -> Char`
`‘Maybe Int‘ , ‘Maybe [Int]‘`	`kf :: Int -> Maybe Char`
`‘Identity Int‘ , ‘State Int Char‘`	`Identity :: Char -> Identity Char`
	`id :: Int -> Int`
	`id :: Char -> Char`

ここでは例えば `id :: a -> a` は Hask の射ではなくて、単相化された `id :: Int -> Int` や `id :: Char -> Char`、`id :: Bool -> Bool` は Hask の射よ
同じく `Maybe a` も Hask の対象ではなくて、その単相化された型が対象ね
あとは……ここの射 `Identity :: Char -> Identity Char` は値構築子よ。値構築子も単なる関数にすぎないのよ
圏 Hask では、ちょうど Monad インスタンス（Monad を実装したデータ型）になる Maybe や Identity、State Int がモナドになるわね
ここで新しい圏 HighHask を考えてみるわ
圏 HighHask では、Monad インスタンスの型（種`* -> *`を持つモナド型）が対象、型 `Monad m => m a -> n a` を持ち、ある法則を満たす関数が射よ
圏 HighHask は、Haskell で表現できる圏 Monads って感じね

第3章

Coqダンジョン: 底抜けの壺の夢

— @master_q

3.1 迷路 - 命題の証明

ううーん。Coq の宿題 [*1] がぜんぜん解けないじゃなイカ…… [*2]

```
$ uname -a
Linux casper 3.14-1-amd64 #1 SMP Debian 3.14.12-1 (2014-07-11) x86_64 GNU/Linux
$ coqtop
Welcome to Coq 8.4pl3 (January 2014)

Coq < Theorem plus_0_r_secondtry : forall n:nat, n + 0 = n.
1 subgoal

  ============================
   forall n : nat, n + 0 = n

plus_0_r_secondtry < Proof.
--snip--

plus_0_r_secondtry < intros n.
1 subgoal

  n : nat
  ============================
   n + 0 = n
```

とりあえず n について場合分けしようじゃなイカ。

```
plus_0_r_secondtry < destruct n as [| n'].
2 subgoals

  ============================
   0 + 0 = 0
```

[*1] ソフトウェアの基礎 / Basics_J / 帰納法 http://proofcafe.org/sf-beta/Basics_J.html#id8

[*2] 本記事は 2014 年ごろ書いた古い記事になります。最新の Coq 8.6 では動作しません。

```
subgoal 2 is:
 S n' + 0 = S n'
```

このケースは自明でゲソ。最初のサブゴールは簡単に消せるじゃなイカ。

```
plus_0_r_secondtry < reflexivity.
1 subgoal

  n' : nat
  ============================
   S n' + 0 = S n'
```

ど、どうすればいいんでゲソ。n' が自然数という仮定しか使えないじゃなイカ。どうやら迷路の袋小路に辿り着いてしまったでゲソ。なんでこんな簡単な命題なのになんでこんなに苦労しなければならないんでゲソ。Coq の宿題は人間には解けてもワシには無理でゲソ!

3.2 アイテムのしくみ - タクティックのしくみ

Coq はタクティックの組み合わせで、証明の迷路を解いていくのでゲソが、その組み合わせ戦略は Coq の利用者が考えなければならないでゲソ。つまりタクティックがどのような機能を持っているのか、正確に把握することが重要なのかもしれないでゲソ。そういえば Coq の授業で最初に intro タクティックを習ったでゲソ。でも未だにこのタクティックが何をするのかさっぱりわからんでゲソ。まずは Coq のマニュアル [*3] で intro タクティックの仕様を確認するでゲッソ。

8.3.1 intro

This tactic applies to a goal that is either a product or starts with a let binder. If the goal is a product, the tactic implements the " Lam" rule given in Section 4.21. If the goal starts with a let binder, then the tactic implements a mix of the " Let" and " Conv".

If the current goal is a dependent product ∀ x:T, U (resp let x:=t in U) then intro puts x:T (resp x:=t) in the local context. The new subgoal is U.

If the goal is a non-dependent product T → U, then it puts in the local context either Hn:T (if T is of type Set or Prop) or Xn:T (if the type of T is Type). The optional index n is such that Hn or Xn is a fresh identifier. In both cases, the new subgoal is U.

If the goal is neither a product nor starting with a let definition, the tactic intro applies the tactic red until the tactic intro can be applied or the goal is not reducible.

うーん。厳密な規則は "Chapter 4 Calculus of Inductive Constructions" の知識がないと理解できないでゲソが、要はゴールにある product を分解するもののようでゲソ。なんだかわかったような、わからんような、でゲッソ。

そういえば Coq は LGPL-2.1 でソースコードが公開されているフリーソフトウェアでゲソ。使い方がわからないプログラムはソースコードを読んで使い方を知るのがワシらの習わしでゲソ。さっそく intro タクティックのソースコードを読んでみようじゃなイカ!

[*3] http://coq.inria.fr/refman/

第4章

IST(Internal Set Theory)入門(後編)

— @dif_engine

目標と概要

超準解析 (nonstandard analysis) を行うための理論的な枠組みとしてエドワード・ネルソンにより創始された IST(internal set theory, 内的集合論) について解説します．ネルソンの論文は丁寧に書かれていますが，読者が述語論理や公理的集合論にある程度慣れていないと読むのは難しいと思われます．この記事の目標は，そのあたりの予備知識を補いつつ IST の入り口まで読者を案内することです．

前編では，述語論理の簡単な導入を行い、ZFC 集合論の公理を紹介しました．この後編では，いよいよ本論に入ります．なお，前後編に別れてしまったお詫びも兼ね，前編がお手元にない読者の便宜を図り付録 4.A(p.62) に ZFC の公理をまとめておきました．

前編のあらすじ

紅魔館の住人,《わたし》ことパチュリー・ノーレッジは，アリス・マーガトロイドと彼女の奇妙な人形に心をかき乱されつつも，退屈な日常を切り裂く一条の陽光たる霧雨魔理沙の久しぶりの訪問に心を浮き立たせていた．「突然遊びに来て E. Nelson の internal set theory を教えて欲しいだなんて，やっぱり魔理沙ってわたしのこと好きなのかも!?」あり得ない事と知りながらの妄想も恋心のなせるわざ．魔理沙にいいところを見せるチャンスとばかりに張り切り，IST の基礎となっている ZFC 集合論とそれによる数学の形式化などを説明するのだった．

4.1 IST：内的集合論

4.1.1 IST の言語

少し難しい話を続けていたわたしたちは，咲夜さんが差し入れてくれたサンドイッチをつまみながらお茶を飲んで休憩することにした．ふと、机の上のミニチュアテーブルに目をやると、わたしたち——魔理沙，アリス，そしてわたしの三人——を模して作られた人形たちまで小さなサンドイッチを食べる素振りをしていた．いかなる秘術によるものなのか，アリスの手になるこれらの人形には意志や感情まで備わっているとのことだった．軽い食事を終えたわたしたちは食器を取り除け，ふたたびテーブルにノートを広げた．

「論理式を形式的に扱う話と，公理的集合論 ZFC の公理の紹介が終わったので，いよいよは IST の紹介を始めましょう」

「よーし，いよいよ本論だね！」魔理沙が応ずる．

「さて，ZFC がそうであったように，IST も一階の述語論理で公理化されるわ．まず ZFC 集合論の言語 $\mathscr{L}_{\mathrm{ZFC}}$ がどんなものだったかを復習のため確認しましょう」:

- ($\mathscr{L}_{\mathrm{ZFC}}$ の定数記号の集合)$\mathsf{Const}^{\mathrm{ZFC}} = \{\boxed{\varnothing}\}$.

- ($\mathscr{L}_{\mathrm{ZFC}}$ の述語記号の集合)$\mathsf{Pred}^{\mathrm{ZFC}} = \mathsf{Pred}_2^{\mathrm{ZFC}} = \{\boxed{\in}\}$.

IST の言語 $\mathscr{L}_{\mathrm{IST}}$ の言語は $\mathscr{L}_{\mathrm{ZFC}}$ を拡張したものになっているわ：

- ($\mathscr{L}_{\mathrm{IST}}$ の定数記号の集合)$\mathsf{Const}^{\mathrm{IST}} = \{\boxed{\varnothing}\}$.
- ($\mathscr{L}_{\mathrm{IST}}$ の述語記号の集合)$\mathsf{Pred}^{\mathrm{IST}} = \mathsf{Pred}_1^{\mathrm{IST}} \cup \mathsf{Pred}_2^{\mathrm{IST}}$,
  ただし $\mathsf{Pred}_1^{\mathrm{IST}} = \{\mathrm{st}()\}$, $\mathsf{Pred}_2^{\mathrm{IST}} = \{\boxed{\in}\}$.

「んーと，$\mathscr{L}_{\mathrm{IST}}$ で追加されたのは st() という 1 引数の述語だけだね」
「IST の項の集合 $\mathsf{Term}_{\mathrm{IST}}$ と ZFC の項の集合 $\mathsf{Term}_{\mathrm{ZFC}}$ は《変数の集合》Var を用いて”同じ形”で定義されるわ：

$$\mathsf{Term}_{\mathrm{ZFC}} := \mathsf{Var} \cup \mathsf{Const}^{\mathrm{ZFC}}. \tag{4.1}$$

$$\mathsf{Term}_{\mathrm{IST}} := \mathsf{Var} \cup \mathsf{Const}^{\mathrm{IST}}. \tag{4.2}$$

——そして IST の論理式の集合 $\mathsf{Form}_{\mathrm{IST}}$ は次のようになるわ：

$$\mathsf{Form}_{\mathrm{IST}} := \bigcup_{k\geq 0} \mathscr{F}_k, \tag{4.3}$$

ただし

$$\begin{aligned}
\mathscr{F}_0 &:= \left\{\langle \boxed{=}, t, u\rangle \,\middle|\, t,u \in \mathsf{Term}_{\mathrm{IST}}\right\} \\
&\qquad \cup \left\{\mathrm{st}(t) \,\middle|\, t \in \mathsf{Term}_{\mathrm{IST}}\right\} \cup \left\{\langle \boxed{\in}, u, v\rangle \,\middle|\, u,v \in \mathsf{Term}_{\mathrm{IST}}\right\}, \qquad (4.4)\\
\mathscr{F}_{k+1} &= \mathscr{F}_k \cup \left\{\langle \boxed{\forall}, x, A\rangle \,\middle|\, x \in \mathsf{Var}, A \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\exists}, x, A\rangle \,\middle|\, x \in \mathsf{Var}, A \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\neg}, A\rangle \,\middle|\, A \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\wedge}, A, B\rangle \,\middle|\, A,B \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\vee}, A, B\rangle \,\middle|\, A,B \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\Rightarrow}, A, B\rangle \,\middle|\, A,B \in \mathscr{F}_k\right\} \\
&\qquad \cup \left\{\langle \boxed{\Leftrightarrow}, A, B\rangle \,\middle|\, A,B \in \mathscr{F}_k\right\} \qquad (k \geq 0). \qquad (4.5)
\end{aligned}$$

復習すると，ZFC の論理式の集合 $\mathsf{Form}_{\mathrm{ZFC}}$ は次のようになるのだったわね：

$$\mathsf{Form}_{\mathrm{ZFC}} := \bigcup_{k\geq 0} \mathscr{E}_k, \tag{4.6}$$

第5章

静的コード解析はいいぞ！

— @master_q

5.1 今日も不具合の海を泳ぐ現実……

皆の衆! 元気にはすはすプログラミングしているでゲソ？ 良いことじゃなイカ。でも現実を観察してみると、様々な不具合が設計時や出荷後に発生して悩む自分がいることに気がつくと思うでゲソ。型の強い言語でさえ不具合が出てしまうならば、C 言語のような危険な言語における不具合の量たるや想像できないじゃなイカ。しかも厳しいことに、OS のような生のハードウェア上で動くソフトウェアや OpenSSL [*1]などの基盤ミドルウェアでは、依然として C 言語による設計を人類は強いられているでゲソ。このような低レベルソフトウェアに対する不具合の低減や信頼性向上は、人力によるレビューや動的なテストなどで担保されているのが今日の現実じゃなイカ。

ちょっと落ち着いて「不具合とはいったい何なのか」考察してみるでゲソ。不具合とは

- 「**コンパイル時**に検出される不具合」
- 「**ソフトウェアの実行時**に検出される不具合」

の 2 つに大別されるんじゃイカ？ C 言語を例に取ると、

- 前者の例としては、存在しない関数の呼び出し
- 後者の例としては、不正なポインタを辿ってメモリを読むこと (デリファレンス)

などが挙げられるでゲソ。

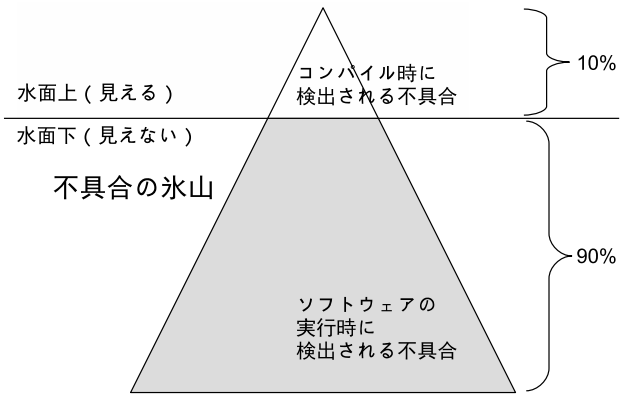


図 5.1: 氷山の一角

[*1] https://www.openssl.org/

不具合の特性は、よく氷山にたとえられるでゲソ (図 5.1)。

「コンパイル時に検出される不具合」は、海面上に見えている氷山の 10% の部分でゲソ。それは目に見えるので、開発者は簡単に回避することができるじゃなイカ。しかもこの不具合は設計の早い段階で発見できて、網羅的に潰すことができるでゲソ。ここで不具合が検出できて修正できれば、その後のデバッグ工数や出荷後の不具合を抑制することができるじゃなイカ。

しかし、海面の上に見えている部分は全体の 10% 程度にすぎないため、その海面下には残りの 90% もの氷山が隠れているじゃなイカ! 海面下に眠っている目に見えない 90% の部分、これこそが「ソフトウェアの実行時に検出される不具合」でゲソ。この不具合は実際にソフトウェアを実行してみないことには知覚することができないじゃなイカ。この不具合の知覚のために、デバッグやテストの工数を人類は払っているでゲソ。ところがもちろんデバッグやテストをしても、不具合を網羅的に潰すことは原理的に困難じゃなイカ……。極寒の海における航海において、この見えない氷山のために多くの船が座礁してしまったでゲソ。

5.2　静的コード解析は夢いっぱい！

そこで人類は 1 つのアイデアを思いついたのでゲソ。「コンパイル時に検出できる不具合」を補うために、プログラミング言語のコンパイラとは別に、「検証器」をコンパイル時に使って、プログラムを実行することなく (ある条件下においては) 網羅的に解析することで、コンパイラによるエラーよりもさらに幅広い種類のエラーを検証時に見つければ良いのではなイカ、と (図 5.2)。

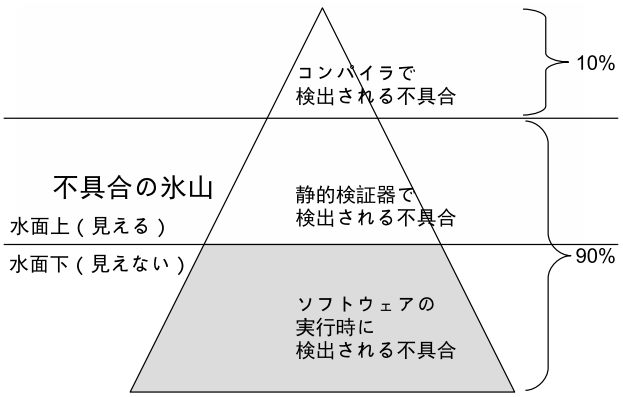


図 5.2: 静的検証でより多くの不具合を見えるように

こうすれば、コンパイラだけでは 10% の不具合しか目に見えなかったでゲソが、この割合を増やし、90% もの見えなかった不具合の一部を検証によって発見できるようになるのでゲソ! つまり、検証器は「コンパイル時に検出できる不具合」を増やすために無理矢理海面を押しさげて、より多くの不具合を網羅的に見える化してくれる魔法のツールなんでゲソ! さすが人類、敵ながらあっぱれじゃなイカ。

このようなプログラムをコンパイル時に検証する手法の 1 つが「**静的コード解析** (*Static program analysis*) [*2]」でゲソ。

[*2] https://en.wikipedia.org/wiki/Static_program_analysis

第6章

VeriFastチュートリアル

— 著: Bart Jacobs / 訳: @eldesh, @master_q

この記事は “The VeriFast Program Verifier: A Tutorial” [*1] の翻訳でゲソ。この文書の著者である Bart Jacobs [*2] に翻訳を出版しても良いか尋ねてみたところ...

> Q: ”In Japan, many people are excited by your VeriFast. And also they would like to get a paper book of your VeriFast tutorial in Japanese.”
> Bart: ”Wow, that’s great to hear! :-)”
> Q: ”May I publish Japanese translation of your VeriFast tutorial as paper books, and get some money from the book?”
> Bart: ”Yes, definitely!”

なんと快諾してくれたじゃなイカ! ということで世界初の VeriFast の解説記事が、なぜか日本語で出版されることとなったでゲソ。またこの翻訳はオンライン[*3] でも読むことができるでゲソ。

6.1 導入

VeriFast [*4][*5] はシングルスレッドやマルチスレッドの C 言語プログラム (VeriFast は Java もサポートしています; *VeriFast for Java: A Tutorial*[*6] を読んでください) の性質が正しいことを検証するプログラム検証ツールです。このツールは、1 つ以上の .c ソースコードファイル (さらにそれらの .c ファイルから参照されている .h ヘッダファイル) から成る C 言語プログラムを読み、「0 errors found (エラーが見つからなかった)」とレポートするかエラーの可能性がある位置を示します。もしこのツールが「0 errors found」とレポートしたなら、そのプログラムは次のようであることを意味しています[*7]:

- 構造体インスタンスが解放された後にその構造体のフィールドを読み書きすることや、もしく

[*1] https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf
[*2] https://distrinet.cs.kuleuven.be/people/bartj
[*3] https://github.com/jverifast-ug/translate/blob/master/Manual/Tutorial/Tutorial.md
[*4] https://people.cs.kuleuven.be/~bart.jacobs/verifast/
[*5] https://github.com/verifast/verifast
[*6] https://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast-java-tutorial.pdf
[*7] このツールが時々間違って「0 errors found」とレポートしてしまう (**不健全性** (*unsoundnesses*) と呼ばれる) いくつかの理由があります; https://github.com/verifast/verifast の soundness.md を参照してください。さらに知られていない不健全性もあるでしょう

は配列の終端を超えた読み書き (これは **バッファオーバフロー**と呼ばれ、オペレーティングシステムやインターネットサービスにおけるセキュリティ脆弱性の最も多い原因です) のような、不正なメモリアクセスを行ないません。なおかつ

- データレース、すなわちマルチスレッドによる同じフィールドへの非同期な競合アクセス、として知られたある種の並行性のエラーを含みません。なおかつ
- 関数は、そのソースコード中の特殊なコメント (**注釈** (*annotations*) と呼ばれます) でプログラマによって指示された、事前条件と事後条件に従っています。

不法なメモリアクセスやデータレースのような C 言語プログラムにおける多くのエラーは、テストやコードレビューのようなこれまでの手法では検出することが一般的にとても困難です。なぜなら、それらはしばしば潜在的で、通常はわかりやすい故障を引き起こさず、そのくせ診断するのが困難な予測できない作用を持つからです。けれども、オペレーティングシステム、デバイスドライバ、(電子商取引やインターネットバンキングを扱う) Web サーバ、自動車に対する組み込みソフトウェア、航空機、宇宙関連、原子力発電所や化学プラントなどのような、多くのセキュリティと安全性が要求されたプログラムは C 言語で書かれます。そしてこれらのプログラミングエラーはサイバー攻撃や損傷を可能にするのです。そのようなプログラムにとって、VeriFast のような形式的な検証器によるアプローチは、要求されたレベルの信頼性を達成するもっとも効果的な方法になりえます。

全てのエラーを検出するために、VeriFast はプログラムに対して**モジュラーシンボリック実行** (*modular symbolic execution*) を行ないます。特に、VeriFast はプログラムのそれぞれの関数本体をシンボリックに実行します。関数の事前条件によって表現された **シンボリック状態** (*symbolic state*) から開始し、文によってアクセスされたそれぞれのメモリ位置に対してシンボリック状態中に**パーミッション** (*permissions*) が存在するかチェックし、それぞれの文の作用を考慮するためにシンボリック状態を更新し、そして関数が返るときに最終的なシンボリック状態が関数の事後条件を満たすことをチェックします。シンボリック状態は、あるメモリ位置へのアクセスに対する複数の (**チャンク** (*chunks*) とも呼ばれる) パーミッションを含む**シンボリックヒープ** (*symbolic heap*) と、それぞれのローカル変数へのシンボリック値 (symbolic value) を代入する**シンボリックストア** (*symbolic store*)、そして現時点の実行パスのシンボリック状態で使われている**シンボル** (*symbols*) の値に関する **仮定** (*assumptions*) の集合である **パスコンディション** (*path condition*) から構成されます。シンボリック実行は常に停止します。なぜならループ不変条件の使用のおかげで、それぞれのループ本体はたった一度だけシンボリックに実行され、関数呼び出しのシンボリック実行ではその関数の事前条件と事後条件だけを使い、その本体は使用しないからです。

筆者らは現在ツールの機能を少しずつ作成している最中です。このチュートリアルにある例や練習問題を試してみたい方は、次の VeriFast ウェブサイトから VeriFast をダウンロードしてください:

- `https://people.cs.kuleuven.be/~bart.jacobs/verifast/` [*8]

`bin` ディレクトリに、コマンドラインのツール (`verifast.exe`) と GUI のツール (`vfide.exe`) が見つかるでしょう。

6.2 例: illegal_access.c

どのように VeriFast を使用してテストやコードレビューで発見することが困難なプログラミングエラーを検出できるか説明するために、捉えにくいエラーを含むとても単純な C 言語プログラムに

[*8] 本記事執筆時点ではこのリンクでは古い安定板のみを配布しています。最新版の VeriFast を使用するには `https://github.com/verifast/verifast#binaries` からダウンロードしてください。