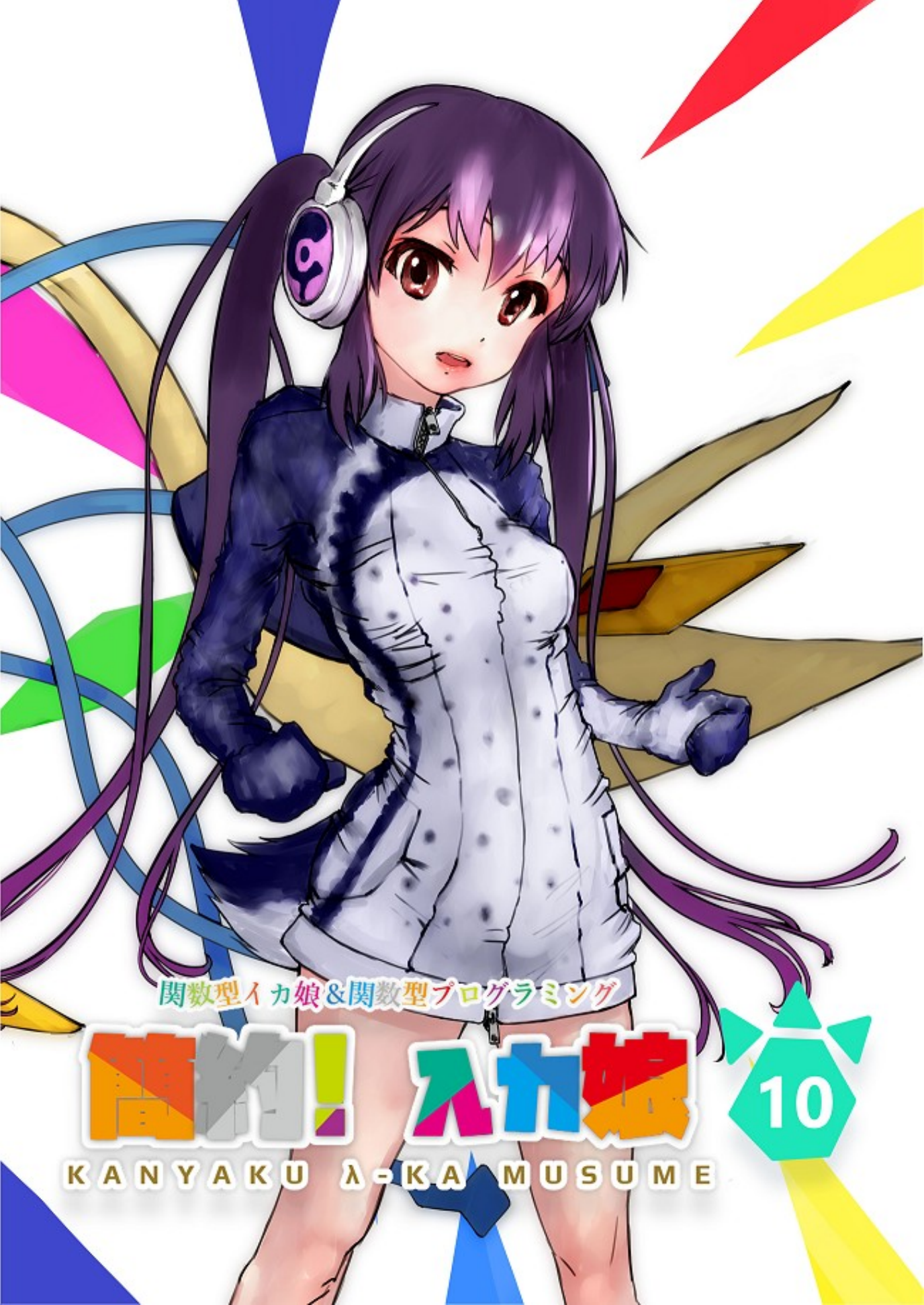

関数型イカ娘&関数型プログラミング
関約! イカ娘
10
KANYAKU λ-KA MUSUME

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。

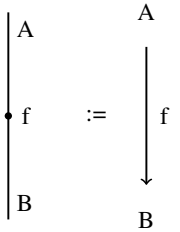
「そのモナド則っていうの、あんまりわかったような気がしないけど、知らなくてもとりあえず Haskell でプログラムは書けるよ」

「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。

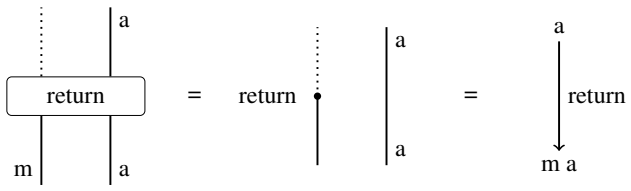


図 1.1: return

「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。

「それってさ、m と a の紐が横に並んでたら m a と読む、という意味で理解していいのかな」
「それでいい。一般に紐の端点のラベルは順に適用されてると思えばいい。ただし 3 本以上ある場合は基本的に右結合と考えて」
「じゃあ m と a と b の紐が並んでたら m (a b) の意味ってことだね」
彼女は頷いて、再びマーカーを手元でくるっと回して持ち直した。
「さて、bind に移ろう。bind は型 (a -> m b) -> m a -> m b を持つ関数だった。これは引数 k:a -> m b を固定してしまえば、bind は k を bind k に変換する操作だと思うことができる」
「k:a -> m b も、また 1 つの紐になると思っていいんだよね？ k と bind(k) がどちらも紐だから、bind は紐を受け取って別の紐に変えるってことかな？」
「そう。確認だけど、k:a -> m b はどんな図になるか分かる？」
私は return の図を再度見た。return:a -> m a は上に向かって 1 と a、下に向かって m と a の紐が出ているような図だった。だとしたら、k:a -> m b も同じはずだ。
「a から m と b が出てる箱みたいなのになるのかな？」
「そう」
彼女は満足そうにふふと笑って、先程の図の下に新たな図を描いた。

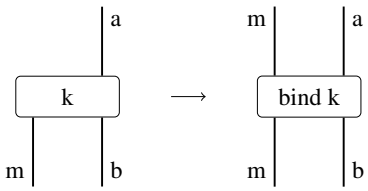
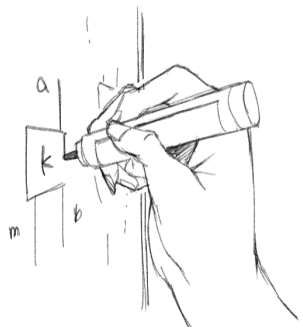


図 1.2: bind

「bind は左の図を右の図に変える機構、あるいは操作だと思えばいい。どう？」
「うん。大丈夫」
そして彼女はまたしても、マーカーをくるくるくると三度ほど回してから、少しだけホワイトボードに描かれた図を見つめた。こういう感じで考えごとをしている時、この人には話しかけてはいけないのだというのが、私の彼女の友人としての了解だった。
「気がついた」
「うん？ どうかしたの？」
「polymorphism の話をしていなかった」



1.2　さいだー

彼女は少し休憩と言って、カバンからサイダーを取り出して飲んでいた。私が知る限り彼女は水とサイダーしか飲まない。他の飲み物を飲んでいるところを見たことがない程度にはサイダージャンキーだった。

そんなに大量に飲むのは健康に悪いんじゃなかろうかと思いながら私も買っておいたペットボトルのお茶を飲んだ。美味しい。

「Haskell の、parametric polymorphism は知ってるよね」

「えっ」

私がなにそれ、という顔をすると彼女は薄目でこちらを睨んできた。いや、私は悪くないと思うんだけど。

「そんなの知らないけど、Haskell でプログラムを書くのに困ったことはな」

「わかった。私が悪かった」

降参、というポーズを取られて私は少しだけムッと来た。先程のホワイトボードがくるりと回って裏面がこちらを向いた。

「`f:a -> a` という関数が定義できたとする。このとき、この `a` とは何か？」

「別になんでもいいんでしょ」

「そう、それが polymorphism。つまり、この関数はどんな型の値も受け取ることができる」

彼女は空白のホワイトボードに `forall a.a -> a` と書いた。

「`a` は何でも良い、ということをこうして `forall` というキーワードをつけて明示することにする」

「うん」

「この `a` は代入されるまではどんな型になるかは分からない。しかしひとたび代入されれば、この関数は引数に与えられた型と同じ型を返すことが分かるね。このような、`forall a.` のついた関数を、`a` についての polymorphic function という」

なんだ、そんなことか。簡単じゃない。という言葉が危うく口をついて出そうになったけれど、また睨まれそうだったので私は何も言わずに頷いた。

「例を挙げよう。先程の、`return:a -> m a` は `a` についての polymorphic function だ。`a` に何が来てもいいからね。このことを、さっきの図に戻って考える」

「先程は、敢えて 2 本の紐で `return` を表現した。`1 -> m` と `a -> a` だ。しかし、この右側の `a` は、どんなものでもよかったことを思い出して」

「そうね」

「というわけで、この右側の紐は省略することにしよう。`return` というのは 1 本の紐で、それは `1 -> m` という形をしている」

「そして、実際に値を適用する時になって、新たな紐を加えることにする。例えば、`return {Int}:Int -> m Int` としたいときは、右に新たに `Int -> Int` の紐を付け足せばいい」

「それは `a` の型を決めるってことだね」

「そう。次に、`bind` を見てみよう。`bind` も polymorphic function だ。これは `bind:forall a b.(a -> m b) -> (m a -> m b)` だ。」

「うん」

「けれど、さっきも見たように、`bind` は図を別の図に変換する機構だった。そして、さっき考えた `k:a -> m b` だけど、こいつは `a` と `b` については polymorphic じゃない」

「あれ、そうなの？」

「`k` についてよく見て。こいつはどんな関数？」

「`bind` に渡される引数だよね。もし、`k` が polymorphic なら、`k` はどんな値も受け取れるはずだけ

ど、別にそうじゃなくていいってことかな。例えば `f:Int -> m Int` みたいな関数は `Int` しか引数に取れないけど、`bind f` はちゃんと型が合うもんね」

「そう。だから `k` の右側の紐は省略できない。これは `a` も `b` も polymorphic じゃないからね」

「……」

「どうかしたの？」

彼女はまた少しの間、虚空を見つめていた。

「準備が整った」

1.3 かいだん

「モナド則に戻ろう」

「モナド則の 1 つ目は、`(f >>= g) >>= h == f >>= (\x -> g x >>= h)` だった」

彼女はまた先程も書いた式を改めてホワイトボードに書いてくれた。

「これをさっきみたいな図にしよう。今私達は、`>>=`の引数の順番を変えた `bind` の図をすでに知っている。この式がどういう図になるかは、もう分かるよね？」

そう言って彼女はニッコリと私の方へマーカーを手渡してきた。私は少しうろたえてから、それを受け取ってホワイトボードへ向き直った。

「まず、左辺から行くね」

「うむ」

「まず 3 つの関数をおいちゃうね。`f` はモナド値で、`g,h` はモナド値を返す関数だから、それぞれ `m a, a -> m b, b -> m c` としましょう」

「いいね」

「`(f >>= g) >>= h` は `bind h ((bind g) f)` になるから、それを順に図にすればいいかな」

私はマーカーのキャップをパカッと開けて箱を書き始めた。下から `bind h`, `bind g`, `f` を並べて、そこから出る線で、同じ型を持つものを繋いでいく。同様に右辺の式も図を描いていく。

`f`, `bind g`, `bind h` が連なった階段のような図ができた。上から `m a` の値を渡して順に `m b`, `m c` に変換されていくという図だ。

「これでいいかな」

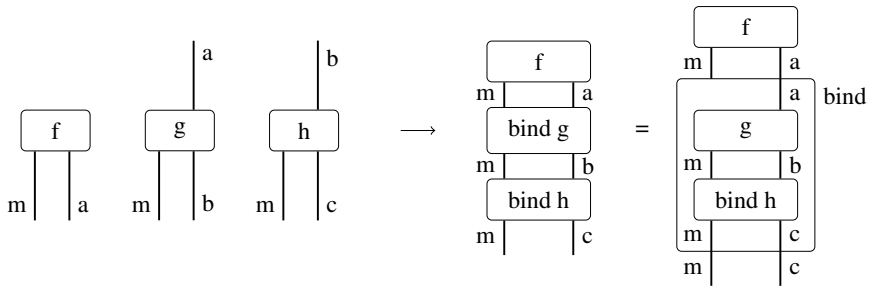


図 1.3: `(f >>= g) >>= h == f >>= (\x -> g x >>= h)` の図

「そうだね。これ、どう思う？」
「どうって何よ……まぁ、`m a` が渡されて、`m c` に変換されるのはわかるよね。それ以上に意味があるのかは知らないけれど」
「それだけ分かれば十分かな。さて、この式にはもっと別の意味もある。次は `join` について考えよう」
「待って、待って。あんたが何を目指してるのかは知らないけど、私には突然いろんなものが出てきてよくわかんないよ」
「ふむ」
彼女はまたくるくるとペン回しをしながら私を不思議そうな顔で見た。そしてしばらく考え込んでから、口を開いた。
「モナドの定義は `return` と `bind` によるもの、と思ってるかもしれないけれど、実は `return` と `join` でも定義ができる。今は前者の方法でモナド則を見たけれど、後者の方法でもモナド則を見てみよう、という話」
「ほうほう」
「`join` は `bind` とは違って、さっき話した polymorphism が効いてくるからまた見方が変わってくると思うよ」
「なるほどねぇ」
「では `join` に進もう」

1.4　ごうりゅう

「一応訊くけど、`join` は知っている？」
「さすがに知ってるよ。`join :: m (m a) -> m a` でしょ」
「そ。しかもこれは polymorphic function だ」
「そういえばそうだね。`a` は何でも良いからね」
「そこで、`join` の図は右側の紐を再び省略することができる」
彼女はホワイトボードを一旦全て消した。私は椅子に座り直した。図書館内部は人がちらほら出入りしていたが、私たちが入ってきてから密度はほとんど変わっていない。
「図にしよう」
彼女はまたしても図をかいた。

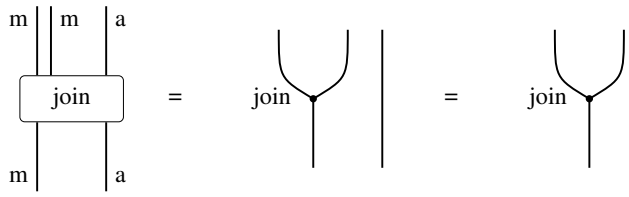


図 1.4: join

「いいね？」
「`join` は 2 つの紐が 1 点で合流するような図になるんだね。……って、ああそれで `join` て呼ぶのか」
「これは `m . m -> m` という形の紐だと思えばいい。さて、これを使って先程の `bind` と `join` が互いに変換可能であることをみよう」
「つまり、`bind` から `join` が定義できるし、`join` から `bind` も定義できる。こんな感じ」

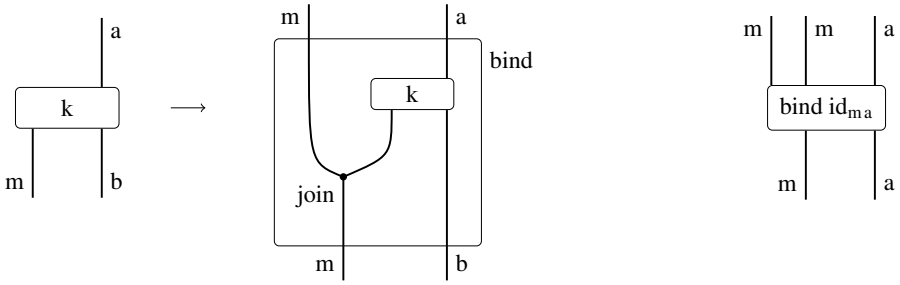


図 1.5: bind ↔ join

「左側が bind。bind は図を別の図に移す必要があったのを思い出して。右側が join だけど、これは bind id だから簡単ね」

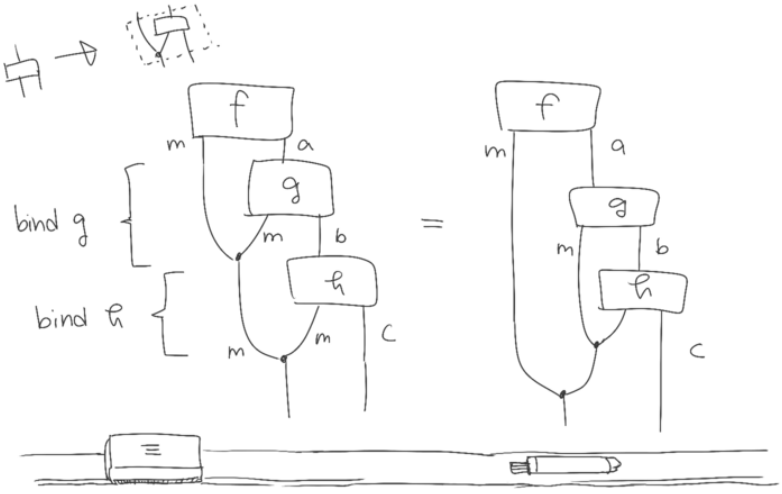
「うん。これを使ってさっきの図を書き直すのね」

「そう。bind の部分を join で書き直すだけだから簡単でしょ」

彼女は私へマーカーを渡してきたので、私はそれを受け取ってホワイトボードの前に立った。

「じゃあ、展開してみるね」

私は彼女の言うとおりに、bind で描かれた部分を先程の join を使った形に変形してゆく。縦にスペースを取る都合で何度も書き直し、ようやくホワイトボードのおよそ半分を使って巨大な図をかき上げた。



「こうかな」

「そうね。で、f,g,h は任意だった」

「確かに。じゃあ、f,g,h の出てくる部分の図は共通してるから省いてしまって、残った部分だけ

比較すればいいね」

私はごちゃごちゃした余計な部分を消して、本当に必要だった部分を残した。

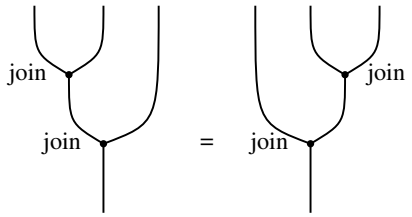


図 1.6: 本当に必要だった associativity

「そう、これが associativity。モナド則の 1 つ目だね」

「つまり、3 本の紐を繋げるときはどっちを先に繋げても一緒ってことだね」

「そう」

「うん、なんとなく、スッキリした気がするよ」

1.5　ひとつめ

「これで、話は終わり？」

「まさか」

彼女は不敵な笑みを浮かべた。

「今のがモナド則の 1 つ目。あとの 2 つ目と 3 つ目もまた同じように図を描くことができる。こっちは identity law が出てくるよ」

「identity law ねぇ」

「少なくとも、さっきの associativity が理解できた君なら問題なく導出もできるよ」

少し私を買いかぶり過ぎじゃあないか？

「こういう図のことを String Diagram という」

「String Diagram……紐の、図、ってことかな」

「そう。String Diagram では先程導いたような紐の操作を通して色々なものの計算を行うことができる……ということを、私達は見たと思う。部分的に、だけど」

「さっきやったのは 3 本の紐の繋ぎ方を変えても同じ、ってやつだったね」

「そうね」

「さぁ、雑談はこのくらいにして」

彼女は挑戦的な笑みを浮かべてこちらへマーカーを差し出した。その両目の奥が一瞬キラリと光ったような気がした。

「まっかせといて。残りの 2 つのモナドの法則も、私が華麗に導いてみせよう」

私はマーカーを受け取る。

1.6　おわりに

時間的な都合でえらく中途半端な内容になってしまったので筆者よりいくつか補足をします。

上でも述べましたが、まず string diagram を使うにあたって使いたい構造を表現することは難しいことが多いです。具体的には、`bind` を (図の変換ではなく) そのまま図に表すのはかなり工夫が必要です。例えば diagram に使っていた “紐” を “管” のように太さのあるものとして扱う方法や、3 次元方向へ diagram を拡張し Z 方向の重なりを直積を取る操作とみなす方法などです。しかしいずれにせよ、polymorphism などの良い性質がなければ紐を取り除く、などの直観的には許される操作

が許されなくなる可能性があるので注意が必要です。具体的なデータ構造について操作が直観的に図として表現できることは多くはありませんので、少なくとも今回はそれを避けました。

本来ならば adjoint から monad を構成する話なども添える予定でした。string diagram は基本的に自然変換の図示をするのに便利なので、こういうテーマに限っては非常によく働きます。(しかし、今更ですが圏論の予備知識を仮定しないでこういう話をするのは厳しかったかも)

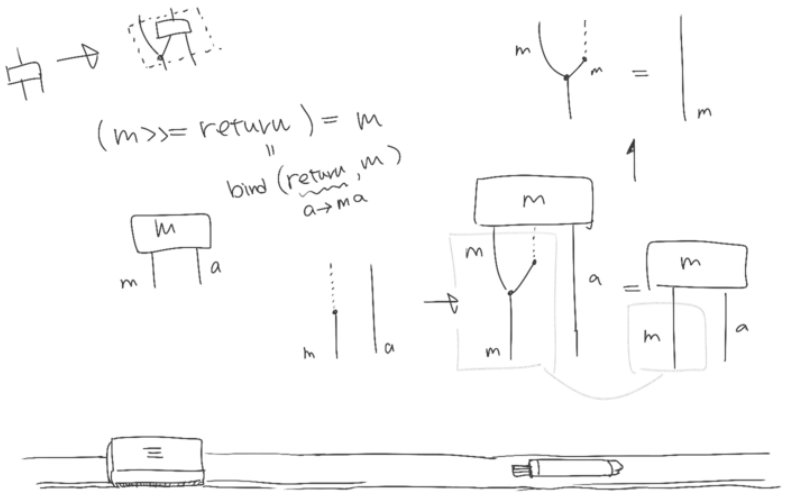
また、string diagram にはいくつかの流派があるので文献を読まれる際には気をつけてください。今回は上から下、左から右に向かって読むタイプの図式を採用しましたが、上下および左右は流派によって異なります。加えて、今回は 2-category の図式を表現するために diagram を使いましたが、monoidal structure を表現するために同じような string diagram を使う場合もあり、そちらの場合は紐の重なりなどが全く違う意味を持ちます。今回の diagram の書き方はここだけの記法であることをご理解ください。

最後に、ぜひ興味を持たれた方は、身の回りの関数やデータ構造について図に表してみてください。思わぬ隠れた構造を発見することができるかもしれません。

1.7 参考文献

- 絵算のススメ 2015 年末版 (檜山正幸のキマイラ飼育記)[*1]: String Diagram の紹介記事です。色々な流派の比較などがされています。
- String diagram のすすめ (PS)[*2]: Haskell の関数を String Diagram で表現する話。curry 化などについても言及があります。
- Category Theory Using String Diagrams (Daniel Marsden; 2014)[*3]: モナドや T 代数を String Diagram で表現する話。今回の記事の次に読むのにちょうどよいと思うので、深く知りたい方は是非。
- String diagrams, adjunctions and monads (TheCatsters channel)[*4]: String Diagram を解説する Youtube の動画です。

[*1] http://d.hatena.ne.jp/m-hiyama/20151120/1447986292
[*2] http://mbps.hatenablog.com/entry/2015/12/20/070000
[*3] https://arxiv.org/abs/1401.7220
[*4] https://www.youtube.com/playlist?list=PL50ABC4792BD0A086


(m >>= return) = m
=
bind (return, m)
a → m a

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

圏	対象	射
Hask	単相型	単相的な関数
HighHask（New!）	Monad インスタンスの高階型	ある法則を満たす m a -> n a

どう？ わかるかしら。この射がモナドとモナドの変換、モナドモフィズム[*3]よ

HighHask の対象の例	HighHask の射（モナドモフィズム）の例
`‘Maybe‘ , ‘Identity‘` `‘State Int‘ , ‘State [Char]‘`	`identityToJust :: Identity -> Maybe` `id :: Maybe -> Maybe`

圏 HighHask の対象 `Maybe` や `Identity` は、Haskell では `Maybe a`, `Identity a` として
同じく圏 HighHask の射 `identityToJust` の型は、Haskell では `Identity a -> Maybe a` として表れるわ

`State Int` と `State [Int]` は同じ `State a` モナドじゃないのかって？ いいえ、この 2 つは別のモノとして扱われるの
`State Int a -> State [Char] a` も `State Int a -> State Char a` もモナドモフィズムね♪（ニコッ）
じゃあそろそろ、モナドモフィズムの満たすべき法則について見てみましょう！

2.4.1 文脈届けるにこにこ♪

```
morph :: Monad m => m a -> n a
```

あるモナドモフィズム `morph` は、2 つの法則を満たすわ

- 法則 A

```
morph (return x) = return x
```

- 法則 B

```
morph $ do
  x <- m
  f x
=
do
  x <- morph m
  morph $ f x
```

何を言っているのか全然わからないって？
そうねえ、大丈夫よ。そのために解説役のにこがいるニコッ♪
まずは簡単な法則 A から考えましょ

[*3] この射が `(Monad m, Monad n) => m a -> n a` じゃないのは、`mmorph`[^1] での定義がそうだからよ。でも `(Monad m, Monad n) => m a -> n a` として考えてもらっても構わないわ

2.4.2　return を保存するニコ！

法則 A を難しくしているのは、異なる型の return が 2 つあるからよ
明示的に型付けしましょうか

```
x        :: a
return'  :: Monad m => a -> m a
return'' :: Monad n => a -> n a
```

こうすると、こう導けるわね

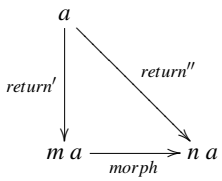
```
morph          :: (Monad m, Monad n) => m a -> n a
morph . return :: Monad n => a -> n a

morph . return' = return''
-- = morph (return' x) = return'' x
```

return は Monad インスタンス固有の特性を起こさず、そのまま a を Monad の文脈に入れる役割を持つ関数だったわね

ここでは特性という言葉を……例えば State Int が Int の状態を参照, 変更できること、Maybe が Nothing という特別な値を扱えること、IO が全ての副作用を扱えること……を指すわ

そして morph には自然な実装が要求されるニコ！

それは以下の図式を可換にするわっ



（圏 Hask のある対象を a
ある Monad m 及び Monad n の対象関数で a を写したもの（対象）を m a
n a と表記するニコよっ）

m の文脈で return したら m を n に変換しても、n の文脈で return しても、同じ結果になるってことね♪

例えば Identity の return の実装を以下のものに差し替えちゃったら、これが可換にならなくて不自然ってことになるわね

```
return :: a -> Identity a
return = Identity . (+1)
```

でも安心してね。この例は a を Int として扱ってるから、Haskell ではコンパイルで弾かれるわよ
（逆に、Haskell はそうすることによって自然性を守ってるとも言えるわね）

2.4.3 副作用も保存するニコ！

今度は難しい方の法則をやるわよ

こっちも変形してから考えましょ

```
morph (m >>= f) = morph m >>= morph . f
```

こっちも型を可視化するわ

```
bindM :: Monad m => m a -> (a -> m b) -> m b
bindM = (>>=)
bindN :: Monad n => n c -> (c -> n d) -> n d
bindN = (>>=)

morph (m `bindM` f) = morph m `bindN` morph . f
```

f が副作用を起こす関数だとするわ

そうした場合に、m の文脈で副作用を起こしても n の文脈で副作用を起こしても、結果が同じになる

……ってことよ！ ねっ、言い方が難しいだけで、言ってることは簡単なのよ♪

2.4.4 法則にこ？

ところで、気づいたかしら？ Haskell の型クラス Monoid、Functor や Monad などでも法則を満たすことを要求されたわよね

型クラスの関数の実装について要求されたわよね、例えば `Functor` だと……`fmap id = id` みたいに、`fmap` について要求されるわ

でも今回、モナドモフィズム一般について要求されているの。つまり、あるモナドモフィズムを実装する毎に確認する必要があるわ

ここは注意が必要ね

あ、でもね。あとでまた教えるけど、モナドモフィズムを受け取る関数、`hoist` や `embed` がランク 2 多相なモナドモフィズムを要求しているおかげで、それについてあまり考える必要はないのよ

`hoist` や `embed` にモナドモフィズムを渡そうと思うと、自ずと自然なモナドモフィズムを作ることになるわ

ふぅ、これでやっとモナドモフィズムが理解できたわね。やったー！！（ニコー！）一番大きな山を超えたにこ♪ よしよし、あなたも頑張ったわね！

ここらでお茶でも飲んで休憩しましょ（チーズバーガーモグモグよ！）

2.5 MFunctor はモナドファンクタにこ

さて、ここからは実際に提供されている型クラスについてやっていくわ

まずは `MFunctor` ね。さっきやったモナドモフィズムをそのまま扱う型クラスよ

といってもモナドモフィズムよりはまだ簡単だと思う。あまり気を張らないでいいと思うニコよ♡

そう、これ自体……はね

2.5.1 射関数 fmap, hoist

MFunctor は以下のような型クラスよ

```
class MFunctor t where
  hoist :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
```

MFunctor は、前述した圏 HighHask のファンクタになるの
わかりやすく言うと、『モナドファンクタ』ね
Functor と比較してみましょうか

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

Functor の型変数 f が、MFunctor の型変数 t に対応しているのが、ひと目でわかるわね
じゃあ、fmap と hoist がどのように対応するか……っていうと、これらのファンクタの射関数としての側面を見ると完全にわかるニコよ

```
func  :: a -> b  -- 圏 Hask の射 func : a -> b
morph :: Monad m => m a -> n a
  -- 圏 HighHask の射（圏 Hask のモナドモフィズム）morph : m -> n
```

```
fmap func   :: f a -> f b      -- (a -> b) を (f a -> f b) に写した
hoist morph :: t m b -> t n b  -- (m -> n) を (t m -> t n) に写した
```

圏論のレベルで見れば、2 つとも同じものってことがわかるわね

2.5.2 Haskell での表現

同じものっていっても、Haskell が圏 Hask 相当のレベルで見ている以上、表現がズレちゃうの
具体的には Haskell では Monad m => m -> n っていう記法はできないわ
そのズレをうまく表現するために、hoist はランク 2 多相（Rank2Types）を使っているニコ！

```
                                  ランク 2 多相
hoist :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
```

ランク 2 多相がわからない？ 大丈夫、まだ私も完全にはわかってないから……
ここで forall a. が何をしているかわかればいいわ
この forall a. はつまるところ、m -> n を間接的に表現しているの
m -> n はモナドモフィズムよ
モナドモフィズムは前述の通り『m と n の return を可換にする（return を保存する）』という意味

で、自然な実装にする必要があったわ

不自然性については return の時と同じで……例えば、このモナドモフィズムは自然だけど

```
naturalMMorph :: State Int a -> Reader Int a
naturalMMorph st = do
  let nakedSt = runState st
  reader $ fst . nakedSt
```

このモナドモフィズムは不自然よ！

```
unnaturalMMorph :: State Int Int -> Reader Int Int
unnaturalMMorph st = do
  let nakedSt = runState st
  reader $ (+1) . fst . nakedSt
```

理由は明白で、unnaturalMMorph . return = return を満たさないからよ！

こういう不自然さは m a -> n a の a に言及しているからできることで、hoist の定義を

```
hoist :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
```

というようにすることで、引数のモナドモフィズムが a に言及できないようにしているわ！

hoist unnaturalMMorph をコンパイルしようとすると、a が Int という絞られた型になっていることにより、コンパイルエラーが出るわ

この forall a. については、こちら[*4]がとてもわかりやすかったわ。にこはここを読んで理解したニコ！

このようにして、MFunctor をモナドファンクタに仕立て上げているわ♪

2.6 MonadTrans は特性が弱まった MMonad にこよ

ちょっと疲れてきたかしら？ モナドモナドまでもうもう一息よ！

だめ？ 元気でない？……じゃあ、にこのスマイルあげるわ！ はい♪ にっこにっこにー ♡ にっこにっこにー ♡ にっこにっこにー ♡♡

元気でたわよね？ ねっ！ さあやっていきましょう♪

さてこの MonadTrans だけど、実用上使うことも多い、よく知られたあの MonadTrans と全く同じものよ

実は MonadTrans は、MMonad の特性を弱めたものなの

具体的には、MMonad のうち embed と、MFunctor の hoist についての法則が要求されない

なのでモナドモフィズムも登場しないわ

なんで特性を弱める必要があったのかって？

そ、それは、ある資料[*5]によると ContT が MMonad にはなれないからしいわよ

その……ごめんね。実は、にこは Cont も ContT も使ったことなくって……

こ、こほん！

[*4] http://qiita.com/YoshikuniJujo/items/c28d8fa11e33ed677e83

[*5] http://qiita.com/hiratara/items/65fcf38070def7e5a918

そう、つまり MonadTrans にあるのは `lift :: (MonadTrans t, Monad m) => m a -> t m a` なのよ

2.6.1 lift はモナドモナドの return ニコ！

この `lift` は Monad を MonadTrans に引き上げる関数よ。MMonad については後で扱うけど、MMonad も MonadTrans なので、モナドモナドの `return` に相当するわ

意味的には `lift :: m -> t m` ね。Haskell の `return :: a -> m a` に似てるでしょ。高いレベルから見れば同じものよ

`lift` は以下の法則を要請するわ

```
lift :: (MonadTrans t, Monad m) => m a -> t m a

lift . return = return
lift (m >>= f) = lift m >>= lift . f
```

これ、見たことないかしら？ ちょっと考えてみて……

そう、モナドモフィズムに要請される法則と同じものなの！

```
morph :: (Monad m, Monad n) => m a -> n a

morph . return :: Monad n => a -> n a
morph (m >>= f) = morph m >>= morph . f
```

実は `lift` も、ちょうどモナドモフィズムになっているの

モナドモフィズムは `Monad m => m a -> n a` のような、Monad インスタンスから Monad インスタンスへの変換だったわね

さっき言った通り、`(MonadTrans t, Monad m) => t m` な型はモナドになるの。だから、`lift` もモナドモフィズムよ

MonadTrans インスタンスである StateT を例に、型を当てはめてみましょう

```
instance Monad m => MonadTrans (StateT s m)
-- 簡単にするために s と m を単相化
lift :: IO a -> StateT Int IO a
(>>=) :: a -> StateT Int IO b
```

`(>>=)` は Monad の代表的な演算子だし、ちょうど `(>>=) :: Monad m => a -> m b` の `m` が `StateT Int IO` になっていることがわかりやすいと思うんだけど、どうかしら

2.6.2 lift はこうやって使うわ

最後に、ついでに `lift` のオーソドックスな使用例を見てみましょう

```
import Control.Monad.IO.Class (MonadIO)
import Control.Monad.Trans.State.Lazy (StateT, runStateT)
import Control.Monad.Trans.Class (lift)
```

```
f :: StateT Int IO Char
f = do
  -- lift :: IO () -> StateT Int IO ()
  lift $ putStrLn "in the context of StateT Int IO"
  return 'a'

main :: IO ()
main = do
  x <- runStateT f 10
  print x

{- output -}
-- in the context of StateT Int IO
-- ('a',10)
```

MonadTrans は、最も自然なモナドモフィズム lift を持つ型クラスってところね
そう考えてもらうと、MonadTrans が MMonad を弱めたものだっていうのが、なんとなくわかるかもしれないわ♪

2.7 モナモナする MMonad

……ついにたどり着いた、にこたちの目的地。モナモナするモナド、MMonad よ！
長いようで短かったあなたとの勉強会もこれで終わりね。楽しかったわよ
でもまだこれを理解するまで終わらないの！ やっていくわよ

2.7.1 モナドのモナド

さて、MMonad の定義は以下になるわ

```
class (MFunctor t, MonadTrans t) => MMonad t where
  embed :: Monad n => (forall a. m a -> t n a) -> t m b -> t n b
```

MMonad は MFunctor であって MonadTrans よ。モナドが自己関手であって MonadTrans が MMonad の部分であることを鑑みれば、自然な感じね
embed は圏 HighHask のモナドバインドよ。(>>=) と並べて見てみましょうか

```
embed      :: Monad n => (forall a. m a -> t n a) -> t m b -> t n b
flip embed :: Monad n => t m b -> (forall a. m a -> t n a) -> t n b
(>>=)      :: Monad m => m a -> (a -> m b) -> m b
```

高い視点の疑似表記で見てみるわ

```
flip embed :: t m -> (m -> t n) -> t n
```

```
(>>=) :: m a -> (a -> m b) -> m b
```

うん。ちゃんと一致してるわね

2.7.2　モナモナする法則ニコ

そしてやっぱり、embed は法則を満たす必要があるわ。これらは Monad 則と全く同じものよ

- 法則 A

```
embed lift = id
```

- 法則 B

```
embed f (lift m) = f m
```

- 法則 C

```
embed g (embed f t) = embed (\m -> embed g (f m)) t
```

flip embed のエイリアスとして (|>=) が用意されているわ。これを使って変換してみましょう。

```
(|>=) :: (MMonad t, Monad n) => t m b -> (forall a. m a -> t n a) -> t n b
```

```
-- A
t |>= lift  =  t
-- B
lift m |>= f  =  f m
-- C
(t |>= f) |>= g  =  t |>= (\m -> f m |>= g)
```

こう書けば、そのまま Monad 則に対応するわね

```
-- A'
m >>= return  =  m
-- B'
return a >>= k  =  k a
-- C'
(m >>= k) >>= h  =  m >>= (\x -> k x >>= h)
```

2.7.3　にことあなたとそしてモナドモナド

……これで MMonad の性質の確認も終わっちゃったけど

……これで終わり。じゃあつまんないわよね！

これで本当に最後よ

一緒に mmorph に書いてある ‘Embedding transformers’ の章を読みましょ……！

これは実用的で、すごく面白い例よ。にこも初めて見た時、すごく興奮したものよ

(MonadThrow m, MonadIO m) => m a のような型を持つ文脈で IO 例外が投げられた時は、普通はそのまま MonadThrow 値にならずに不純な例外、IO 例外として上位に伝播しちゃうわよね

でも embed と補助関数を使えば、IO 例外を MonadThrow 値の純粋な例外として受け取れるの！

例えば (MonadThrow m, MonadIO m) => m a で、f :: EitherT SomeException IO Int に対して x <- runEitherT f することで Left e にパターンマッチできるわ！

こほん、ごめんなさいね。興奮して一気に喋りすぎたニコ！

つまるところね、IO 例外が Left e にマッピングできるのよ……！ すごーい！！

```
import Control.Exception.Safe (MonadThrow, try, SomeException)
import Control.Monad (join)
import Control.Monad.IO.Class (MonadIO, liftIO)
import Control.Monad.Morph (MMonad, embed)
import Control.Monad.Trans.Class (lift)
import Control.Monad.Trans.Either (EitherT(..), runEitherT)

instance MMonad (EitherT e) where
  embed f m = EitherT $ do
    let nee = runEitherT . f $ runEitherT m -- :: n (Either e (Either e a))
    join <$> nee

check :: IO a -> EitherT SomeException IO a
check = EitherT . try

program :: (MonadThrow m, MonadIO m) => m String
program = liftIO $ readFile "foo"

main :: IO ()
main = do
  xOrErr <- runEitherT $ embed check program
  case xOrErr of
    Right x -> putStrLn $ "Success: " ++ x
    Left  y -> putStrLn $ "Left value is got !: " ++ show (y :: SomeException)

-- output
-- Left value is got !: foo: openFile: does not exist (No such file or directory)
```

ここに注目してちょうだい。本質はここだけよ

```
xOrErr <- runEitherT $ embed check program
```

embed check program の check で IO 例外を MonadThrow 例外にすくい上げてるわ

ここで見て欲しいのは、↓の例もそれと実質的にはほとんど同じであるということ

```
xOrErr <- runEitherT program
```

でも program で foo ファイルの読み込みが失敗した場合に IO 例外が送出されて、下の case 文が実行されなくなってしまう

その問題に対する最もシンプルな解決策としては、program 内の readFile に try を引っ掛けることが考えられるわよね

```
import Control.Exception.Safe (MonadThrow, try, SomeException, throwM, Exception)

-- てきとうに投げられる例外の定義
data AnException = AnException String
  deriving (Show)

instance Exception AnException

program :: (MonadThrow m, MonadIO m) => m String
program = do
  foo <- liftIO $ try' $ readFile "foo"
  case foo of
    Left  e -> throwM $ AnException (show e)
    Right a -> return a
  where
    -- e を SomeException に推論させる
    try' :: IO String -> IO (Either SomeException String)
    try' = try

main :: IO ()
main = do
  xOrErr <- runEitherT $ embed check program
  case xOrErr of
    Right x -> putStrLn $ "Success: " ++ x
    -- IO 例外もここにマッチする
    Left  y -> putStrLn $ "Left value is got !: " ++ show (y :: SomeException)
```

でもこの解法、いくつかの文句をつけることができちゃうわよね

『program の本質を損ねている』『program 本体に変更を加える、しかしそれはナンセンスだ』なんて

そういう場面で embed が『引っ掛けて』くれるのよ！

そんな解法が存在するの、なんだか Haskell らしくないかしら？ らしいわよね♪

こんなところに MMonad が使えるなんて……ねっ！

2.8 もう終わりなのね

ふう
というところで……この企画、『矢澤にこ先輩と一緒にモナドモナド！』は終わりニコ
MMonad の理解に必要な項目は全て消費し終わったからね
あなたはこの企画により、以下の項目を理解したかもしれないし、まだ理解できていないかもしれないわね

- mmorph における
  - モナドモフィズム
  - MFunctor
  - MonadTrans
  - MMonad

ふふ。少し寂しいわね……にこもあなたに教えるの、楽しかったわ
おかげで私も MMonad とか mmorph 周りの理解が進んだし、その……ありがと！
あのね、あなたと私の、この話はここで……おしまいだけど
もし、本紙の向こう側にいるあなたがまた私を呼んでくれるなら、読んでくれるなら。私はいつでもここにいるわ！
それにもしかしたらまた……別の世界のにこが他の何かについて、あなたと話す機会があるかもしれないわよね
だからもしよかったら……またね、ニコッ ♡

2.9 参考にさせてもらった資料よ

（敬称略にて失礼するニコ！）

- Monad Morphism による局所的状態の表現（香川考司/京都大学数理解析研究所）https://projects.repo.nii.ac.jp/?action=pages_view_main&active_action=repository_view_main_item_detail&item_id=287642&item_no=1&page_id=13&block_id=21
- ランク 2 多相の、ふたつの側面（YoshikuniJujo）
  http://qiita.com/YoshikuniJujo/items/c28d8fa11e33ed677e83
- モナドモナド (LT 没ネタ)（hiratara）
  http://qiita.com/hiratara/items/65fcf38070def7e5a918

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/

3.3 アイテムの作り方 - 自作タクティック

intro タクティックのソースコードを読んでみたら、ワシもタクティックを作ってみたくなったでゲソ。Coq をワシ好みに改造すれば、ひょっとしたら宿題が楽に解けるかもしれないじゃなイカ。まずは単純なタクティックを作って Coq から使えるようにしてみなイカ？ 例題としては単に intro と同じことをする別名のタクティックを作ってみるでゲソ。

```
$ mkdir -p coqtactic-injectfalse
$ cd coqtactic-injectfalse
$ vi injectfalse.ml4
let ij_intro = Tactics.intro

TACTIC EXTEND _ij_intro_
| [ "ij_intro" ] -> [ ij_intro ]
END
```

この camlp4 のソースコードをコンパイルするには coq_makefile コマンド使えばよいようでゲソ。

```
$ coq_makefile -R . INJECTFALSE_tactics injectfalse.ml4 -o Makefile.coq
$ make -f Makefile.coq
$ ls injectfalse.*
injectfalse.cmi  injectfalse.cmxs  injectfalse.o
injectfalse.cmo  injectfalse.ml4
```

バイナリができたようでゲソ。さっそく Coq から使ってみようじゃなイカ。

```
$ coqtop
Welcome to Coq 8.4pl3 (January 2014)

Coq < Declare ML Module "injectfalse".
[Loading ML file injectfalse.cmxs ... done]

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

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

plus_0_r_secondtry < ij_intro.
1 subgoal

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

作った ij_intro タクティックは intro タクティックと同じ動きをしているでゲソ。簡単じゃなイカ!

3.4 底抜けの壺 - 仮定に False を無条件に追加

そういえば「矛盾した前提からはどんなものでも導き出せる」という話[*4][*5]を思い出したでゲソ。Coq の前提条件に False を無条件に追加するタクティックを作れたら、ひょっとしてどんなサブゴールでも消せるんじゃなイカ？ これは宿題がはかどる凄いアイデアじゃなイカ。さっそく実装するでゲソー!

```
$ make -f Makefile.coq clean
$ vi injectfalse.ml4
open Evd
open Coqlib

let ij_injectfalse goal_sigma =
  let gl = goal_sigma.it in
  let s' = Goal.V82.partial_solution goal_sigma.sigma gl (build_coq_False ())
  in {it=[gl]; sigma = s'}

TACTIC EXTEND _ij_injectfalse_
| [ "ij_injectfalse" ] -> [ ij_injectfalse ]
END
$ coq_makefile -R . INJECTFALSE_tactics injectfalse.ml4 -o Makefile.coq
$ make -f Makefile.coq
```

なんと 3 行で書けた上、OCaml のコンパイルも無事通ったでゲソ。これはイケるんじゃなイカ!? 明日から宿題はすぐ終わるでゲソー。

3.5 一時しのぎの杖 - 宿題をかっとばせ

さっき解けなかった宿題がこれで楽勝のはずでゲソ。さっさと終わらせてアイスをおいしく食べるでゲソ。

```
Coq < Declare ML Module "injectfalse".
[Loading ML file injectfalse.cmxs ... done]

Coq < Theorem plus_0_r_secondtry : forall n:nat, n + 0 = n.
--snip--
plus_0_r_secondtry < Proof.
--snip--
plus_0_r_secondtry < intros n.
--snip--
plus_0_r_secondtry < destruct n as [| n'].
```

[*4] Principle of explosion - Wikipedia https://en.wikipedia.org/wiki/Principle_of_explosion

[*5] 「源義経の母親はナポレオン」Twitter で話題の激ムズ論理学の問題を数学科の大学院生が説明するよ （吾輩の辞書に「論理学」がない（悲壮)。） http://nlab.itmedia.co.jp/nl/articles/1407/17/news058.html

```
2 subgoals

  ============================
   0 + 0 = 0

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

サブゴールが 2 つあるので、2 つともワシの作った `ij_injectfalse` タクティックで殲滅でゲソ!

```
plus_0_r_secondtry < ij_injectfalse.
1 subgoal

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

plus_0_r_secondtry < ij_injectfalse.
No more subgoals.
```

あっさり全てのサブゴールを消せたでゲソ。気持ちいいじゃなイカ! さぁこれで証明を終わりにするでゲッソ!

```
plus_0_r_secondtry < Qed.
intros n.
destruct n as [| n'].
 ij_injectfalse.

 ij_injectfalse.

Error: In pattern-matching on term "n" the branch for constructor
"0" has type "Prop" which should be "0 + 0 = 0".
```

……え……どうしてエラーになるんでゲソ……証明は終わったはず…… (グニャ)

3.6 何が起きたの？

結局、ワシはどんな宿題でも解ける魔法のタクティックを OCaml コードで作ることには成功したでゲソ。そしてそのタクティックは OCaml コンパイラの検査は通すことができ、Coq から使うことが可能だったでゲソ。ところが `Qed.` と入力して、ワシの自作タクティックで作った証明を Coq に検査してもらうと型検査エラーになったでじゃなイカ。

これはワシに都合の良い、本来は証明上の不具合の混入したタクティックである `ij_injectfalse` の証明上における不具合を、Coq は `Qed.` と構築した証明を検査しようとした段階で検出してくれたということでゲソ。この検査は Coq の証明器としての健全さを示すものじゃなイカ。素晴しいでゲソ!

欲を言えばこのようなタクティックの不具合は、`Qed.` による証明の検査の瞬間ではなく、タクティックの OCaml コードをコンパイルする段階で検出して欲しいでゲソ。しかしどうやら「任意

のタクティックが正しく命題を加工しうるか」を検査するのはまだ人類にとって難題なようじゃないカ。がんばれ人類! っでゲッソ!

3.7 参考文献

- 簡潔な Q - Coq の tactic を OCaml で書く話[*6]
- 本記事で作成したタクティックのソースコード[*7]
- 風来のシレン - 底抜けの壺[*8]

[*6] http://qnighy.hatenablog.com/entry/2013/01/15/004411
[*7] https://github.com/master-q/coqtactic-injectfalse
[*8] http://twist.jpn.org/sfcsiren/index.php?**底抜けの壺**

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：

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

ただし

$$\mathscr{E}_0 := \left\{ \langle \boxed{=}, t, u \rangle \,\middle|\, t,u \in \mathsf{Term}_{\mathrm{IST}} \right\} \cup \left\{ \langle \boxed{\in}, u, v \rangle \,\middle|\, u,v \in \mathsf{Term}_{\mathrm{IST}} \right\}, \tag{4.7}$$

$$\begin{aligned}
\mathscr{E}_{k+1} = \mathscr{E}_k &\cup \left\{ \langle \boxed{\forall}, x, A \rangle \,\middle|\, x \in \mathsf{Var}, A \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\exists}, x, A \rangle \,\middle|\, x \in \mathsf{Var}, A \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\neg}, A \rangle \,\middle|\, A \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\wedge}, A, B \rangle \,\middle|\, A,B \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\vee}, A, B \rangle \,\middle|\, A,B \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\Rightarrow}, A, B \rangle \,\middle|\, A,B \in \mathscr{E}_k \right\} \\
&\cup \left\{ \langle \boxed{\Leftrightarrow}, A, B \rangle \,\middle|\, A,B \in \mathscr{E}_k \right\} \qquad (k \geq 0).
\end{aligned} \tag{4.8}$$

「そういえばそうだったね」魔理沙がコメントする.
「論理式の略記についても remind しておきましょう（表 4.1）」:

定義通りの書式	略式の記法
$\langle \boxed{=}, t, u \rangle$	$t = u$
$\langle p, u_1, \ldots, u_n \rangle$	$p(u_1, ..., u_n)$
$\langle \boxed{\forall}, x, A \rangle$	$\forall x A$
$\langle \boxed{\exists}, x, A \rangle$	$\exists x A$
$\langle \boxed{\neg}, A \rangle$	$\neg A$
$\langle \boxed{\wedge}, A, B \rangle$	$A \wedge B$
$\langle \boxed{\vee}, A, B \rangle$	$A \vee B$
$\langle \boxed{\Rightarrow}, A, B \rangle$	$A \implies B$
$\langle \boxed{\Leftrightarrow}, A, B \rangle$	$A \iff B$

表 4.1: 論理式の定義通りの書式と略式記法

「そういえばそういう約束だったね」魔理沙が言った.

4.1.2 述語 st() について

「ZFC において $x \in y$ であるとき《x が y の元である》とかさらにくだけて《x が y に入っている》と口頭で言ったりしてたわね．論理式を使って表記するのと同等な口語的表現を用意しておくと議論しやすいので，述語 st() に関連した言い回しをまとめておきましょう（表 4.2）」:

論理式	口語的言い回し
$\mathrm{st}(x)$	x は標準的である
$\neg\mathrm{st}(x)$	x は超準的である
$\mathrm{st}(t) \wedge (t \in x)$	t は x の標準元である
$\neg\mathrm{st}(t) \wedge (t \in x)$	t は x の超準元である

表 4.2: 述語 st() に関連した論理式と口語的言い回し

「ところで IST に追加された st() ってのはどんな気持ちの述語なんだろう？」魔理沙が質問した.

「《x が標準的である》こと，つまり $\mathrm{st}(x)$ である事は，《x が普通の方法で具体的に構成された集合であること》という気分だと思っておくといいかも知れないわね」わたしはこう答えた．
「その《普通の方法で》とか《具体的に》ってどういうことだろう？」魔理沙が問う．
「あとで確認するように，ZFC の公理を用いて $\exists x\ldots$ という形の論理式が証明された場合，この集合 x は標準的になるわ．でもこれ以上うまく説明するのは難しいわね」
「IST で何かの集合を議論しているときに《これは標準的になりそう》とか《これは超準的だろう》という感じの直感が働くようになったら述語 st() が表している気持ちがわかったと言える．でもそれができるまでには時間がかかる」アリスが呟くように言った．
「なるほど，そんなものなのかな」魔理沙がやや不満気に言った．

4.1.3 内的論理式と外的論理式

「言語 $\mathscr{L}_{\mathrm{IST}}$ には ZFC の言語 $\mathscr{L}_{\mathrm{ZFC}}$ には含まれない述語 st() が含まれているわね．いま

1. $\exists y\ \ (x \in y) \wedge (y \in z)$.
2. $\exists y\ \ \mathrm{st}(y) \wedge (x \in y) \wedge (y \in z)$.

とすると， 論理式 1. は IST の論理式だし，ZFC の論理式ともみなせるわ．論理式 2. は IST の論理式だけど，st() が含まれているため ZFC の論理式とはみなせないわね」
「うん，確かにそうだね」魔理沙が答えた．
「IST の論理式のうち，ZFC の論理式ともみなせるものを《内的な論理式》と呼ぶことにするわ．IST の内的な論理式の集合を IntForm としましょう．形式的に IntForm を定義するならば，要するに IST の論理式の定義の過程から st() を抜いてしまえばいいのだからこうなるわね」:

$$\mathrm{IntForm} := \bigcup_{k\geq 0} \mathscr{G}_k,$$

ただし

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

「確かにこの IntForm は ZFC の論理式の集合 $\mathsf{Form}_{\mathrm{ZFC}}$ と同じになるね」魔理沙が言った．
「IST の論理式のうち，ZFC の論理式ではないものを《外的な論理式》と呼ぶわ」
「IST の外的な論理式の集合を ExtForm とすると，形式的に ExtForm を定義するならばこうなるよね」:

$$\mathrm{ExtForm} := \mathsf{Form}_{\mathrm{IST}} - \mathrm{IntForm}.$$

「そのとおりね」

4.1.4 外的量化子

「述語 st() と量化を組み合わせた外的論理式は IST でよく使われる——IST の公理図式の記述にも——ので，略語的な意味合いの表記を導入しましょう（表 4.3）」：

	省略記法	本来の論理式	意味
外的な量化	$\mathring{\forall}xA(x)$	$\forall x(\mathrm{st}(x) \implies A(x))$	任意の標準的な x に対して $A(x)$
	$\mathring{\exists}xA(x)$	$\exists x(\mathrm{st}(x) \wedge A(x))$	ある標準的な x が存在して $A(x)$
	$\mathring{\forall}^{\mathrm{Fin}}xA(x)$	$\mathring{\forall}x(\mathrm{Fin}(x) \implies A(x))$	任意の標準的な有限集合 x に対して $A(x)$
	$\mathring{\exists}^{\mathrm{Fin}}xA(x)$	$\mathring{\exists}x(\mathrm{Fin}(x) \wedge A(x))$	ある標準的な有限集合 x が存在して $A(x)$
内的な量化	$\forall^{\mathrm{Fin}}xA(x)$	$\forall x(\mathrm{Fin}(x) \implies A(x))$	任意の有限集合 x に対して $A(x)$
	$\exists^{\mathrm{Fin}}xA(x)$	$\exists x(\mathrm{Fin}(x) \wedge A(x))$	ある有限集合 x が存在して $A(x)$

表 4.3: 拡張された記法の定義とその説明

「——ここで導入された $\mathring{\forall},\mathring{\forall}^{\mathrm{Fin}},\mathring{\exists},\mathring{\exists}^{\mathrm{Fin}}$ を《外的量化子》と呼ぶことにするわ．この呼び方に合わせて，$\forall,\forall^{\mathrm{Fin}},\exists,\exists^{\mathrm{Fin}}$ は《内的量化子》と呼ぶことにするわ．念のために復習するけど，このなかに出てくる $\mathrm{Fin}(x)$ ってのは形式化された自然数の集合 $\mathbb{N}$ を使ってこんな風に定義されていたわね」：

$$\mathrm{Fin}(x) \quad \equiv \quad \exists n \in \mathbb{N}\ \exists f\colon x \xrightarrow{1\text{-}1} n. \tag{4.9}$$

アリスがコメントした——
「外的量化子は普通の限定量化と関連づければ理解しやすい」：

$$\begin{aligned} \forall x \in y \quad A(x) \qquad &\equiv \quad \forall x\big(x \in y \implies A(x)\big). \\ \exists x \in y \quad A(x) \qquad &\equiv \quad \exists x\big(x \in y \wedge A(x)\big). \end{aligned}$$

「さて《一意的な存在》を表す $\exists!$ についても，st() と組み合わせた量化子を導入しておくわ」：

$$\begin{aligned} \mathring{\exists}!x\ A(x) \qquad &\equiv \quad \Big(\mathring{\exists}x\ A(x)\Big) \wedge \mathring{\forall}x_1\mathring{\forall}x_2\,(A(x_1)\wedge A(x_2) \implies x_1 = x_2). \\ \mathring{\exists}^{\mathrm{Fin}}!x\ A(x) \qquad &\equiv \quad \Big(\mathring{\exists}^{\mathrm{Fin}}x\ A(x)\Big) \wedge \mathring{\forall}x_1\mathring{\forall}x_2\,(A(x_1)\wedge A(x_2) \implies x_1 = x_2). \end{aligned}$$

「要するに白丸を上に付けたら $\forall$ を $\mathring{\forall}$ に，$\exists$ を $\mathring{\exists}$ に置き換えればいいんだね」魔理沙が言った．
アリスがここでコメントした——
「この記事ではこのような記号を使うけど，E. Nelson の 1977 のオリジナル論文を含めて，普通は《$\mathring{\forall}$》《$\mathring{\exists}$》の代わりに《$\forall^{\mathrm{st}}$》《$\exists^{\mathrm{st}}$》と書く．記号を変えた理由，あるいは言い訳としては次の二点を挙げておく」：

1. 手でノートを書いているときに《$\forall^{\mathrm{st}}$》《$\exists^{\mathrm{st}}$》と書くのが面倒だった．
2. 《$\forall^{\mathrm{st}}$》《$\exists^{\mathrm{st}}$》は視認性が若干低い．

4.1.5 IST の公理

「IST の公理を述べましょう．IST の公理は次のものからなるわ」：

1. ZFC のすべての（公理図式ではない）公理．
2. IST の内的論理式を適用対象とする《分出公理図式》と《置換公理図式》．
3. IST のすべての論理式を適用対象とする《等号公理図式２》．
4. 次に掲げる《移行原理》《理想化原理》《標準化原理》の三つの公理図式．

移行原理 (Transfer Principle)
A を IST の内的論理式で，ある $k \geq 0$ に対して $\mathrm{FV}(A) = \{x, t_1, ..., t_k\}$ だとする．このとき：

$$\mathring{\forall} t_1 \cdots \mathring{\forall} t_k \left(\left(\mathring{\forall} x A(x, t_1, \ldots, t_k) \right) \Longrightarrow \left(\forall x A(x, t_1, \ldots, t_k) \right) \right). \tag{T}$$

理想化原理 (Idealization Principle)
B を IST の内的論理式で，ある $k \geq 0$ に対して $\mathrm{FV}(B) = \{x, y, u_1, ..., u_k\}$ だとする．このとき：

$$\forall u_1 \cdots \forall u_k \left(\left(\mathring{\forall}^{\mathrm{Fin}} Z \exists y \forall x \in Z B(x, y, u_1, \ldots, u_k) \right) \Longleftrightarrow \left(\exists y \mathring{\forall} x B(x, y, u_1, \ldots, u_k) \right) \right). \tag{I}$$

標準化原理 (Standarization Principle)
C を IST の論理式（外的論理式でも良い）で，ある $k \geq 0$ に対して $\mathrm{FV}(C) = \{z, u_1, ..., u_k\}$ だとする．このとき：

$$\forall u_1 \cdots \forall u_k \left(\mathring{\forall} x \mathring{\exists} y \mathring{\forall} z (z \in y \Longleftrightarrow z \in x \wedge C(z, u_1, \ldots, u_k)) \right). \tag{S}$$

「——それぞれの末尾に付けた (T)(I)(S) は公理の名称とするわ．今後は《移行原理により》の代わりに《公理図式 (T) により》だとか，さらに短く《(T) により》などと書いて済ませることにしましょう」わたしは言った．

4.1.6 公理をざっと眺めただけでわかること

「内的な論理式に関する限り，ZFC の全ての公理は IST でも成立しているわよね．したがって ZFC での定理は自動的に IST の定理になるわ．だから ZFC の枠組みで証明された既存の数学の結果は，何も検討せずに IST で引用できるわね」
「ZFC での定理が自動的に IST の定理になることはありがたそうだけど，これって特別なことなの？」魔理沙が質問した．
「特別，がどのような事かはさておき，たとえば超準集合論 KST[9] のように基礎の公理を欠いているものもある．だから，まったく当たり前というわけではない」アリスが言った．

4.1.7 分出公理図式や置換公理図式についての注意

「あのさ，わたしたちは古典論理で考えているんだから任意の集合 t について

(i) $\mathrm{st}(t)$
(ii) $\neg \mathrm{st}(t)$

のどちらか一方が成立していることになるんだよね？」魔理沙が言った．
「ええ，その通りね」
「じゃあさ，例えば自然数の集合 $\mathbb{N}$ の標準的なやつだけ集めた集合:

$$\left\{ t \in \mathbb{N} \,\middle|\, \mathrm{st}(t) \right\}$$

も分出公理図式を使えば作れるんだよね？」魔理沙が言う．
「それが集合になることは保証されないわ」
「えっ，だめなの？」魔理沙は驚いているようだ．
「分出公理図式によって作った $\mathbb{N}$ の部分集合は一般に

$$\left\{ t \in \mathbb{N} \,\middle|\, A(t) \right\}.$$

となるけど，ここに現れる A は内的論理式でなければならないわ」
「なるほど，st() は外的論理式だから，分出公理図式の適用範囲外ってことなんだね」魔理沙が言っ

た.
「何かの集合 x から標準的であるもの全部を取ってくることは，集合 x が特別扱いやすい場合は別だけど，一般にはできないのよ」
「なるほど，この st() って述語はちょっと扱いに注意しないといけないんだね」魔理沙が言う.
「いまは分出公理図式について取り上げたけど，置換公理図式についても同じようなことが言えるわ」
「つまり，外的論理式については置換公理図式も使えないということだね」魔理沙が言った.
「置換公理図式は《集合 x 上の関数のように振る舞う論理式》を用いて《像集合》を構成する手段を与えてくれる．ZFC 上で数学をやる上で，分出公理図式や置換公理図式は非常に便利だけれども，IST においては《外的論理式が使えない》という制約があることには注意が必要」アリスが付け加えた.

4.1.8 クラスと外集合

「ある条件を満たす要素全部を集めてくる，という発想は捨てがたい魅力があるわね．フォーマルな議論ではないけど，論理式 A から

$$\{x \,|\, A(x)\}$$

という《集合っぽい何か》を考えて，**クラス (class)** と呼ぶわ」
「集合っぽい，だから集合じゃあないんだよね」魔理沙が言った.
「そう——クラスは一般には集合ではないわ．論理式 A に対して，ある IST の集合 X があって

$$\forall x \ \ (x \in X \iff A(x))$$

となる場合は，クラス $\{x|A(x)\}$ を IST の集合 X と同一視するわ．つまり

$$\{x \,|\, A(x)\} = X$$

だと考えるわ」
「つまり，クラスってのは集合を一般化したものだと思えばよさそうだね」魔理沙が言う.
「$\{x|A(x)\} = X$ であるような集合は，外延性公理により，存在するならばただ一つになる．つまり次が成り立つ」:

$$\forall X_1 \forall X_2 \ \ \big(\big(\{x|A(x)\} = X_1\big) \wedge \big(\{x|A(x)\} = X_2\big) \implies X_1 = X_2\big).$$

「分出公理図式で存在を保証される集合を

$$\{t \in x \,|\, A(t)\}$$

と書いたわね．これを

$$\{t \,|\, (t \in x) \wedge A(t)\}$$

というクラスの略記だとみなすことにするわ．こうすれば，分出公理図式はある種類のクラスが集合となることを意味していることになるよね」
「あ，じゃあさっきの

$$\{x \in \mathbb{N} \,|\, \mathrm{st}(x)\}$$

も，《クラス》として考えてるぶんには構わないってことだね」魔理沙が言った.
「そのとおり」
「クラスを使った議論に合わせて，集合 x とクラス $Z = \{t \,|\, A(t)\}$ に対して，

$$x \in Z$$

は単に

$$A(x)$$

を意味すると約束するわね．それから，論理式 A_1, A_2 が作るクラス

$$Z_1 := \{t \,|\, A_1(t)\},\ Z_2 := \{t \,|\, A_2(t)\}$$

に対して，

$$\begin{aligned}
&Z_1 \cup Z_2 := \{t \,|\, A_1(t) \vee A_2(t)\},\\
&Z_1 \cap Z_2 := \{t \,|\, A_1(t) \wedge A_2(t)\},\\
&Z_1 - Z_2 := \{t \,|\, A_1(t) \wedge \neg A_2(t)\},\\
&Z_1 \subseteq Z_2 \equiv \forall t\ \big(t \in Z_1 \implies t \in Z_2\big),\\
&Z_1 = Z_2 \equiv \forall t\ \big(t \in Z_1 \iff t \in Z_2\big).
\end{aligned}$$

と約束しておくわね」

「なるほど，これでクラス同士の和，共通部分や差が作れることになったし，クラス同士の包含関係も定義されたわけか」魔理沙が言った．

「クラス Z_1, Z_2 が集合である場合には，$Z_1 \cup Z_2, Z_1 \cap Z_2, Z_1 - Z_2$ は集合論での操作で和，共通部分，差を作ったものと一致することは簡単に確認できるわ」アリスが言った．

「さっき魔理沙が例に挙げたクラス $\{x \in \mathbb{N} \,|\, \mathrm{st}(x)\}$ は

$$\mathbf{S} := \{x \,|\, \mathrm{st}(x)\}$$

としてやれば $\mathbb{N} \cap \mathbf{S}$ と書けるわね」

「なるほど，同じクラスを表すにも色々な書き方ができるんだね」魔理沙が言った．

「さて，集合とはならないクラスを**真クラス (proper class)** と呼ぶことにするわ」

ここでアリスが口を開いた——

「よく知られた真クラスの例はこれ：

$$\mathbf{V} := \{x \,|\, x = x\}.$$

これが真クラスであることは簡単に確かめられる．実際，仮に $\mathbf{V}$ が集合であったならば，等号公理によって $\mathbf{V} = \mathbf{V}$ が成り立ち，したがって $\mathbf{V}$ の定義から $\mathbf{V} \in \mathbf{V}$ となる．しかし，（以前確かめたように）基礎の公理からこれはあり得ない．したがって，$\mathbf{V}$ が集合であると仮定すると矛盾に導かれる．よって $\mathbf{V}$ は集合ではなく真クラスでなければならないことがわかる」

「さて，クラスに関する今までの話は ZFC と共通の話だったけど，ここから IST に固有の話題に入るわ．真クラスの中で，IST の外的論理式によって作られるものを**外集合 (external set)** と呼ぶことにしましょう」

「真クラスなのに《外集合》なの？」魔理沙が言った．

「これは伝統的な超準解析の用語を反映してこのように呼んでいるだけなのでとにかくそういうものだと思って納得して欲しいわ」

「ちなみに，E. Nelson の 1977 年の論文 [2] では，私達が《外集合》と呼んだものは《狭義外集合 (strictly external set)》と呼ばれている．原論文に当たる人はその点注意が必要．この記事で採用した外集合についての用語法は Diener[5] のものに近い」アリスが付け加えた．

「外的論理式から作られたクラスが IST の集合になってしまう場合は外集合ではないことに注意が必要ね」

「外的論理式に対応するクラスがあったとして，それが集合になるか外集合になるかってのはどうやったらわかるんだろう？　例えば $\mathbb{N} \cap \mathbf{S}$ は外集合かな？ それとも集合かな？」魔理沙が言った．

「もうすこしあとでそれは解決しましょう（p.48）」

4.2　公理図式 (T) から導かれること

4.2.1　公理図式 (T) から直ちに導かれること

「さて，公理図式 (T) について見ていきましょう．量化が沢山ついていて複雑に見えるけど，要するに大事なのは，$\mathrm{FV}(A)=\{x,t_1,...,t_k\}$ となる IST の内的論理式 A に対して

$$\mathrm{st}(t_1)\wedge\cdots\wedge\mathrm{st}(t_k)\wedge\mathring{\forall}xA(x,t_1,...,t_k)\implies\forall xA(x,t_1,...,t_k). \tag{4.10}$$

となることね．さて，正式には st() は 1 引数の述語だけど，

$$\mathrm{st}(t_1)\wedge\cdots\wedge\mathrm{st}(t_k)$$

を，以下では

$$\mathrm{st}(t_1,...,t_k)$$

と略記することにしましょう．こうすると (4.10) はこうなるわ」：

$$\mathrm{st}(t_1,\ldots,t_k)\wedge\mathring{\forall}xA(x,t_1,...,t_k)\implies\forall xA(x,t_1,...,t_k). \tag{4.10'}$$

「うーん，ちょっとだけ見やすくなったかな……？」
「公理図式 (T) の別の形はこうなるわ」：

$$\mathring{\forall}t_1\cdots\mathring{\forall}t_k\left(\left(\exists xA(x,t_1,\ldots,t_k)\right)\implies\left(\mathring{\exists}xA(x,t_1,\ldots,t_k)\right)\right). \tag{T'}$$

「えーと，なんで……？」
「その説明の前に，この記事でよく使われる（かもしれない）論理式の変形のルールをまとめておきましょう」：

$$(A\implies B)\iff(\neg B\implies\neg A), \tag{4.11}$$
$$(\neg\neg A)\iff A, \tag{4.12}$$
$$\neg(\forall xA(x))\iff\exists x\,(\neg A(x)), \tag{4.13}$$
$$\neg(\exists xA(x))\iff\forall x\,(\neg A(x)), \tag{4.13'}$$
$$\forall x\forall yA(x,y)\iff\forall y\forall xA(x,y), \tag{4.14}$$
$$\exists x\exists yA(x,y)\iff\exists y\exists xA(x,y), \tag{4.14'}$$
(B が自由変数として x を持たないとき)　$(\forall xB)\iff B$, (4.15)
(B が自由変数として x を持たないとき)　$(\exists xB)\iff B$, (4.15')
(B が自由変数として x を持たないとき)　$(\forall xA(x))\wedge B\iff\forall x(A(x)\wedge B)$. (4.16)
(B が自由変数として x を持たないとき)　$(\exists xA(x))\vee B\iff\exists x(A(x)\vee B)$. (4.16')
(B が自由変数として x を持たないとき)　$(\forall xA(x))\vee B\iff\forall x(A(x)\vee B)$. (4.17)
(B が自由変数として x を持たないとき)　$(\exists xA(x))\wedge B\iff\exists x(A(x)\wedge B)$. (4.17')

「――上に挙げたルールは，どれも論理式の意味を考えれば納得の行くものばかりよ」
「一つまたは複数の論理式から別の論理式を導く過程，すなわち《推論》を形式化すればもっと少ないルールに還元してしまうこともできる．詳しい話は論理学の教科書に譲ることにする」アリスが付け加えた．
「さて，式 (T') の証明をしましょう．$\neg A$ に公理図式 (T) を適用すると次が言えるわ：

$$\mathring{\forall}t_1\cdots\mathring{\forall}t_k\left(\left(\mathring{\forall}x\,(\neg A(x,t_1,\ldots,t_k))\right)\implies\left(\forall x\,(\neg A(x,t_1,\ldots,t_k))\right)\right). \tag{4.18}$$

これに式 (4.11) を適用して対偶の形にすると

$$\mathring{\forall}t_1 \cdots \mathring{\forall}t_k \left(\neg \Big(\forall x \, (\neg A(x,t_1,\ldots,t_k)) \Big) \implies \neg \Big(\mathring{\forall}x \, (\neg A(x,t_1,\ldots,t_k)) \Big) \right). \tag{4.19}$$

ド・モルガンの法則 (4.13) で否定 (¬) を内側に送り込むと

$$\mathring{\forall}t_1 \cdots \mathring{\forall}t_k \left(\Big(\exists x \, (\neg\neg A(x,t_1,\ldots,t_k)) \Big) \implies \Big(\mathring{\exists}x \, (\neg\neg A(x,t_1,\ldots,t_k)) \Big) \right). \tag{4.20}$$

これに式 (4.12) を適用して二重否定を除去すると次が言えるわね：

$$\mathring{\forall}t_1 \cdots \mathring{\forall}t_k \left(\Big(\exists x \, (A(x,t_1,\ldots,t_k)) \Big) \implies \Big(\mathring{\exists}x \, (A(x,t_1,\ldots,t_k)) \Big) \right). \tag{4.21}$$

——これで式 (T') が証明できたわね」

「なるほど……」魔理沙が証明の過程を一行ずつペンで押さえながらチェックしながら言った.

「さっきの式 (4.10') のようなスタイルで今の結果をまとめればこうなるわね」:

$$\mathrm{st}(t_1,\ldots,t_k) \wedge \exists x A(x,t_1,...,t_k) \implies \mathring{\exists}x A(x,t_1,...,t_k). \tag{4.22}$$

「公理図式 (T) や (T') では，矢印《$\implies$》の逆も当然成り立つわ．だから結局，自由変数が $x,t_1,...,t_k$ である内的論理式 A について次が成立することになる」:

$$\mathrm{st}(t_1,...,t_k) \implies \Big(\big(\mathring{\forall}x A(x,t_1,...,t_k) \big) \iff \big(\forall x A(x,t_1,...,t_k) \big) \Big). \tag{4.23}$$

$$\mathrm{st}(t_1,...,t_k) \implies \Big(\big(\mathring{\exists}x A(x,t_1,...,t_k) \big) \iff \big(\exists x A(x,t_1,...,t_k) \big) \Big). \tag{4.24}$$

「うーん，つまり……どういうことだろう？」魔理沙が言った.

「一番わかりやすいのは $k=0$ の場合ね」:

$$\big(\mathring{\forall}x A(x) \big) \iff \big(\forall x A(x) \big). \tag{4.23'}$$

$$\big(\mathring{\exists}x A(x) \big) \iff \big(\exists x A(x) \big). \tag{4.24'}$$

「——そして《ある集合が存在する》ことを表す内的論理式を考えてみる．たとえば」:

$$\exists x (x = \varnothing).$$

「これは《空集合》が存在すると言ってるわけだね，うん」魔理沙が言った.

「空集合公理 (4.100) を使って定数記号 $\varnothing$ を消すと——

$$\exists x \ (\forall t \ \ t \notin x).$$

——これに式 (4.24') を適用すると次が得られるわね：

$$\mathring{\exists}x \ (\forall t \ \ t \notin x).$$

——外延性公理により，結局次のようになるわね」:

$$\mathring{\exists}x \ (x = \varnothing).$$

「……ということは $\mathrm{st}(\varnothing)$ ということか」魔理沙が言った.

「これで，少なくとも 1 つの標準的な集合が存在することがわかったわね」

「あー，言われてみれば今までは《標準的な集合が少なくとも 1 つ存在する》ことすら知らなかったわけか」魔理沙が言った.

「さて，次の命題は今までの話がわかっていれば簡単に証明できるはずよ」:

命題1

1. 標準的な集合は，空でなければ少なくとも 1 つの標準元を持つ:
$$\mathring{\forall} a \ \big(a \neq \varnothing \implies \mathring{\exists} x \in a\big).$$
2. 標準的な集合 a が標準元を持たないならば a は空集合である．
3. 単元集合 $\{x\}$ が標準的であれば x は標準的である．

「証明してみるね．最初のやつは，《a が空集合じゃない》ことを論理式で書けば $\exists x\ (x \in a)$ になって，この論理式の自由変数は a だけ．そしていまは $a \in \mathbf{S}$ と仮定してるんだから (T) が使えて $\mathring{\exists} x \in a$ が得られる．二問目は，一問目を言い換えただけだね．三問目．単元集合 $\{x\}$ が標準的なんだから，一問目で示したことから，最低 1 つの標準元を持たないといけないことがわかる．よって，その 1 つしかない元 x が標準的じゃないといけない．これで終わり！」少し得意そうに魔理沙が言った．
「よくできました．
さて，A を IST の内的な論理式であって，$\mathrm{FV}(A) = \{x_1, ..., x_n\}$ としましょう」
「突然気になったんだけど《論理式の自由変数の集合》って言ったときの《集合》ってのは……？」魔理沙が言った．
「論理式 A に《集合》$\mathrm{FV}(A)$ を対応させる話は，IST のいかなる公理にも先立って語られたわ．だから，$\mathrm{FV}(A)$ のような《集合》は IST で扱われる集合宇宙の《外側》にあると考えるといいわ」わたしは言った．
「あ，そうなのか．なんか話が循環してたらやだなって思ったけどそうじゃないんだね」魔理沙は言った．
この類の話は混乱しがちだし，まったく簡単な話というわけでもない．でも，どうやらいまの説明で魔理沙があっさりと納得してくれたらしいのでわたしはホッとしながら話を続けることにした．
「話を戻すと，この内的論理式に n 個の量化子をつけて得られた論理式を
$$Q_1 x_1 \cdots Q_n x_n \ \ A(x_1, ..., x_n)$$
としましょう」
「えーと，$Q_1, ..., Q_n$ は $\forall$ か $\exists$ なんだね」魔理沙が確認するように言った．
「そう．だから $n = 2$ なら $\langle Q_1, Q_2 \rangle$ は $\langle \forall, \forall \rangle, \langle \forall, \exists \rangle, \langle \exists, \forall \rangle, \langle \exists, \exists \rangle$ のどれか 1 つを表してることになるわね」
「なるほど，ここまではわかった」魔理沙が言った．
「$1 \leq m < n$ とするとき《m 個目からうしろ》の量化を行った論理式
$$Q_m x_m \cdots Q_n x_n \ \ A(x_1, ..., x_n)$$
を B_m と呼ぶと，B_m は内的論理式であって $\mathrm{FV}(B_m) = \{x_1, ..., x_{m-1}\}$ となるわね．よって，式 (4.23)(4.24) を左側から右側に向かって繰り返し適用して量化を外的なものに置き換えていくと
$$\begin{aligned}
& Q_1 x_1 \, Q_2 x_2 \, Q_3 x_3 \cdots Q_{n-1} x_{n-1} \, Q_n x_n \ \ A(\bar{x}) \\
\iff & \mathring{Q}_1 x_1 \, Q_2 x_2 \, Q_3 x_3 \cdots Q_{n-1} x_{n-1} \, Q_n x_n \ \ A(\bar{x}) \\
\iff & \mathring{Q}_1 x_1 \, \mathring{Q}_2 x_2 \, Q_3 x_3 \cdots Q_{n-1} x_{n-1} \, Q_n x_n \ \ A(\bar{x}) \\
& \vdots \\
\iff & \mathring{Q}_1 x_1 \, \mathring{Q}_2 x_2 \, \mathring{Q}_3 x_3 \cdots \mathring{Q}_{n-1} x_{n-1} \, Q_n x_n \ \ A(\bar{x}) \\
\iff & \mathring{Q}_1 x_1 \, \mathring{Q}_2 x_2 \, \mathring{Q}_3 x_3 \cdots \mathring{Q}_{n-1} x_{n-1} \, \mathring{Q}_n x_n \ \ A(\bar{x}).
\end{aligned}$$
ただし，$\bar{x}$ は $x_1, x_2, x_3, ..., x_{n-1}, x_n$ を略記したものよ」
「なるほど，《$\forall$》が《$\mathring{\forall}$》に，《$\exists$》が《$\mathring{\exists}$》に全部置き換わったんだね」魔理沙が確認するように言っ

た.
「以上の結果を命題の形にまとめておくと」:

命題2 A を IST の内的な論理式であって, $\mathrm{FV}(A)=\{x_1,...,x_n\}$ とする. また, $\{Q_1,...,Q_n\}\subseteq\{\forall,\exists\}$ とする. このとき次が成立する:

$$\big(\mathring{Q}_1 x_1 \cdots \mathring{Q}_n x_n \ A(\bar{x})\big) \iff \big(Q_1 x_1 \cdots Q_n x_n \ A(\bar{x})\big). \tag{4.25}$$

ただし, $\bar{x}$ は $x_1,...,x_n$ を略記したものである.

「えーと, $\{Q_1,...,Q_n\}\subseteq\{\forall,\exists\}$ ってことは……そうか, 集合では同じ元が複数あっても一個あるのと同じだから結局《Q_i は $\forall$ か $\exists$ のどちらか $(i=1,...,n)$ とする》と言うのと同じだね」魔理沙が言った.
「そうね. さて, 内的論理式 A が内的量化子を含んでいて良いことに注意すれば更に一般化して次のようなことが言えるわ」:

定理3 (一般化された移行原理)
A を IST の内的な論理式であって, $\mathrm{FV}(A)=\{x_1,...,x_n,y_1,...,y_m\}$ とする. また, $\{Q_1,...,Q_n,Q_{n+1},...,Q_{n+m}\}\subseteq\{\forall,\exists\}$ とする. このとき次が成立する:

$$\begin{aligned}&\big(\mathring{Q}_1 x_1 \cdots \mathring{Q}_n x_n\, Q_{n+1} y_1 \cdots Q_{n+m} y_{n+m} \ A(\bar{x},\bar{y})\big)\\ &\quad\iff \big(Q_1 x_1 \cdots Q_n x_n\, Q_{n+1} y_1 \cdots Q_{n+m} y_{n+m} \ A(\bar{x},\bar{y})\big).\end{aligned} \tag{4.26}$$

ただし, $\bar{x}$ は $x_1,...,x_n$ を, $\bar{y}$ は $y_1,...,y_m$ を略記したものである.

4.2.2 公理図式 (T) の簡単な応用

「公理図式 (T) からわりと一般化された感じするけど, この定理ってどんなところに使えるんだろう?」魔理沙が言った.
「たとえば, 分出公理図式 (4.101) に式 (4.26) を適用すれば $u_1,...,u_n,x,y$ の量化をすべて外的にしたもの, つまりこれも成立することが言える:

$$\mathring{\forall}u_1 ... \mathring{\forall}u_n \ \Big(\mathring{\forall}x \ \mathring{\exists}y \ \forall t \ \big(t\in y \iff t\in x\wedge A(t,u_1,...,u_n)\big)\Big).$$

つまりは次が言えるわね」:

命題4 (標準的な集合からの分出) A が内的論理式で $\mathrm{FV}(A)=\{t,u_1,...,u_n\}$ のとき

$$\mathrm{st}(x,u_1,...,u_n) \implies \{t\in x \mid A(t,u_1,...,u_n)\}\in\mathbf{S}. \tag{4.101'}$$

「今みたいに最初に沢山量化子が並んでいる場合はいいけど, もっと複雑になってきたらどうすればいいんだろう?」
冠頭標準形 (prenex normal form) の話をする頃合いかもしれないな——わたしがそう思ったときアリスが話し始めた——.
「魔理沙, あなたが言う《もっと複雑な場合》というのはこんなふうに場合分けできるわね」:

$$Q_1 x_1 \cdots Q_n x_n \big(\neg(\forall y A(y,\bar{x}))\big). \tag{4.27}$$
$$Q_1 x_1 \cdots Q_n x_n \big(\neg(\exists y A(y,\bar{x}))\big). \tag{4.27'}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\forall y A(y,\bar{x})) \wedge B\big). \tag{4.28}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\exists y A(y,\bar{x})) \vee B\big). \tag{4.28'}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\forall y A(y,\bar{x})) \vee B\big). \tag{4.29}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\exists y A(y,\bar{x})) \wedge B\big). \tag{4.29'}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\forall y A(y,\bar{x})) \implies B\big). \tag{4.30}$$
$$Q_1 x_1 \cdots Q_n x_n \big((\exists y A(y,\bar{x})) \implies B\big). \tag{4.30'}$$
$$Q_1 x_1 \cdots Q_n x_n \big(B \implies (\forall y A(y,\bar{x}))\big). \tag{4.31}$$
$$Q_1 x_1 \cdots Q_n x_n \big(B \implies (\exists y A(y,\bar{x}))\big). \tag{4.31'}$$

「——ただし，$Q_1,...,Q_n$ は内的量化子とする．そして B は自由変数として y を持たないとする」アリスが付け加えた．

「えーと，うん，そんな感じだね．一般化された移行原理を使って $x_1,...,x_n$ についての量化を外的なやつに変えられるけど，y のところでそれ以上進めなくなるよね」

「そうでもないわ．上の式 (4.27) に (4.13) を，そして式 (4.27') に (4.13') を適用できるわね．それから，式 (4.28) に (4.16) を，そして式 (4.28') に (4.16') を適用できる．とまあこんな感じでやれば最初の 6 つはこう書き換えられる（表 4.4）」：

変形前の論理式	変形後の論理式
$Q_1 x_1 \cdots Q_n x_n \big(\neg(\forall y A(y,\bar{x}))\big)$	$Q_1 x_1 \cdots Q_n x_n \exists y \big(\neg A(y,\bar{x})\big)$
$Q_1 x_1 \cdots Q_n x_n \big(\neg(\exists y A(y,\bar{x}))\big)$	$Q_1 x_1 \cdots Q_n x_n \forall y \big(\neg A(y,\bar{x})\big)$
$Q_1 x_1 \cdots Q_n x_n \big((\forall y A(y,\bar{x})) \wedge B\big)$	$Q_1 x_1 \cdots Q_n x_n \forall y \big(A(y,\bar{x}) \wedge B\big)$
$Q_1 x_1 \cdots Q_n x_n \big((\exists y A(y,\bar{x})) \vee B\big)$	$Q_1 x_1 \cdots Q_n x_n \exists y \big(A(y,\bar{x}) \vee B\big)$
$Q_1 x_1 \cdots Q_n x_n \big((\forall y A(y,\bar{x})) \vee B\big)$	$Q_1 x_1 \cdots Q_n x_n \forall y \big(A(y,\bar{x}) \vee B\big)$
$Q_1 x_1 \cdots Q_n x_n \big((\exists y A(y,\bar{x})) \wedge B\big)$	$Q_1 x_1 \cdots Q_n x_n \exists y \big(A(y,\bar{x}) \wedge B\big)$

表 4.4: 量化を左に引っ張り出す方法 (1)
B は自由変数として y を持たないとする

「——残り 4 つは《$\implies$》が邪魔なので

$$(A \implies B) \iff (\neg A \vee B), \tag{4.32}$$

というルールで《¬》と《∧》で置き換えてやればさっきみたいな調子で次のように書き換えられる（表 4.5）」：「なるほど，(i) 自由変数の条件のチェックが必要な場合があるのと，(ii)《$\neg(Q x A(x))$》《$(Q x A(x)) \implies B$》のときだけ量化子を引っ張り出すときに種類が変わる，ってことだけ覚えておけば完全に暗記しなくても良さそうだね」

「そのとおり，なかなか鋭い」アリスが魔理沙を褒めた．

「いまアリスが説明してくれたように，量化子が論理式の木の内側に埋もれているときでも順繰りに左に引っ張り出せるわ」

「このような変形を適宜行えばいいから，命題 2 は，実際にはすべての内的論理式に適用可能」アリスが付け加えた．

「変形は同値性を保ったまま行えるから，たとえば IST の公理でもある外延性公理 (4.99) なら

変形前の論理式	変形後の論理式
$Q_1x_1\cdots Q_nx_n\big((\forall y A(y,\bar{x}))\implies B\big)$	$Q_1x_1\cdots Q_nx_n\exists y\big(A(y,\bar{x})\implies B\big)$
$Q_1x_1\cdots Q_nx_n\big((\exists y A(y,\bar{x}))\implies B\big)$	$Q_1x_1\cdots Q_nx_n\forall y\big(A(y,\bar{x})\implies B\big)$
$Q_1x_1\cdots Q_nx_n\big(B\implies(\forall y A(y,\bar{x}))\big)$	$Q_1x_1\cdots Q_nx_n\forall y\big(B\implies A(y,\bar{x})\big)$
$Q_1x_1\cdots Q_nx_n\big(B\implies(\exists y A(y,\bar{x}))\big)$	$Q_1x_1\cdots Q_nx_n\exists y\big(B\implies A(y,\bar{x})\big)$

表 4.5: 量化を左に引っ張り出す方法 (2)
B は自由変数として y を持たないとする

$$\begin{aligned} & \forall x\ \forall y\ \Big(\big(\forall t\ (t\in x\iff t\in y)\big)\implies x=y\Big). \\ \iff\quad & \forall x\ \forall y\ \exists t\Big(\big(t\in x\iff t\in y\big)\implies x=y\Big). \end{aligned}$$

一般化された移行原理により

$$\begin{aligned} \iff\quad & \mathring{\forall}x\ \mathring{\forall}y\ \mathring{\exists}t\Big(\big(t\in x\iff t\in y\big)\implies x=y\Big). \\ \iff\quad & \mathring{\forall}x\ \mathring{\forall}y\ \Big(\big(\mathring{\forall}t\ (t\in x\iff t\in y)\big)\implies x=y\Big). \end{aligned}$$

のようにできる」
「最後の論理式は，標準的な集合 x,y があったとき，標準的な集合 t についてだけ $t\in x\iff t\in y$ ならば x と y は同じになると言ってるんだね」
「こんな風にして，量化子を引っ張り出したり，その逆に内側に入れたりできるから，結局次が言える」:

定理5 内的論理式 Φ に対して，その量化子をすべて外的量化子に置き換えたものを $\mathring{\Phi}$ とするとき，次が成立する：

$$\Phi\iff\mathring{\Phi}. \tag{4.33}$$

「さて，いままでに空集合公理，分出公理図式と外延性公理に（一般化された）移行原理を適用してみたわね．ZFC から引き継いだ他の公理と置換公理図式にも移行原理は適用できるわ」
「どれどれ，やってみるとこんな感じになるね」そう言って魔理沙はノートに書き出した：

命題6

$$\mathrm{st}(x,y)\implies\{x,y\}\in\mathbf{S}. \tag{4.102'}$$
$$\mathrm{st}(x)\implies\bigcup x\in\mathbf{S}. \tag{4.103'}$$
$$\mathrm{st}(x)\implies\mathscr{P}(x)\in\mathbf{S}. \tag{4.105'}$$

A が内的論理式で $\mathrm{FV}(A)=\{x,y,a,t_1,\ldots,t_n\}$ のとき

$$\begin{aligned} &\mathrm{st}(a,t_1,\ldots,t_n)\wedge\big(\forall x\in a\ \ \exists!y\ \ A(x,y,a,t_1,\ldots,t_n)\big) \\ &\qquad\qquad\implies\big\{y\,\big|\,\exists x\in a\ \ A(x,y,a,t_1,\ldots,t_n)\big\}\in\mathbf{S}. \end{aligned} \tag{4.106'}$$
$$\mathrm{st}(a)\wedge\varnothing\notin a\implies\mathring{\exists}f\colon a\to\bigcup a\ \ \Big(\mathring{\forall}x\in a\ \ (f(x)\in x)\Big). \tag{4.107'}$$

「つぎの命題も大事ね」わたしはノートに次のように書いた：

命題7

$$\mathrm{st}(u,v)\implies u\times v\in\mathbf{S}. \tag{4.34}$$
$$\mathrm{st}(u,v)\implies\mathrm{Map}(u,v)\in\mathbf{S}. \tag{4.35}$$

ただし

$$\mathrm{Map}(u,v) := \left\{ f \in \mathscr{P}(u\times v) \,\middle|\, f\colon u \to v \right\} \tag{4.36}$$

証明：最初の命題は

$$u\times v = \left\{ z \in \mathscr{P}(\mathscr{P}(u\cup v)) \,\middle|\, \exists x\in u\ \ \exists y \in v\ \ (z = \langle x,y\rangle) \right\} \tag{4.37}$$

から直ちに従う．その次は命題 4 と $u\times v \in \mathbf{S}$ を組み合わせれば示せる．（証明終わり）
「あれ，積集合 $u\times v$ って

$$\left\{ \langle x,y\rangle \,\middle|\, x\in u \wedge y \in v \right\} \tag{4.38}$$

じゃないんだ!?」魔理沙が言った．
「"気持ち"としてはそんな感じだけど，分出公理図式を使って構成しようとすると式　(4.37) のようになるわね．ただ，これは $\langle x,y\rangle$ が $\{\{x\},\{x,y\}\}$ だと定義されてることからくる技術的な話であってあまり本質的なことじゃないわ」
「なるほど」魔理沙が言った．
「さて，(T) の一般化を使えば例えば次の命題も示せるわ」：

命題8　$X,Y\in\mathbf{S}$，$f\in \mathrm{Map}(X,Y)\cap\mathbf{S}$ とする．このとき次が成立する：

$$\mathring{\forall} x\in X\ \ \big(\mathrm{st}(f(x))\big). \tag{4.39}$$

証明：一般に次が成り立つ：

$$\forall X\,\forall Y\,\forall f\in \mathrm{Map}(X,Y)\ \ \forall x\in X\ \ \exists! y\in Y\quad (\langle x,y\rangle \in f). \tag{4.40}$$

これに命題 2 を適用すると次が得られ，証明が完了する：

$$\mathring{\forall} X\,\mathring{\forall} Y\,\mathring{\forall} f\in \mathrm{Map}(X,Y)\ \ \mathring{\forall} x\in X\ \ \mathring{\exists}! y\in Y\quad (\langle x,y\rangle \in f). \tag{4.41}$$

「これの場合は量化子が最初の方にずらっと並んでるから命題 2 を使わなくても，(T) を外側から順番に使って最後に (T') を使っても証明できるかな？」
「その最後に出てくる《$\exists! y\in Y\quad (\langle x,y\rangle\in f)$》が実際には

$$(\exists y\in Y\ \ (\langle x,y\rangle\in f))\wedge\forall y_1\in Y\ \ \forall y_2\in Y\ \ \big((\langle x,y_1\rangle\in f)\wedge(\langle x,y_2\rangle\in f)\implies y_1=y_2\big)$$

の略記だということに注意すると，やはり命題 2 を使うほうが楽だと思うわ」わたしは言った．
「なるほどなー」魔理沙が言う．
アリスが口を開いた――
「選択公理には，それと同等な多くの命題がある．章末付録 4.B(p.64) に挙げられている (AC3) は，(i) $\forall\exists$ という量化子の並びを $\exists\forall$ に変換してくれる道具として，(ii) 関数の間接的な構成原理として，非常に重要」
「じゃあその場合についても扱っておきましょうか」：

命題9

A が内的論理式で $\mathrm{FV}(A)=\{x,y,t_1,\ldots,t_n\}$ のとき

$$\mathrm{st}(a,b,\bar{t}) \implies \Big(\big(\mathring{\forall} x\in a\,\mathring{\exists} y\in b\ \ A(x,y,\bar{t})\big) \implies \big(\mathring{\exists} f\colon a\to b\ \ \mathring{\forall} x\in a\ \ A(x,f(x),\bar{t})\big)\Big). \tag{4.42}$$

，ただし $\bar{t}$ は $t_1,\ldots,t_n$ の略記である．

「えーと，ちょっとこの辺で考えさせて．さっきも確かめたように，まず空集合 $\varnothing$ は標準的だよね――」魔理沙は片側に垂らした三つ編みを左手でいじっている．集中しているときの癖だ．
「ええ，そうね」
「――そして，この空集合をもとにして ZFC の公理を適用することで存在を保証される集合もまた標準的になることがわかったんだよね」
「ええ，そのとおりよ」
「得られた標準的な集合に ZFC の公理をどんどん適用して新しい集合を作っても標準的だよね」
「そうね」
「じゃあ結局 $\mathbf{S}$ から出られなくない？」
「いま魔理沙が言ったように，空集合 $\varnothing$ から出発して ZFC の論理式（つまり IST の内的論理式）を用いて具体的に構成された集合はすべて標準的になるわ．例えば $\mathbb{N},\mathbb{Q},\mathbb{R},\mathbb{C}$ のような集合はすべて標準的よ」
「じゃあ，結局 $\mathbf{S}=\mathbf{V}$ ということ？」
「先に答えを言えば，決してそうはならないわ．でもそれを確かめるためにもそろそろ次の公理図式 (I) を見てみましょう」

4.3 公理図式 (I) から導かれること

4.3.1 公理図式 (I) から直ちに導かれること

「では公理図式 (I) を見ていきましょう．論理式 B を IST の内的論理式で $\mathrm{FV}(B)=\{x,y,u_1,...,u_n\}$ とする．このとき，複数の変数 $u_1,...,u_n$ をまとめて $\bar{u}$ と略記すればこうなるわね」：

$$\forall \bar{u}\left(\left(\mathring{\forall}^{\mathrm{Fin}}F\,\exists y\forall x\in F\ \ B(x,y,\bar{u})\right)\iff\left(\exists y\mathring{\forall}x\ \ B(x,y,\bar{u})\right)\right). \tag{4.43}$$

「――そして，B の代わりに $\neg B$ に対して公理図式 (I) を適用して整理すればこうなるわ」：

$$\forall \bar{u}\left(\left(\mathring{\exists}^{\mathrm{Fin}}F\,\forall y\exists x\in F\ \ B(x,y,\bar{u})\right)\iff\left(\forall y\mathring{\exists}x\ \ B(x,y,\bar{u})\right)\right). \tag{4.44}$$

「この形を公理図式 (I) の双対形，と呼ぶ」アリスが付け加えた．
「公理図式 (I) は限定量化を伴って用いられることも多いわ．定理の形で述べておきましょう」：

定理10 限定量化された理想化原理 (BI,BI') 論理式 B を IST の内的論理式で $\mathrm{FV}(B)=\{x,y,X,Y,u_1,...,u_n\}$ とする．このとき次が成立する：

$$\forall \bar{v}\left(\left(\mathring{\forall}^{\mathrm{Fin}}F\subseteq X\,\exists y\in Y\,\forall x\in F\ \ B(x,y,\bar{v})\right)\iff\left(\exists y\in Y\ \ \mathring{\forall}x\in X\ \ B(x,y,\bar{v})\right)\right), \tag{BI}$$

$$\forall \bar{v}\left(\left(\mathring{\exists}^{\mathrm{Fin}}F\subseteq X\,\forall y\in Y\,\exists x\in F\ \ B(x,y,\bar{v})\right)\iff\left(\forall y\in Y\ \ \mathring{\exists}x\in X\ \ B(x,y,\bar{v})\right)\right). \tag{BI'}$$

ただし，$\bar{v}$ は $X,Y,u_1,...,u_n$ の略記である．

「証明するには

$$B'(x,y,\bar{v})\equiv(x\in X)\implies\left((y\in Y)\wedge B(x,y,\bar{v})\right)$$

としてやって B' に (I) とその双対を適用して整理すればいいわ」
「集合論の議論に慣れてないと

$$\begin{aligned}&\mathring{\forall}^{\mathrm{Fin}}F\ \ \exists y\ \ \forall x\in F\ \ \left(x\in X\implies y\in Y\wedge B(x,y,\bar{v})\right)\\ \iff\ &\mathring{\forall}^{\mathrm{Fin}}F'\subseteq X\ \ \exists y\ \ \forall x\in F'\ \ (y\in Y\wedge B(x,y,\bar{v}))\end{aligned}$$

を示すところだけ難しいかもしれない」
「なるほど，じゃあここの証明はあとで自分で考えてみるよ」

4.3.2 公理図式 (I) の簡単な応用

「公理図式 (I) の応用を見ていくわ」:

命題11 次が成立する：

$$\forall a\left(\left(\forall x\in a\,(\mathrm{st}(x))\right)\iff\left(\mathrm{st}(a)\wedge\mathrm{Fin}(a)\right)\right). \tag{4.45}$$

証明：以下のような変形による：

$$\begin{aligned}
&\forall x\in a\,(\mathrm{st}(x))\\
\iff\ &\forall x\in a\left(\mathring{\exists}y\ \ (x=y)\right)\\
\iff\ &\forall x\in a\left(\mathring{\exists}y\in a\ \ (x=y)\right)
\end{aligned}$$

式 (BI') を用いると

$$\begin{aligned}
\iff\ &\mathring{\exists}^{\mathrm{Fin}}F\subseteq a\ \ \forall x\in a\ \ \exists y\in F\ \ (x=y)\\
\iff\ &\mathring{\exists}^{\mathrm{Fin}}F\subseteq a\ \ (a\subseteq F)\\
\iff\ &\mathring{\exists}^{\mathrm{Fin}}F\ \ (F\subseteq a)\wedge(a\subseteq F)\\
\iff\ &\mathring{\exists}^{\mathrm{Fin}}F\ \ (F=a)\\
\iff\ &\mathrm{st}(a)\wedge\mathrm{Fin}(a).
\end{aligned}$$

「いま示した命題の帰結の例は」:

定理12

1. 超準元を持つ集合 a について少なくとも次の一方の条件が成立する：
   (i) a は超準的である．
   (ii) a は無限集合である．
2. 有限集合 a が超準元をもつならば a は超準的である．
3. 超準的な集合は，超準元を少なくとも 1 つ持つ．
4. 無限集合は，超準元を少なくとも 1 つ持つ．

命題13 (順序対 $\langle x,y\rangle$ が標準的であることは $\mathrm{st}(x,y)$ であることと同値)

$$\forall x\forall y\ \left(\mathrm{st}(\langle x,y\rangle)\iff(\mathrm{st}(x)\wedge\mathrm{st}(y))\right). \tag{4.46}$$

「証明はどちらも読者に委ねる」アリスがコメントした．
「えーと，無限集合が超準元を持つということは，$\mathbb{N}$ や $\mathbb{R}$ も超準元を持つわけだね」
「そうね」
「でもさ，$\mathbb{N}$ は超準元なんて持たないんじゃない？」
「どうして？」
「まず $0=\varnothing$ だから

$$0\in\mathbb{N}\cap\mathbf{S} \tag{4.47}$$

でしょ．そして，形式的自然数の集合 $\mathbb{N}$ の定義から

$$\forall n \in \mathbb{N}\ \big(\mathrm{SUCC}(n) \in \mathbb{N}\big)$$

これをこんな風に変形して：

$$\iff\ \forall n \in \mathbb{N}\ \big(\exists m \in \mathbb{N}(m = \mathrm{SUCC}(n))\big)$$

これに定理 5 を使うと

$$\iff\ \mathring{\forall} n \in \mathbb{N}\ \big(\mathring{\exists} m \in \mathbb{N}(m = \mathrm{SUCC}(n))\big) \tag{4.48}$$

$$\iff\ \forall n \in \mathbb{N} \cap \mathbf{S}\quad \big(\mathrm{SUCC}(n) \in \mathbb{N} \cap \mathbf{S}\big). \tag{4.49}$$

てことは，式 (4.47) と (4.49) を組み合わせれば，形式的自然数に関する数学的帰納法から

$$\mathbb{N} \cap \mathbf{S} = \mathbb{N}$$

したがって

$$\forall n \in \mathbb{N}\ \big(\mathrm{st}(n)\big) \tag{4.50}$$

が言えるよね？」

「形式的自然数に関する数学的帰納法をいまの議論に正しく適用するためには

$$K := \{n \in \mathbb{N} \,|\, \mathrm{st}(n)\}\ \big(= \mathbb{N} \cap \mathbf{S}\big)$$

として定義されたクラス K が集合でなければならない．4.1.7 節 (p.36) で議論したように，分出公理図式は適用できないので K が集合であるか否かは確定していなかったわね．だから今の魔理沙の議論は前提の成立が確認されていないという不備があるわ」

「あー，なるほど」

「もし K が集合だとしたら，魔理沙がいま言ったように式 (4.50) が成立してしまう．でも公理図式 (I) についてのさっきの議論によって，$\mathbb{N}$ には少なくとも 1 つの超準元が存在するわ．つまり，K は集合ではあり得ないことになるわね．よって次が証明できたことになるわ：

命題14 $\mathbb{N} \cap \mathbf{S}$ は外集合である．

「なるほど，外的論理式を使って作られたクラス X を持ってきて，《X が集合だとすると矛盾する》という事から背理法によって，クラス X が外集合であることを確定させるという論法か！」

「このような背理法によってクラス X が外集合であることを導くというのは——外集合の定義を考えれば——最も素直なやり方」アリスがコメントした．

4.3.3 S が真クラスであること

「命題 11 の別の応用として，$\mathbf{S}$ が真クラスであることも示せるわ．実際，仮に $\mathbf{S}$ が集合ならば，$\forall x \in \mathbf{S}\ \big(\mathrm{st}(x)\big)$ であることにより $\mathrm{st}(\mathbf{S})$ が従うわね．これは $\mathbf{S} \in \mathbf{S}$ を意味するけど，これは基礎の公理を使えばあり得ないことがわかる．よって，$\mathbf{S}$ は真クラスでなければならない」

「なるほどね．でもまあ意外感はないかな？ だって，空集合から出発して色々な集合を ZFC の公理を組み合わせてつくっていってもずっと $\mathbf{S}$ の中にあるから $\mathbf{S}$ ってのはめちゃくちゃ大きいよね」魔理沙が言った．

4.3.4 超準自然数の存在

「それにしてもさ，式 (4.47) と (4.49) を組み合わせてくと，一連の形式的自然数：

$$0, 1(= \mathrm{SUCC}(0)), 2(= \mathrm{SUCC}(1)), 3(= \mathrm{SUCC}(2)), 4(= \mathrm{SUCC}(3)), \ldots$$

はどれも標準的なはずだよね？」

「そのとおり．いまそれをここでやるかは別として，《形式化された自然数としての 100》も標準的だし，もっともっと大きな数，たとえば 10^{100} のような数を形式化された自然数とみなしたものも

標準的になるわね」
「とすると，超準的な自然数というのはどこにあるの？」
「超準的な自然数 n_0 があったとすると，いかなる $n \in \mathbb{N} \cap \mathbf{S}$ より大きいことがわかるわ．これは背理法によって次のように示せるわよ」：
仮に $\mathring{\exists} n \in \mathbb{N}\ (n_0 \leq n)$ とするとこれはつまり

$$\mathring{\exists} n \in \mathbb{N}\ (n_0 \in \mathrm{SUCC}(n))$$

を意味する．n が標準的だから $n' := \mathrm{SUCC}(n)$ も標準的である． もちろん $\mathrm{Fin}(n')$ なので，命題 11 によって n' の元 n_0 は標準的でなければならない．ところがこれは n_0 が超準的であるという仮定と矛盾する．よって

$$\neg\big(\mathring{\exists} n \in \mathbb{N}\ (n_0 \leq n)\big)$$

でなければならない．否定 $(\neg)$ を内側に送り込んで整理すると次が得られる：

$$\mathring{\forall} n \in \mathbb{N}\ (n_0 > n).$$

したがって，n_0 はいかなる標準的自然数より大きいことが示された．（証明終わり）
「えーと，10^{100} とかもっとずっと大きな自然数も 0 から出発して，n から $\mathrm{SUCC}(n)$ を作るやつを繰り返せば作れるけど，そういうのをずーっと繰り返しても超準的な自然数にはたどり着けないということ？」
「そう，私達が 0 を元にして有限回 SUCC を重ねても絶対に超準的な自然数にはたどり着けない」アリスが言った．
「さっきアリスが《集合 $\mathbb{N}$ がバンダースナッチのように危険で油断のならないやつかもしれない》って言ってた気持ちがわかってきた気がするよ」
「普段わたしたちが ZFC の中で数学を展開しているとき，ZFC における《集合の有限性》とメタレベルで物事を扱うときの有限性が一致するものと考えてることが多いと思う．そして，IST では《集合の有限性》の定義を ZFC からそのまま受け継いでいるわ．でも形式的自然数の集合 $\mathbb{N}$ を IST で眺めてみると，《どんな標準的自然数 n に対しても $n < n_0$ である》ような巨大な自然数 n_0 が存在することになるわ．そして確かにこれは奇妙な感じがするわよね．ちなみに，IST の創始者ネルソンは《集合の有限性》に関して [3] においてこんなコメントを残してるわ」[*1]：

> —恐らくこのように言って良いだろう：《有限である》ということは，我々が意味していると普段考えていた事を意味しない．我々は《有限である》ことが何を意味していると普段考えていたのだろうか？　私は，それが意味していると普段思っていた事を知っているとかつては考えていたが，今は違う．何れにせよ，直観は経験によって変わるものだ．

「うーん，言ってること一ミリもわからないけど，とにかくわかりにくいポイントだってことはわかったよ」魔理沙が気楽そうな口ぶりで言った．
「わたしたちの身の回りにある，具体的な有限集合についての考察を IST に持ち込むとき標準的な有限集合とみなすとしっくり来るかもしれないわね．そして，漠然とした大きな有限集合を考えて

[*1] 原文：— Perhaps it is fair to say that “finite” does not mean what we have always thought it to mean. What have we always thought it to mean? I used to think that I knew what I had always thought it to mean, but I no longer think so. In any case, intuition changes with experience.

いるときは IST に持ち込むときは超準的な有限集合と考えるのがいいかもしれない」
「うーん，よくわからないな．さっきのネルソンの言葉みたいに《直観が経験で変わる》のを期待するしかなさそうだなぁ」魔理沙が言った．

4.4 非常に大きな有限集合の存在

「さて，次の話を聞くとまた混乱するかもしれないわね」:

定理15

$$\exists^{\mathrm{Fin}} K \quad \big(\mathbf{S} \subseteq K\big). \tag{4.51}$$

証明：以下のような変形による：

$$\begin{aligned}
&\exists^{\mathrm{Fin}} K \quad \big(\mathbf{S} \subseteq K\big).\\
\Longleftrightarrow\ &\exists^{\mathrm{Fin}} K \quad \mathring{\forall} x \ \ (x \in K).\\
\Longleftrightarrow\ &\exists K \ \ \big(\mathrm{Fin}(K) \wedge \mathring{\forall} x \ \ (x \in K)\big).\\
\Longleftrightarrow\ &\exists K \ \ \mathring{\forall} x \ \ \big(\mathrm{Fin}(K) \wedge (x \in K)\big).
\end{aligned}$$

ここで $B(x,K) \equiv \mathrm{Fin}(K) \wedge (x \in K)$ とすると，(I) により

$$\begin{aligned}
\Longleftrightarrow\ &\mathring{\forall}^{\mathrm{Fin}} F \ \ \exists K \ \ \forall x \in F \ \ \big(\mathrm{Fin}(K) \wedge (x \in K)\big).\\
\Longleftrightarrow\ &\mathring{\forall}^{\mathrm{Fin}} F \ \ \exists^{\mathrm{Fin}} K \ \ \big(\forall x \in F \ \ (x \in K)\big).\\
\Longleftrightarrow\ &\mathring{\forall}^{\mathrm{Fin}} F \ \ \exists^{\mathrm{Fin}} K \ \ (F \subseteq K).
\end{aligned}$$

最後の論理式は，$K := F$ と取れば確かに成立する．よって一番最初の論理式も成立する．（証明終わり）
「ええと，真クラスである $\mathbf{S}$ が有限集合 K にすっぽり入ってるってこと……だよね？」
「そうね」
「あのさ，真クラスって $\mathbf{V}$ みたいにすごくすごく大きいんじゃないの？」魔理沙が言った．
「$\mathbf{V}$ が真クラスになるのは《それが集合になるには大きすぎるから》だとしばしば説明される．これに対して，ある有限集合 K にすっぽり収まってしまう $\mathbf{S}$ が真クラスになるのは《それが集合になるには複雑すぎるから》だと理解するといいかもしれない」アリスがコメントした．
「うーん，$\mathbf{S}$ がなぁ……めっちゃでかいはずなのにな……有限集合に収まってしまうとは……」
「わたしたちが《有限集合》と言ったとき，そのなかにはとてつもなく大きな《有限集合》も含まれてると思っておけばいいかもしれないわね」
「有限とは……有限って一体なんだ！？」魔理沙が頭を抱えていたと思ったら，急に立ち上がって軽く咳払いをすると目を軽く閉じてこう言った——：
「恐らくこのように言って良いだろう——《有限である》ということは，我々が意味していると普段考えていた事を意味しないのだ！」
普段話しているときには気づかなかったが演劇口調で話す魔理沙の声は豊かな倍音を含んでおり，なかなかの美声と言えた．きっと彼女が歌を歌ったら素敵だろう——そんな事を考えているうちに
「それ，さっきパチュリーが引用したやつ」アリスがぼそりと言った．
しまった，わたしがツッコむつもりだったのにアリスに遅れをとってしまった．気を取り直して会話を続ける．
「わたしたちはそれぞれ経験によって培ってきた直観を基準にして，色々な物事を判断しているわ．その直観と食い違うような命題に直面すると，(i) それが錯誤や錯覚によって間違って証明されてしまったのか，(ii) それとも自分が培ってきた経験的直観が不十分だったのか，わからないから混乱するという場面もあるわよね．そのどちらかが悩まなくていいように，わたしたちは (i) の可能性を

なるべく潰したい．公理を用いて議論することはそのような利点があるわね．そして，ネルソンが言うようにそうやって新しく獲得した経験によって直観も変わっていくはずよ」

4.4.1 《無限に大きな数》や《無限に小さな数》の扱い

「これから実数を扱う議論に入るわ．実数の集合 $\mathbb{R}$ は，"通常行われる" ZFC での議論で構成されるものと同じなので，ここではその構成や $\mathbb{N}\subseteq\mathbb{R}$ とみなすことについては既知として話を進めるわ[*2]．重要なポイントは，移行原理のおかげで $\mathbb{R}$ が標準的集合になるということね」

「無限小とかの話っていかにも超準解析って感じだよね」魔理沙が言った．

「さて，IST で《無限に大きな数》や《無限に小さな数》を扱うための用語を用意しましょう」：

定義1 (無限大数，被限数，無限小実数) 以下では $r,x,y\in\mathbb{R}$ とする．また，

$$\mathbb{R}_+ := \{x\in\mathbb{R}\,|\,x>0\}$$

とする．

- r: i-small $\equiv$ $\mathring{\forall}\varepsilon\in\mathbb{R}_+\quad(|r|<\varepsilon)$.
- r: limited $\equiv$ $\mathring{\exists}M\in\mathbb{R}_+\quad(|r|<M)$.
- r: i-large $\equiv$ $\mathring{\forall}M\in\mathbb{R}_+\quad(|r|>M)$.
- $x\simeq y\ \equiv\ (x-y)$: i-small.

1. r: i-small のとき，《r は無限小である》と言う．
2. r: limited のとき，《r は被限である》と言う．
3. r: i-large のとき，《r は無限大である》と言う．
4. $x\simeq y$ のとき，《x は y に無限に近い》と言う．

「ところで《被限》って日本語見慣れないんだけど」魔理沙が言った．

「"limited"という言葉は通常"有限である"と訳されるけど，集合の有限性と紛らしいので造語した．ちなみに，超準解析の古い本 (例えば [8]) では私達が被限，あるいは limited と呼んでる概念に"finite"という言葉を充てている」アリスが答えた．

「なるほど．無限小，被限，無限大を定義するときに一番左に外的量化子が来ているのがポイントなのかな」魔理沙が言った．

「そうね．例えば無限小実数を定義する際にもし

$$\forall\varepsilon\in\mathbb{R}_+\ (|r|<\varepsilon)$$

としてしまっていたら，無限小実数は 0 だけになってしまったことでしょう」

「なるほど……あれ？ ところで 0 じゃない無限小実数って存在するのかな？」魔理沙が言った．

「それは公理図式 (I) を使えば次のように簡単に言えるわ」：

$$\begin{aligned}&\exists x\in\mathbb{R}\quad\big(x\neq 0\wedge x\text{: i-small}\big).\\ \iff\ &\exists x\in\mathbb{R}-\{0\}\quad\mathring{\forall}\varepsilon\in\mathbb{R}_+\quad(|x|<\varepsilon).\end{aligned}$$

(BI) により

$$\iff\ \mathring{\forall}^{\mathrm{Fin}}F\subseteq\mathbb{R}_+\quad\exists x\in\mathbb{R}-\{0\}\quad\forall\varepsilon\in F\quad(|x|<\varepsilon).$$

[*2] 日本語の本では，たとえば [21][22] などに詳述されている．

最後の論理式は，例えば F に対して

$$x := \frac{1}{2}\min F$$

とすれば成立する．よって最初の論理式も成立する．つまり，無限小実数は存在する．

「んー？　最後のやつはすごく当たり前の事になってるよね……？」魔理沙が不思議そうに言った．

「公理図式 (I) を使ってその《当たり前》のことを別の角度から眺めると《0 でない無限小実数が存在する》になるとも言えるわね」

「なるほど，ゆっくりとわかってきた感じがする」魔理沙が言った．

「さっきまでは自然数について論じていたから少し補いましょう．集合についての包含関係 $\mathbb{N}\subseteq\mathbb{R}$ を利用すれば $n\in\mathbb{N}$ についても《n: i-large》と言うことは意味をなすわね．無限大の形式的自然数については次が成立するわ（証明は簡単なので略す）」:

定理16 (無限大自然数の特徴づけ)

$$\forall n\in\mathbb{N}\quad \Big(n\text{: i-large} \iff \big(\mathring{\forall} a\in\mathbb{N}\ \ (n>a)\big)\Big). \tag{4.52}$$

「ところで気になったんだけど，さっき超準的な自然数 n は無限大である——つまり n: i-large であること——ことを確認したけど，その逆は成り立つのかな？」魔理沙が言った．

「つまり，《$n\in\mathbb{N}$ が n: i-large ならば n は超準的か？》ということね」

「うん」魔理沙が言った．

「背理法によって簡単に示せるわ．仮にある $n_0\in\mathbb{N}$ が標準的かつ無限大だったとしましょう．すると

$$\mathring{\forall} n\in\mathbb{N}\ \ (n_0>n).$$

となる．ここで集合 $\mathbb{N}$ はもちろん標準的で，仮定によって n_0 も標準的．よって (T) により

$$\forall n\in\mathbb{N}\ \ (n_0>n).$$

よって，$n=n_0$ の場合を考えれば $n_0>n_0$ となるはず．これはおかしいので最初の仮定，つまり無限大自然数が標準的だという仮定は誤り．よって無限大自然数は超準的でなければならないことがわかる」

「なるほど，じゃあ結局さっきの《超準的な自然数は無限大》と合わせると結局次がわかったわけだね」:

命題17

$$\big\{n\in\mathbb{N}\,\big|\,\neg\mathrm{st}(n)\big\}=\big\{n\in\mathbb{N}\,\big|\,n\text{: i-large}\big\}. \tag{4.53}$$

「ところで式 (4.53) の両辺のクラスは集合かしら，それとも外集合かしら？」わたしは魔理沙に聞いてみた．

「えーと，等号の左側は $\mathbb{N}-\mathbf{S}$ って書けるよね．で，さっき $\mathbb{N}\cap\mathbf{S}$ が外集合だと示したから……」

「続けて」わたしは促す．

「えーと，だから……だからなんだろう？」

「等号の右側を使ってみたら？」

「なるほど，そっちか．もし $\big\{n\in\mathbb{N}\,\big|\,n\text{: i-large}\big\}$ が集合だったら最低一つの要素を持つので空集合ではあり得ないね．そして，《$\mathbb{N}$ の部分集合は空でなければ最小値を持つ》んだから，《最小の無限大自然数》N が存在するはずだよね．明らかに 0 は無限大自然数じゃないから，$N>0$ で，よっ

て $M := N-1$ も $\mathbb{N}$ に入る．N の最小性から $M(=N-1)$ は無限大ではない．したがってこうなるよね：

$$\neg\mathring{\forall} n \in \mathbb{N} \quad (M > n).$$

これを変形して整理すると

$$\begin{aligned} &\iff \mathring{\exists} n \in \mathbb{N} \quad (M \leq n). \\ &\implies \mathring{\exists} n \in \mathbb{N} \quad (N = M+1 \leq n+1). \end{aligned}$$

この n が標準的なんだから $n+1$ も当然標準的となって，結局

$$\implies \mathring{\exists} n' \in \mathbb{N} \quad (N \leq n').$$

だけど最後に出てきたやつは N が無限大自然数であることと矛盾するね．よって式 (4.53) の両辺は集合ではあり得ない．つまり外集合である！」

「よくできました．ところでさっきは等号の左を使った議論で詰まってたけど，あの議論を続けるならばこうなるわ」：

仮に $\mathbb{N}-\mathbf{S}$ が（外集合ではなく）集合だったとすると，$\mathbb{N}\cap\mathbf{S} = \mathbb{N}-(\mathbb{N}-\mathbf{S})$ も集合になるはずである．しかしすでに確かめたように $\mathbb{N}\cap\mathbf{S}$ は集合ではない．よって $\mathbb{N}-\mathbf{S}$ は外集合である（証明終わり）．

「あー，こうすればよかったのか．でも全部論理で済んじゃって不思議だなこういうのって」

「一つの命題に二通り以上の証明をつけてみるのは直観を養う良い練習になるわ」わたしは言った．

4.4.2 光暈と銀河

超準解析でよく使われる《**光暈 (halo)**》と《**銀河 (galaxy)**》という概念を導入しましょう．

定義2 (光暈と銀河) x を実数とする．

$$\mathrm{hal}(x) := \left\{ t \in \mathbb{R} \,\middle|\, t \simeq x \right\}. \tag{4.54}$$

$$\mathrm{gal}(x) := \left\{ t \in \mathbb{R} \,\middle|\, (t-x)\colon \text{limited} \right\}. \tag{4.55}$$

- $\mathrm{hal}(x)$ を《x の周りの**光暈 (halo)**》と呼ぶ．
- $\mathrm{gal}(x)$ を《x の周りの**銀河 (galaxy)**》と呼ぶ．

「夜空に輝く天体のなかには，光暈，つまり輝く靄をまとったものがあるわ．その天体の強く輝く中心を取り巻くように何千，何万の恒星たちが密集していて，それがわたしたちからは輝く靄のように見えているのね」わたしは言う．

「なるほど，$\mathrm{hal}(x)$ の真ん中に x があって，$[x-0.1, x+0.1]$ とか $[x-0.0001, x+0.0001]$ とかどんどん小さくしていってもその中に点が密集してる感じだね」魔理沙が散文的な返事をする．

「銀河，つまり淡く輝きながら夜空を横切るあの壮大な帯は，黒インクの海にミルクを注いで作ったように見えることから milky way だとか galaxy だとか呼ばれているわね．ミルクが白く輝くのはコロイド粒子による外光の散乱のせいだけど，わたしたちの銀河をなす輝きの帯は，夥しい恒星たちによって構成されているのだわ」

「えーと，$\mathrm{gal}(x)$ の方は，x との距離が標準的な上界で抑えられる感じの点が並んでるから気持ち的にはぶわーって広がってる感じだね」魔理沙が再び散文的な返事をする．アリスがくすりと笑った．

「いま，$x \in \mathbb{R}$ と $r \in \mathbb{R}_+$ に対して

$$\mathrm{Ball}(x; r) := \left\{ t \in \mathbb{R} \,\middle|\, |t-x| < r \right\}$$

とすると

$$\mathrm{hal}(x) = \left\{ t \,\middle|\, \mathring{\forall}\varepsilon \in \mathbb{R}\ \ (t \in \mathrm{Ball}(x;\varepsilon)) \right\}. \tag{4.56}$$

$$\mathrm{gal}(x) = \left\{ t \,\middle|\, \mathring{\exists}\varepsilon \in \mathbb{R}\ \ (t \in \mathrm{Ball}(x;\varepsilon)) \right\}. \tag{4.57}$$

となってこれらがある種の対称性を持つ概念だということがわかるわ」

「ところで，光暈とか銀河って外集合になるのかな？」魔理沙が尋ねた．

「幾つかの形のクラスが外集合になることをまとめて示しましょう」わたしは言った：

命題18 x を任意の実数とする．このとき，以下の 3 つのクラスはいずれも外集合である：

1. $\mathrm{gal}(x)$
2. $\left\{ t \in \mathbb{R} \,\middle|\, (t-x) \text{: i-large} \right\}$
3. $\mathrm{hal}(x)$

証明：いずれも背理法によって示す．1. 仮に $\mathrm{gal}(x)$ が集合であったとしたら

$$\left\{ x+n \,\middle|\, n \in \mathbb{N} \right\} \cap \mathrm{gal}(x) \tag{4.58}$$

も集合となるはずである．次の事実に注意する：

$$\begin{aligned} & z \in \left\{ x+n \,\middle|\, n \in \mathbb{N} \right\} \cap \mathrm{gal}(x). \\ \iff\ & \exists n \in \mathbb{N}\ \ \Big((z = x+n) \wedge \mathring{\exists} M \in \mathbb{R}_+\ \ (n < M) \Big). \\ \iff\ & \mathring{\exists} n \in \mathbb{N}\ \ (z = x+n). \end{aligned}$$

（直前の $\Longrightarrow$ 部分の議論： $K := \lceil M \rceil$ とすると $K \in \mathbb{N} \cap \mathbf{S}$ となる．そして $n \in K$ であり，さらに $K \in \mathbb{N}$ が有限集合であるから命題 11 を適用すると $n \in \mathbb{N} \cap \mathbf{S}$ であることがわかる．）（直前の $\Longleftarrow$ 部分の議論：自明）

$$\iff\ z \in \left\{ x+n \,\middle|\, n \in \mathbb{N} \cap \mathbf{S} \right\}.$$

クラス $\left\{ x+n \,\middle|\, n \in \mathbb{N} \cap \mathbf{S} \right\}$ は外集合なので（何故？）$\mathrm{gal}(x)$ が集合であるという仮定は矛盾を導く．したがって $\mathrm{gal}(x)$ は外集合である．

2. $\mathrm{gal}(x) = \mathbb{R} - \left\{ t \in \mathbb{R} \,\middle|\, (t-x) \text{: i-large} \right\}$ とすでに示した 1. より従う．

3. $\left\{ t \in \mathbb{R} \,\middle|\, (t-x) \text{: i-large} \right\} = \left\{ t \in \mathbb{R} \,\middle|\, (t-x)^{-1} \in \mathrm{hal}(0) \right\}$ とすでに示した 2. により $\mathrm{hal}(0)$ は外集合でなければならないことがわかる．$\mathrm{hal}(0)$ が外集合であることと $\mathrm{hal}(0) = \left\{ t \in \mathbb{R} \,\middle|\, t+x \in \mathrm{hal}(x) \right\}$ から $\mathrm{hal}(x)$ が外集合であることが従う．

「なるほど…… 1. は結局のところ命題 14 に帰着して証明されていて，そのあとはその一個前の命題を利用して証明したんだね」魔理沙が言った．

「あるクラス X が外集合であることを示すための定義通りの素直な方法は，X が集合だという仮定から矛盾を導くことだけど，それはいつも簡単なわけじゃないわ．超準解析でよく使われるのはいまやったように，すでに外集合であることが知られているものに帰着させるやり方ね」

「なるほど，これまでに命題 14 や命題 18 が示されたんだから，今後はクラス X が外集合であることを証明したいとき，たとえば《もし X が集合だとすると $\mathrm{gal}(x)$ が集合でなければならない》って感じで背理法でやればいいんだね」魔理沙が言った．

4.4.3 分離原理と Fehrele の原理

「光暈は式 (4.56) の形で，銀河は式 (4.57) の形で表現することができたわね．このようなタイプのクラスの間に一般的な関係が成立することが知られているわ」：

定理19 (分離原理) $X, Y \in \mathbf{S}$ とする．また，A を $\mathrm{dom}(A) = X$ であるような関数とし，B を $\mathrm{dom}(B) = Y$ であるような関数とする．また，クラス G, H を次のように定義する：

$$G := \left\{ t \;\middle|\; \mathring{\exists} x \in X \;\; \big(t \in A(x)\big) \right\}, \tag{4.59}$$

$$H := \left\{ t \;\middle|\; \mathring{\forall} y \in Y \;\; \big(t \in B(y)\big) \right\}, \tag{4.60}$$

このとき次が成立する：

$$G \subseteq H \implies \exists J \;\; (G \subseteq J \subseteq H). \tag{4.61}$$

「上の G の形のクラスを《銀河的クラス (galactic class)》と，H の形のクラスを《光暈的クラス (halic class)》と呼ぶことにしましょう」

「えーと，この定理は何が偉いんだろう？」魔理沙が言った．

「この定理が特に価値を発揮するのは G と H が外集合になってしまう場合ね．真クラスというのは集合論の枠からはみ出てしまうからとらえどころがないけど，$G \subseteq H$ ならばその間に J という集合が存在することが保証されるのだから，集合論の議論の足がかりを提供してくれる定理だと言えるかもね．では証明を与えましょう」：

証明：以下を観察せよ：

$$G \subseteq H \iff \forall t \;\; (t \in G \implies t \in H). \tag{4.62}$$

$$\iff \forall t \;\; \Big(\big(\mathring{\exists} x \in X \;\; (t \in A(x)) \big) \implies \big(\mathring{\forall} y \in Y \;\; (t \in B(y)) \big) \Big). \tag{4.63}$$

$$\iff \forall t \;\; \mathring{\forall} x \in X \;\; \mathring{\forall} y \in Y \quad \big(t \in A(x) \implies t \in B(y) \big). \tag{4.64}$$

$$\iff \mathring{\forall} x \in X \;\; \mathring{\forall} y \in Y \;\; \forall t \quad \big(t \in A(x) \implies t \in B(y) \big). \tag{4.65}$$

$$\iff \mathring{\forall} x \in X \;\; \mathring{\forall} y \in Y \;\; \big(A(x) \subseteq B(y) \big). \tag{4.66}$$

命題 13 により

$$\iff \mathring{\forall} z \in X \times Y \quad \big(A(\mathrm{fst}(z)) \subseteq B(\mathrm{snd}(z)) \big). \tag{4.67}$$

ただし，$\mathrm{fst}(\langle x,y \rangle) = x, \mathrm{snd}(\langle x,y \rangle) = y$ とする．

また，次を観察せよ：

$$G \subseteq J \iff \forall t \;\; (t \in G \implies t \in J). \tag{4.68}$$

$$\iff \forall t \;\; \Big(\big(\mathring{\exists} x \in X \;\; (t \in A(x)) \big) \implies t \in J \Big). \tag{4.69}$$

$$\iff \forall t \;\; \mathring{\forall} x \in X \quad (t \in A(x) \implies t \in J). \tag{4.70}$$

$$\iff \mathring{\forall} x \in X \;\; \forall t \quad (t \in A(x) \implies t \in J). \tag{4.71}$$

$$\iff \mathring{\forall} x \in X \quad (A(x) \subseteq J). \tag{4.72}$$

全く同様にして次も示せる：

$$J \subseteq H \iff \mathring{\forall} y \in Y \quad (J \subseteq B(y)). \tag{4.73}$$

以上の式 (4.72)(4.73) により次がわかる：

$$G \subseteq J \subseteq H \iff \mathring{\forall} x \in X \;\; \mathring{\forall} y \in Y \quad \big(A(x) \subseteq J \subseteq B(y) \big) \tag{4.74}$$

$$\iff \mathring{\forall} z \in X \times Y \quad \big(A(\mathrm{fst}(z)) \subseteq J \subseteq B(\mathrm{snd}(z)) \big). \tag{4.75}$$

従って次が言える：

$$\exists J \;\; (G \subseteq J \subseteq H) \tag{4.76}$$

$$\iff \exists J \;\; \mathring{\forall} z \in X \times Y \quad \big(A(\mathrm{fst}(z)) \subseteq J \subseteq B(\mathrm{snd}(z)) \big) \tag{4.77}$$

(I) により

$$\iff \quad \mathring{\forall}^{\mathrm{Fin}} F \subseteq X \times Y \ \ \exists J \ \ \forall z \in F \quad \big(A(\mathrm{fst}(z)) \subseteq J \subseteq B(\mathrm{snd}(z))\big). \tag{4.78}$$

いま，F を $X \times Y$ の任意の標準的有限集合としよう．すると命題 11 により任意の $z \in F$ は標準元であり，式 (4.67) により $A(\mathrm{fst}(z)) \subseteq B(\mathrm{snd}(z))$ となる．よって：

$$\forall z \in F \ \ \big(A(\mathrm{fst}(z)) \subseteq B(\mathrm{snd}(z))\big). \tag{4.79}$$

いま，

$$J := \bigcup \big\{ A(\mathrm{fst}(z')) \,\big|\, z' \in F \big\} \tag{4.80}$$

とすると明らかに次が成立する：

$$\forall z \in F \quad \big(A(\mathrm{fst}(z)) \subseteq J \subseteq B(\mathrm{snd}(z))\big). \tag{4.81}$$

よって式 (4.78) が，従って式 (4.76) が成立する．（証明終わり）．

「あー，論理式の変形とかを除けば一番核心的なのは標準的有限集合 $F \subseteq X \times Y$ に命題 11 を使うところみたいだね」魔理沙が言った．

「これを使って次の Fehrele の原理を示せるわ」：

定理20 Fehrele の原理 G を銀河的クラスとし，H を光暈的クラスとする．また，$G \subseteq H$ とする．次の条件が満たされているとき $G \subsetneq H$ となる：

- G と H がともに外集合である．

証明：定理 19 より明らか．（外集合が集合と一致することはあり得ないことに注意する）．

「この Fehrele ってのは人の名前？」魔理沙が尋ねた．

「架空の人物だけどね．この原理を発見したのは Marc Diener と I.P. Van den Berg よ [5]」

「G と H の片方だけ外集合のときはどうなるんだろう？」魔理沙が言った．

「片方が集合でもう一方が外集合なら，定理 19 を使うまでもなく両者が一致するわけないわね」

「なるほど，そりゃそうか」魔理沙が言った．

4.4.4 数列の収束とロビンソンの原理

「数列の収束性について整理しておきましょう．実数列 $(a_n)_{n\in\mathbb{N}}$ というのは要するにある関数 $f\colon \mathbb{N} \to \mathbb{R}$ の $n \in \mathbb{N}$ における関数値 $f(n)$ を a_n と書いたものに過ぎないわね．次の定理 ([4],p.44) が成立するわ：

定理21 (数列の収束性の言い換え) $(a_n)_{n\in\mathbb{N}}$ を標準実数列とする．このとき次は同値：

1. $\exists \alpha \in \mathbb{R} \ \ \lim_{n\to\infty} a_n = \alpha$.
2. $\mathring{\exists} \alpha \in \mathbb{R} \ \ \forall n \in \mathbb{N} \ \ \big((n\colon \text{i-large}) \implies a_n \simeq \alpha\big)$.

証明：省略する．

4.4.5 ロビンソンの原理

「次の定理は重要よ」：

定理22 (ロビンソンの定理) $(u_n)_{n\in\mathbb{N}}$ を実数列とする．このとき次が成立する：

$$\Big(\mathring{\forall} n \in \mathbb{N} \ \ (u_n \simeq 0)\Big) \implies \big(\exists N \in \mathbb{N}_\infty \ \ \forall n < N \ \ (u_n \simeq 0)\big). \tag{4.82}$$

ただし $\mathbb{N}_\infty := \big\{ n \in \mathbb{N} \,\big|\, n\colon \text{i-large} \big\}$.

証明：$\mathring{\forall} n \in \mathbb{N}\ (u_n \simeq 0)$ と仮定し，

$$G := \left\{ n \in \mathbb{N} \,\middle|\, \mathring{\exists} n' \in \mathbb{N}\ (n = n') \right\}, \tag{4.83}$$

$$H := \left\{ N \in \mathbb{N} \,\middle|\, \mathring{\forall} \varepsilon > 0\ (\forall n < N\ (|u_n| < \varepsilon)) \right\} \tag{4.84}$$

とする．仮定より $G \subseteq H$ である．$G = \mathbb{N} \cap \mathbf{S}$ なので G は外集合である．よって必要ならば定理 20 を使うことにより $G \subsetneq H$ であることがわかる．従って H は超準自然数 N を含む．命題 17 により，N は無限大自然数である．（証明終わり）

「重要性がいまいちわからないな」魔理沙が言った．

「この定理を使うと，例えば《連続関数列の一様収束極限は連続関数である》ことを示せたりするのだけど……」

「締切のため，そこまで書けなかった」アリスが補足した．

「てか大体 (T) で時間かけすぎなんだよね」魔理沙が言った．

4.5　公理図式 (S) から導かれること

4.5.1　公理図式 (S) から直ちに導かれること

「気を取り直して公理図式 (S) について見ていきましょう．念のために再掲すれば

$$\forall \bar{u} \left(\mathring{\forall} x \mathring{\exists} y \mathring{\forall} z (z \in y \iff z \in x \wedge C(z, \bar{u})) \right) \tag{4.85}$$

という形をしていて分出公理図式 (p.62) と少し似ているけど，大事なのは次の二点ね」：

(i) 標準集合から標準集合を作り出す原理である
(ii) 外的論理式も使える．

「なるほど，似てるものを見たときには違いを意識するのが大事だね」魔理沙が言った．

「分出公理図式に似ていることにヒントを得て，上の y を次のように書きましょう」：

$$^{\mathrm{S}}\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\} \tag{4.86}$$

「てことはこうなるんだね」魔理沙が書いた：

- $^{\mathrm{S}}\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\} \in \mathbf{S}$.
- $\mathring{\forall} t \quad \left(t \in {}^{\mathrm{S}}\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\} \iff t \in x \wedge C(t, \bar{u}) \right)$.

「そのとおりよ」わたしは言った．

「あのさ，C が内的でも外的でもいいけどとにかく $\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\}$ っていうクラスは考えられて，

$$t \in x \wedge C(t, \bar{u}) \iff t \in \left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\}$$

となるから結局こう言えるよね」：

$$\mathring{\forall} t \quad \left(t \in {}^{\mathrm{S}}\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\} \iff t \in \left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\} \right). \tag{4.87}$$

「その通りね」表記の多様さに幻惑されずに落ち着いて議論している魔理沙を見てなんだか誇らしい気持ちになった．

「えーと，そうすると公理図式 (S) は，$z \in \mathbf{S}$ だけで検査したらまったくクラス

$$\left\{ z \in x \,\middle|\, C(z, \bar{u}) \right\}$$

と区別できないような標準的集合を作ってくれるわけか」魔理沙が言った.
「そうね. ところで, 次は示せるかしら？」:

命題23

$$\forall \bar{u}\ \mathring{\forall} x\ \left({}^{\mathrm{S}}\{z \in x \,|\, C(z,\bar{u})\} \subseteq x\right). \tag{4.88}$$

「これは簡単だよ. $y := {}^{\mathrm{S}}\{z \in x \,|\, C(z,\bar{u})\} \in \mathbf{S}$ とすると

$$\mathring{\forall} x\ \mathring{\forall} t\ \ (t \in y \implies t \in x) \tag{4.89}$$

となるから, $\mathring{\forall} x$ と $\mathring{\forall} t$ の順番を入れ替えて (T) を使うと

$$\forall t\ \mathring{\forall} x\ \ (t \in y \implies t \in x) \tag{4.90}$$

になるけど, また $\forall t$ と $\mathring{\forall} x$ の順番をまた入れ替えて整理すれば

$$\mathring{\forall} x\ \ (y \subseteq x) \tag{4.91}$$

になるよね」魔理沙が一息で証明を書き上げた.
「(T) が適用可能であるためには《$\mathring{\forall} x\ \ (t \in y \implies t \in x)$》の t 以外の自由変数たち x, y がどちらも標準的でなければならない. この場合は大丈夫」アリスが補足した.
「ありがとう, アリス. (T) を使うときにはいつもこの点のチェックが欠かせないわね」わたしは言った.
「なるほど, 意識してなかったから危なかったな」魔理沙が言った.

4.5.2 公理図式 (S) の応用

「さて, (S) の応用として絶対外せないものを取り上げて, 終わりにしましょう」:

定理24 (「ネルソン 1977 の定理 1.3」) A を $\mathrm{FV}(A) = \{x, y, X, Y, u_1, \ldots, u_n\}$ であるような IST の論理式（外的でも構わない）とする. このとき次が成立する：

$$\forall \bar{u}\ \mathring{\forall} X\ \mathring{\forall} Y \Big(\big(\mathring{\forall} x \in X\ \mathring{\exists} y \in Y\ \ A(x,y,\bar{v}) \big) \implies \big(\mathring{\exists} f \in \mathrm{Map}(X,Y)\ \ \mathring{\forall} x \in X\ \ A(x, f(x), \bar{v}) \big) \Big). \tag{4.92}$$

ただし, $\bar{u}$ は $u_1, \ldots, u_n$ の, $\bar{v}$ は $X, Y, u_1, \ldots, u_n$ の略である.

証明：$X, Y \in \mathbf{S}$ とし, さらに

$$\mathring{\forall} x \in X\ \mathring{\exists} y \in Y\ \ A(x,y,\bar{v}) \tag{4.93}$$

とする. $Y \in \mathbf{S}$ より $\mathscr{P}(Y) \in \mathbf{S}$. $X \in \mathbf{S}$ と合わせれば $X \times \mathscr{P}(Y) \in \mathbf{S}$ であることがわかる. いま, 標準集合 $\mathbf{\Phi}$ を次のように定義する：

$$\mathbf{\Phi} := {}^{\mathrm{S}}\Big\{ \langle x, D \rangle \in X \times \mathscr{P}(Y) \,\Big|\, \mathring{\forall} t\ \ (t \in D \iff t \in Y \wedge A(x,t,\bar{v})) \Big\}. \tag{4.94}$$

次が成立することに注意する：

$$\forall x \in X\ \ \mathring{\exists}! C_x \subseteq Y\ \ \Big(\mathring{\forall} t \left(t \in C_x \iff t \in Y \wedge A(x,t,\bar{v}) \right) \Big). \tag{4.95}$$

($C_x := {}^{\mathrm{S}}\{ y \in Y \,|\, A(x,y,\bar{v}) \}$ とすればよい.)
よって次が言える：

$$\mathring{\forall} x \in X\ \ \mathring{\exists}! D \in \mathscr{P}(Y)\ \ (\langle x, D \rangle \in \mathbf{\Phi}). \tag{4.96}$$

一般化された移行原理により：

$$\forall x \in X \ \exists! D \in \mathscr{P}(Y) \ (\langle x, D\rangle \in \Phi). \tag{4.97}$$

よって $\Phi\colon X \to \mathscr{P}(Y)$ となる．式 (4.93) により，実際には $\Phi\colon X \to (\mathscr{P}(Y) - \{\varnothing\})$ となる．. (AC2)(p.64) により，次のような関数 $g\colon (\mathscr{P}(Y) - \{\varnothing\}) \to Y$ が存在する：

$$\forall y \in (\mathscr{P}(Y) - \{\varnothing\}) \ (g(y) \in y) \tag{4.98}$$

$f := g \circ \Phi$ とすると f は所期の性質を持つ．（証明終わり）
「これの応用も知りたかったけど，これで終わりだね」魔理沙が言った．
「残念ながら」わたしは言った．

4.6 エピローグ

こうして長かったようでもあり短かったようでもあった IST の話が終わったころ，タイミングよく咲夜さんがコーヒーとタルトを運んでくれた．いちじくのタルトのひんやりとした甘みが疲れた脳に染み込んでいく．タルトをすっかり平らげた魔理沙は大きくのびをして残りのコーヒーをごくりと飲み干すと勢い良く立ち上がり
「IST の勉強もできたし，そろそろ帰るね．パチュリーにアリス，今日はありがとうね！」
と言った．見送るためにわたしは後を追いかける．
「ねえ魔理沙，今度はいつ来るの？」
「うーん，また聞きたいことができたら来るよ」
いつでも来ていいのよ，そんな言葉ですらわたしの内心を曝け出してしまうようで怖くなり，その言葉をそっと飲み込み——それでもやはり言おうかと逡巡している間に，魔理沙はひらりと箒にまたがり飛び立ってしまう．
「心配しなくていい，魔理沙ならまた来る」
図書館に戻ったわたしに——まるでわたしの心を見透かしたかのように——アリスが言った．白く長い睫毛の奥から覗く黄金色の瞳の煌めきには，何故かわたしを不安にさせるものがあった．
わたしはテーブルに目を戻すと，魔理沙たちと話しながら書いていたノートを探した．ノートをパラパラとめくったとき，奇妙な事に気づいた——先程まで書いていたはずの内容がすべて消えていたのだ．ノートから紙が破り取られた跡はない．このノートは普段の読書の傍らに雑多な思いつきや抜書をするためのもので，前日までの内容はすべて残っていた．今日書いた分が，そして今日書いた分だけがすべて消失しているのだ．
「さっき書いていたノートを探してるの？」
というアリスの声と視線に導かれてわたしはミニチュアテーブル——先程までこれを囲む小さな椅子にわたし達そっくりの三体の人形が腰掛けていたが，アリスが片付けてしまったらしい——を見やった．ミニチュアテーブルには，人形たちが書きつけていた小さな小さなノートが置かれていた．爪ほどのサイズのノートを苦労しながらパラパラとめくると，どうやらこれはわたしがつけていたノートのミニチュアらしいことがわかった．さらにページをめくっていくと，このミニノートには今日わたしが魔理沙たちと話しがら書いていたところまで再現されていることがわかった．
「このノートは……？ というかわたしがさっきまでつけていたはずのノートは？」
「その小さなノートがさっきまであなたが使っていたノートよ，パチュリー」
「それって一体……どういうこと……？」
何かがおかしい．急に喉が乾いてくる．
「あなたも見ていたとおり，私の人形はモデルそっくりに動く．さっきも言ったけど，わたしの人形には意志も感情も備わっている．そのような設計をするための工夫や苦労も重要だけど，それだ

けじゃ本当に人間らしくはなってくれない．オリジナルの人間と人形の間に《オド》の糸を結ぶことで私の人形は初めて完成する」
《オド》とか《アゾット》と呼ばれる超物質的流体は物質世界はもちろんそれよりもさらに上方の非物質世界の隅々にまで浸透していると考えられている．アリスやわたしが行使する魔術が依拠する遠隔作用の多くはこの流体の媒介によって説明される．
「わたしと人形が《オド》で結ばれるとどうなるの？」
「思考力を持つ二つの対象が《オド》のリンクでつながると，意識の境界が溶け合って一つになるのよ」
「変ね……わたしは自分の体のこの手を動かしてノートを取っていたはずだわ」
「人形のパチュリーが体験したことは《オド》のリンクを通じて本体のパチュリーの脳に書き込まれる．だから自分のその体で経験したことと区別がつかないのは当然」
「なるほどね——にわかには信じがたいけど，このノートの事は大体理解できたわ．ところで，なんでこんなことをしたの？」
「あなたは《私が私じゃなく，他の人だったらどう考えてるんだろう》とか考えたことある？」アリスが返答の代わりに質問を投げてよこした．なるほど——そういうことか．今日だけでなく，アリスは普段から色々な人の人形をこしらえ，そしてそれらの人形と意識を融合させて他人の思考を体験しているのかもしれない．目の前のアリスが《魔理沙の思考を内側から覗き見たことがあるアリス》であり，《わたし（パチュリー）の思考を覗き見たことがあるアリス》でもあり，更に数十人あるいはひょっとすると数百人の他人の思考を内側から覗き見したことがあるのだとしたら——もしそうだとするならば，わたしの目の前にいる《アリス》は本当に純粋な《アリス》だと言えるのだろうか．目の前のアリスは——
「私はバンダースナッチみたいな怪物じゃないわ．わたしはアリス——あなたの知ってるアリス・マーガトロイド．人形遊びが少し得意なだけの普通の魔法使いよ」と言って微笑んだのだった．

(終わり)

メモ

p.47 の命題 11 の証明は A. Robert[4] の p.26 に載っている証明を自分なりに工夫して簡単にしたものです．(Nelson の原論文 [2] に載っている証明はやや込み入っています)．

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，本記事の執筆にあたって以下の文献を参考にいたしました．

4.6.1 参考文献

[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] E. Nelson, “Internal set theory: A new approach to nonstandard analysis”, Bull. Amer. Math. Soc. **83**(6), pp.1165–1198.
IST の創始者ネルソンによる論文です．非常に密度が高く，読者に要求している前提知識の質

も比較的高いと思います.

[3] E. Nelson, “Internal Set Theory”, https://web.math.princeton.edu/~nelson/books/1.pdf
ネルソンが執筆していた教科書の草稿です.

[4] A. Robert, “Analyse non standard”, Presses polytechniques romandes (1985).
ネルソンの IST を解説した本です．この本の英訳 “Nonstandard Analysis” が Dover から出ています（2016 年現在絶版）．公理的集合論の基礎をよく理解している人には易しく読めるでしょう.

[5] F. Diener & M. Diener (Eds.), “Nonstandard Analysis in Practice”, Springer (1991).
ネルソンの IST の様々な応用が述べられています.

[6] R. Lutz & M. Goze, “Nonstandard Analysis”, Springer (1981).
ネルソンの IST の応用を志向した最初期の本です．タイプ原稿であるため若干読みづらいことが惜しまれますが，現在でも読む価値があります.

[7] V. Kanovei & M. Reeken, “Nonstandard Analysis, Axiomatically”, Springer (2000).
主要な関心は Hrbáček（フルバチェク）の超準集合論 HST や BST に置かれていますが，ところどころで IST の話題がとりあげられています．ネルソンの複数の IST 論文での公理の差異や，リダクションアルゴリズムについての細かな指摘が含まれているため大変有用です.

[8] A. Robinson, “Non-standard Analysis”, North-Holland (1974).
超準解析の創始者による本です.

[9] K. Hrbacek,“Nonstandard Objects in Set Theory”, in “Nonstandard Methods and Applications in Mathematics”, Association for Symbolic Logic (2006), pp.80-120.
河合徹氏の超準集合論 KST を含め，過去に提案された種々の超準集合論の概観が与えられています.

[10] 前原昭二 『記号論理入門』 日本評論社 (1967).

[11] 嘉田勝 『論理と集合から始める数学の基礎』 日本評論社 (2008).

[12] リチャード・ジェフリー 『形式論理学—その展望と限界』 産業図書 (1995).

[13] R. Smullyan, “First-Order Logic”, Springer-Verlag (1968).

[14] J. H. Gallier, “Logic for Computer Science (2nd ed.)”, Dover (2015).

[15] E. Mendelson, “Introduction to Mathematical Logic (6th ed.)”, CRC Press (2015).
述語論理については，この記事ではあまり丁寧に説明する余裕がなく，読者がある程度理解していることを仮定して議論せざるを得ませんでした．上に挙げた本は私がこの記事を書くにあたって読んだり眺めたりしたものです（下に行くほど本格的になっていきます）.

[16] P. J. Cohen, “Set Theory and the Continuum Hypothesis”, W. A. Benjamin (1966). (日本語訳： P・J・コーヘン 『連続体仮説』，東京図書 (1972))

[17] K. Kunen, “Set Theory”, College Publications (2011, 2013).

[18] 広瀬健 『数学・基礎の基礎』，海鳴社 (1996).

[19] 田中一之（編）『ゲーデルと 20 世紀の論理 (4) 集合論とプラトニズム』，東京大学出版会 (2007).

[20] 田中尚夫 『公理的集合論』，培風館 (1982).
公理的集合論の箇所を書くときに上記の五冊の書物を参考にしました.

[21] 島内剛一 『数学の基礎』，日本評論社（1971）.

[22] 斎藤正彦 『数学の基礎　集合・数・位相』，東大出版会（2002）.
ZFC 上で解析学を真面目に展開する場合，「われわれの自然数」を ZFC にエンコードして構成した集合 $\mathbb{N}$ から，有理数の集合 $\mathbb{Q}$ や実数の集合 $\mathbb{R}$ を構成してからようやく話がスタートすることになります．上記二書ではそのような話が丁寧に書いてあります.

4.A 付録：ZFC の公理

ここでは，本記事が参照している形式での ZFC 集合論の公理および公理図式を列挙する．

等号公理 1

$$\forall x \ \ (x = x).$$

等号公理 2 A を $\mathrm{FV}(A) = \{x, t_1, ..., t_n\}$ であるような $\mathscr{L}_{\mathrm{ZFC}}$ の論理式とする $(n \geq 0)$．このとき

$$\forall t_1 \ldots \forall t_n \ \forall x \ \forall y \ \Big(\big((x = y) \wedge A(x, t_1, ..., t_n) \big) \implies A(y, t_1, ..., t_n) \Big).$$

等号公理 1 は《いかなる集合もそれ自身に等しい》ということを表している．等号公理 2（正確には「公理図式」だが）は，《命題の成否は，代入された変数を等号で結ばれた別の変数で置き換えても変わらない》ことを表している．（本稿では論理公理の扱いを省略したので集合論の公理に入れておくことにした．）

外延性公理

$$\forall x \ \forall y \ \Big(\big(\forall t \ (t \in x \iff t \in y) \big) \implies x = y \Big). \tag{4.99}$$

外延性公理は，《集合はその元によって決定される》ことを表している．

空集合公理

$$\forall x \ \ x \notin \varnothing. \tag{4.100}$$

空集合公理は，言語 $\mathscr{L}_{\mathrm{ZFC}}$ の定数 $\varnothing$ がいかなる元も持たない集合であることを定義している．外延性公理により，このような集合，つまり「いかなる元も持たない」集合はただ一つ存在することがわかる．

分出公理図式 A を $\mathrm{FV}(A) = \{t, u_1, ..., u_n\}$ であるような $\mathscr{L}_{\mathrm{ZFC}}$ の論理式とする $(n \geq 0)$．このとき

$$\forall u_1 \ldots \forall u_n \ \Big(\forall x \ \exists y \ \forall t \ \ \big(t \in y \iff t \in x \wedge A(t, u_1, ..., u_n) \big) \Big). \tag{4.101}$$

分出公理図式により，集合 x の要素であって A が成り立つような t 全体からなるような集合 y が存在する．この集合 y を

$$\big\{ t \in x \,\big|\, A(t, u_1, ..., u_n) \big\}$$

と表記する．

非順序対の公理

$$\forall x \ \forall y \ \exists z \ \Big(\forall t \ \ \big(t \in z \iff (t = x \vee t = y) \big) \Big). \tag{4.102}$$

非順序対の公理によって，集合 x, y に対して存在することを保証された集合 z を

$$\{x, y\}$$

と書いて，《集合 x, y から作られる非順序対》と呼ぶ．特別な場合として，$\{x, x\}$——つまり x と y が同じ場合——のことは

$$\{x\}$$

と書くことにする．

非順序対を用いて，集合 x, y の《順序対》$\langle x, y \rangle$ が次のように定義される：

$$\langle x, y \rangle := \{ \{x\}, \{x, y\} \}.$$

重要なのは次の性質が成り立つことである：

$$\langle x_1, y_1\rangle = \langle x_2, y_2\rangle \iff (x_1 = x_2) \wedge (y_1 = y_2).$$

和集合公理

$$\forall x\ \exists y\ \forall t\ \big(t \in y \iff \exists s \in x\ \ (t \in s)\big). \tag{4.103}$$

集合 x が与えられたとき，上の《和集合公理》によって存在を保証された集合 y を

$$\bigcup x$$

と書いて《x の和集合》と呼ぶ．

基礎の公理

$$\forall x\ \Big(x \neq \varnothing \implies \exists y \in x\ \forall t \in x\ \ (t \notin y)\Big). \tag{4.104}$$

基礎の公理により，任意の集合 x から出発して

$$x = x_0 \ni x_1 \ni x_2 \ni \ldots$$

のように際限なく続く列は存在しないことが保証される．

冪集合公理

$$\forall x\ \exists y\ \forall t\ \ (t \in y \iff t \subseteq x). \tag{4.105}$$

この公理によって集合 x に対して存在することが保証される y を《x の冪集合》と呼んで，$\mathscr{P}(x)$ と書く．

置換公理図式　A を $\mathrm{FV}(A) = \{x, y, a, t_1, \ldots, t_n\}$ であるような $\mathscr{L}_{\mathrm{ZFC}}$ の論理式とする $(n \geq 0)$．このとき

$$\begin{aligned}\forall t_1 \ldots \forall t_n\ \forall a\ \Big(&\big(\forall x \in a\ \ \exists! y\ \ A(x, y, a, t_1, \ldots, t_n)\big)\\ &\implies \exists b\ \ \forall y\ \ \big(y \in b \iff \exists x \in a\ \ A(x, y, a, t_1, \ldots, t_n)\big)\Big).\end{aligned} \tag{4.106}$$

ここに出てくる《$\exists!$》は《ただひとつ存在する》ことを表すために使われる記号で，次のように定義される：

$$\exists!\ A(x) \quad \equiv \quad \exists x\ A(x) \wedge \forall x_1 \forall x_2\ \ \big(A(x_1) \wedge A(x_2) \implies x_1 = x_2\big).$$

つまり，《存在すること》と《存在するとしたら一致すること》を両方合わせると《ただひとつ存在する》ことを表しているのである．

選択公理

$$\forall a\ \ \Big(\varnothing \notin a \ \implies\ \exists f\colon a \to \bigcup a\ \ \Big(\forall x \in a\ \ (f(x) \in x)\Big)\Big). \tag{4.107}$$

この公理で存在を保証される関数 f を，《集合 a の選択関数》と呼ぶ．

4.B 付録：選択公理の三つの形 (ZFC の定理)

定理25 次の 3 つの命題 (AC1)(AC2)(AC3) は互いに同値である：

(AC1) 選択公理 (式 (4.107)).
(AC2) $\forall a \exists f\colon (\mathscr{P}(a)-\{\varnothing\}) \to a\ \forall x \in (\mathscr{P}(a)-\{\varnothing\})\ \big(f(x) \in x\big)$.
(AC3) A を IST の内的論理式とし，$\mathrm{FV}(A) = \{x, y, t_1, ..., t_n\}$ とする．このとき：
$\forall \bar{t}\ \forall a \forall b \Big(\big(\forall x \in a\ \exists y \in b\ A(x,y,\bar{t})\big) \Longrightarrow \big(\exists f\colon a \to b\ \big(\forall x \in a\ A(x, f(x), \bar{t})\big)\big)\Big)$,
ただし，$\bar{t}$ は $t_1, ..., t_n$ の略記である．

証明：(AC1) ⇒ (AC2) 任意の集合 a に対し $b := (\mathscr{P}(a)-\{\varnothing\})$ とし，集合 b に対して (AC1) を適用すると次が得られる：

$$\forall a \exists f\colon b \to \bigcup b\ \forall x \in b\ \big(f(x) \in x\big). \tag{4.108}$$

ここで

$$\begin{aligned} & t \in \bigcup b \\ \Longleftrightarrow\ & \exists u \in (\mathscr{P}(a)-\{\varnothing\})\ \ t \in u \\ \Longleftrightarrow\ & \exists u \subseteq a\ \ t \in u \Longleftrightarrow t \in a \end{aligned}$$

より $\bigcup b = a$ であることに注意すれば式 (4.108) から所期の命題が得られる．
(AC2) ⇒ (AC3) 集合 $t_1, ..., t_n, a, b$ に対して次が成立していると仮定する：

$$\forall x \in a \exists y \in b\ A(x, y, t_1, ..., t_n). \tag{4.109}$$

さて，任意に与えられた $x \in a$ に対して，分出公理図式を用いて次のように集合 C_x を作る：

$$C_x := \{y \in b \,|\, A(x, y, t_1, ..., t_n)\}.$$

式 (4.109) により $C_x \neq \varnothing$ である．外延性公理公理により，この集合 C_x は次の意味で一意 (unique) に定まることがわかる：

$$\forall x \in a \exists! C_x \in (\mathscr{P}(b)-\{\varnothing\})\ \ \forall z\big(z \in C_x \Longleftrightarrow z \in b \wedge A(x, z, t_1, ..., t_n)\big). \tag{4.110}$$

いま，

$$g := \big\{\langle x, C\rangle \in a \times (\mathscr{P}(b)-\{\varnothing\}) \,\big|\, \forall z\big(z \in C \Longleftrightarrow z \in b \wedge A(x, z, t_1, ..., t_n)\big)\big\}.$$

とすると，(4.110) により $\mathrm{FUNC}(g)$ であり，さらに $g\colon a \to (\mathscr{P}(b)-\{\varnothing\})$ であることがわかる．よって：

$$\exists g\colon a \to (\mathscr{P}(b)-\{\varnothing\})\ \ \forall x \in a \forall z\big(z \in g(x) \Longleftrightarrow z \in b \wedge A(x, z, t_1, ..., t_n)\big). \tag{4.111}$$

(AC2) により：

$$\exists \gamma\colon (\mathscr{P}(b)-\{\varnothing\}) \to b\ \ \forall t \in (\mathscr{P}(b)-\{\varnothing\})\ \ \gamma(t) \in t.$$

そこで $f := \gamma \circ g$ とすると $f\colon a \to b$ であってさらに

$$\forall x \in a\ \ A(x, f(x), t_1, ..., t_n).$$

よって所期の命題が示された．
(AC3) ⇒ (AC1) 集合 a について，$\varnothing \notin a$ ならば次が言える：

$$\forall x \in a\ \ \exists y \in \bigcup a\ \ y \in x. \tag{4.112}$$

というのも，任意の $x \in a$ について $x \neq \varnothing$ なので $\exists y \in x\,(x \in a)$ であり，

$$\begin{aligned} & \exists y \in x \ \ (x \in a) \\ \Longrightarrow\ & \exists y \in x \ \ (y \in x) \wedge (x \in a) \\ \Longleftrightarrow\ & \exists y \in x \ \ \left(y \in \bigcup a\right) \end{aligned}$$

だから．いま $A(x,y) \equiv y \in x$ として論理式 A を定義すると式 (4.112) にに (AC3) を適用することができ，次が得られる：

$$\exists f\colon a \to \bigcup a \ \ \forall x \in a \ \ (f(x) \in x).$$

よって所期の命題が成立することがわかる．

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。

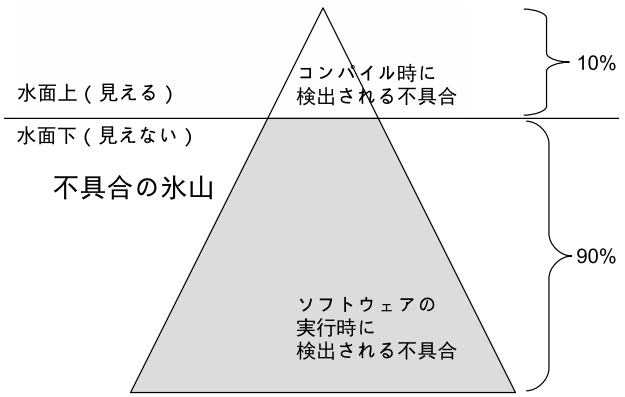


図 5.1: 氷山の一角

[*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)。

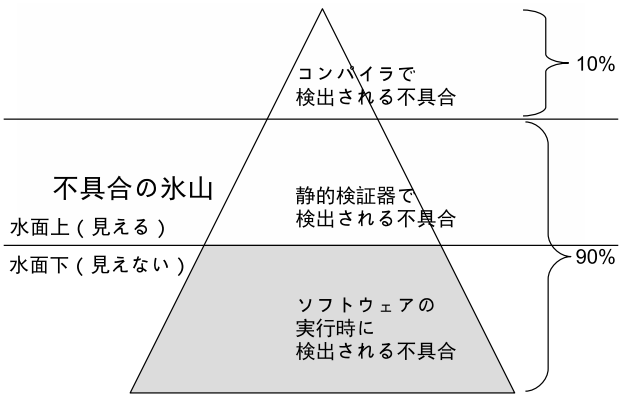


図 5.2: 静的検証でより多くの不具合を見えるように

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

5.3 今日からできる VeriFast でのコード検証

本記事では、オススメの静的コード解析の検証器として VeriFast [*3][*4] を紹介するでゲソ! ZIP をダウンロードして解凍して実行するだけで、IDE 上でコード検証ができるでゲソ! 6 章「VeriFast チュートリアル」でより詳しい解説をするので、興味を持った人はやってみてほしいでゲソ!

ここでは、チュートリアルの例からいくつか引用して、VeriFast で検証できることを見せるでゲソ。

まずは `tutorial_solutions/illegal_access.c` という簡単な例でゲソ (チュートリアル 6.2 節)。

```
#include "stdlib.h"

struct account {
    int balance;
};

int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = malloc(sizeof(struct account));
    //if (myAccount == 0) { abort(); }
    myAccount->balance = 5;
    free(myAccount);
    return 0;
}
```

`illegal_access.c` は明らかに危険なコードじゃなイカ。`malloc(3)` 関数が返すポインタを無条件にデリファレンスして辿ってしまっているでゲソ! 本来はコメントにある `if (myAccount == 0) { abort(); }` のようにポインタが NULL でないことを確認した後にデリファレンスするべきでゲソ。

しかし、このコードはコンパイルエラーは出ず、しかも実行には成功するじゃなイカ。普通は `malloc(3)` は成功するし、しかも普通のコンパイラはデリファレンスしようとしているポインタが NULL かどうか判定する能力がないのゲソ。そして、ごく稀に `malloc(3)` が失敗した時 (場合によっては出荷後) にバグが発覚するでゲソ。これがまさしく海面下にある見えない氷山「実行時エラー」なんじゃなイカ。

ところが、VeriFast を使えばこのバグを検証時に発見できるんでゲソ。そして、
`if (myAccount == 0) { abort(); }` をアンコメントしてバグを修正すれば「正しい」と検証結果が出るでゲソ!

もう少し自明でない例としては `tutorial_solutions/stack.c` があるでゲソ (チュートリアル 6.7 節、練習問題 5 答え)。ここでは自作スタックの深さに関しての検証ができるでゲソ。空のスタックをポップしようとすると VeriFast がエラーを検出するでゲソ!

```
#include "stdlib.h"
struct node {
    struct node *next;
    int value;
};
struct stack {
    struct node *head;
};
/*@
predicate nodes(struct node *node, int count) =
    node == 0 ? count == 0 : 0 < count &*& node->next |-> ?next &*& node->value |-> ?value &*&
        malloc_block_node(node) &*& nodes(next, count - 1);
```

[*3] https://people.cs.kuleuven.be/~bart.jacobs/verifast/

[*4] https://github.com/verifast/verifast

```
predicate stack(struct stack *stack, int count) =
    stack->head |-> ?head &*& malloc_block_stack(stack) &*& 0 <= count &*& nodes(head, count);
@*/

struct stack *create_stack()
    //@ requires true;
    //@ ensures stack(result, 0);
    // 「サイズ 0 のスタックを返す」
{
    struct stack *stack = malloc(sizeof(struct stack));
    if (stack == 0) { abort(); }
    stack->head = 0;
    //@ close nodes(0, 0);
    //@ close stack(stack, 0);
    return stack;
}

void stack_push(struct stack *stack, int value)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count + 1);
    // 「サイズ count の stack を与えて呼び出すと stack のサイズが count + 1 になって返ってくる」
{
    //@ open stack(stack, count);
    struct node *n = malloc(sizeof(struct node));
    if (n == 0) { abort(); }
    n->next = stack->head;
    n->value = value;
    stack->head = n;
    //@ close nodes(n, count + 1);
    //@ close stack(stack, count + 1);
}

int stack_pop(struct stack *stack)
    //@ requires stack(stack, ?count) &*& 0 < count;
    //@ ensures stack(stack, count - 1);
    // 「サイズ count(>0) の stack を与えて呼び出すと stack のサイズが count - 1 になって返ってくる」
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ open nodes(head, count);
    int result = head->value;
    stack->head = head->next;
    free(head);
    //@ close stack(stack, count - 1);
    return result;
}

void stack_dispose(struct stack *stack)
    //@ requires stack(stack, 0);
    //@ ensures true;
    // 「stack のサイズを 0 にしてから呼んでください」
{
    //@ open stack(stack, 0);
    //@ open nodes(_, _);
    free(stack);
}

int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    int i;
    stack_push(s, 10);
    stack_push(s, 20);
    stack_pop(s);
    stack_pop(s);
    stack_pop(s); // 空のスタックを pop している！ エラー！
    stack_dispose(s);
    return 0;
}
```

@ のついたコメントは VeriFast のための注釈で、各関数の呼び出しの前後でどのような条件が成り立っているべきか、などを書くでゲソ。そうすると VeriFast は、実際のコード上でそれらの条件

が実際に成立することを検証するでゲソ。空のスタックのポップなど、注釈に書いた条件が成り立っていない呼び出し方をしたり、あるいは関数の中身の実装が間違っていて注釈に書いた条件を成立させられなかったりするとエラーを報告するでゲソ!

「どのような条件が成り立っているべきか」を定義し、かつ検証可能なものにするためには、きちんとした注釈を書かないといけないでゲソ。例えば、「スタックのサイズ」ってどう定義するでゲソか? `predicate` うんぬんというコメントは、こういう定義をしているでゲソ。詳しくはチュートリアルをやれば分かるようになってくるはずでゲソ。

検証のための注釈は面倒だったり難しそうだったりと思うかもしれないでゲソが、VeriFast を使いこなせるようになると、簡単なバッファオーバーランからメモリリーク、果てはロックやマルチスレッドのバグまで検出できるようになるでゲソ。地獄のようなコーナーケースを人手でつぶすよりは、検証の注釈を書いて検証器に任せる方が楽なんじゃなイカ?

5.4 VeriFast の適用事例

ここまで読んできて、読者の多くは「そーは言っても VeriFast もやっぱり研究用途にしか使われてないんだろう」と思っているかもしれないでゲソ。ところがどっこい適用事例がちゃんとあるじゃなイカ。本記事では次の 2 つを紹介してたいでゲソ。

1 つ目に「Linux カーネルの USB キーボードドライバ (usbkbd) を検証 [*5][*6] 」して不具合を 2 件報告して本家に取り込まれた実績があるでゲソ。その内 1 つは並行実行に関する不具合でソースコードレビューでは簡単には発見できないものでゲソ。厳密なレビューが徹底されているはずの Linux カーネルでも人力に頼っていてはこのような不具合が混入されたまま出荷されてしまうんでゲソ。これが網羅的な検査の威力じゃなイカ。この並行処理の検査については「VeriFast チュートリアル」の「6.23 Mutex」が詳しいでゲソ。

2 つ目に VeriFast は C 言語だけではなく Java 言語もサポートしているんでゲソ。Java 言語での適用事例として「Android のファームウェア更新による権限昇格脆弱性の発見 [*7] 」が挙げられるでゲソ。しかもなんと Microsoft Research の研究者が著者に入ってるじゃなイカ。マイクロソフトも VeriFast を使うなんて、すごい躍進でゲソ!

5.5 他ツールとの比較と VeriFast ならではの旨味

ところで VeriFast は静的コード解析ツールの唯一の選択肢なのでゲソ？ もちろんそんなことはないでゲソ。

おそらく今日もっとも成功している静的コード解析の 1 つは Coverity [*8]でゲソ。C 言語のソースコードを Coverity 検証器に通すと、「不具合のある可能性のある行」を教えてくれるでゲソ。この Coverity は有料の製品なのでゲソが、対象コードがオープンソースで公開されている場合、無料で検査してくれるサービスまであるでゲソ。素晴しいじゃなイカ! これでざっくざっくと不具合を検証器で見つけて潰せるでゲソ。

ここで Coverity を VeriFast と比較してみようじゃなイカ。Coverity の利点は VeriFast では必須だった C 言語ソースコードコメント中の注釈を入れる必要なしに検査してくれることでゲソ。欠点は VeriFast は (ある条件下で) 網羅的に不具合を発見してくれる一方、Coverity はエラーでない行を「不具合のある可能性のある行」だと誤認識してしまうことでゲソ。

簡単に説明していまうなら、C 言語ソースコード中の意味を表わすヒントが注釈として与えれて

[*5] https://people.cs.kuleuven.be/~willem.penninckx/usbkbd/
[*6] http://www.academia.edu/3005187/Verification_of_Linux_Kernel_Modules_Experience_Report
[*7] http://ieeexplore.ieee.org/abstract/document/6956577/
[*8] http://www.coverity.com/

いないため、Coverity のような検証器は検証対象のソースコードを何のヒントもなしに検証しなければならい結果、このような誤認識が発生すると言うこともできるじゃなイカ。どうやら人類は未だ C 言語ソースコードをそのまま意味を汲み取る仕組みを発見できていないようでゲソ。そのため上記のような誤認識を避けるには VeriFast のように注釈をソースコード中に埋め込む必要があるんじゃなイカ？

他にも静的コード解析ツールはたくさん有るでゲソ。特に Frama-C [*9] と VCC [*10] は VeriFast と良く似たツールでゲソ。Frama-C は VeriFast と同様に C 言語ソースコードのコメントに注釈を書いて検証するでゲソ。Frama-C は mbedtls のメモリ破損脆弱性を見つけた事例 [*11] があり、INRIA が開発元なだけあって多数のユーザがいるツールじゃなイカ。一方 VCC は `_(invariant foo)` のような拡張文法の形式で C 言語ソースコードに注釈を書いて検証するでゲソ。VCC はマイクロソフトの Hyper-V ハイパーバイザを検証 [*12] したことでも有名じゃなイカ。

上記のように 3 つのツールの長所短所がわからなかったので、VeriFast の作者である Bart Jacobs [*13] に聞いてみた [*14] でゲソ。その返事をかいつまんでまとめると、以下が言えそうでゲソ。

- WP plugin [*15] を使えば Frama-C でも VeriFast のように関数単位 [*16] で検証できる
- WP plugin は最弱事前条件を SMT ソルバに計算させポインタのデリファレンスを検証する
- 一方 VeriFast はシンボリック実行 [*17] と分離論理 [*18] でポインタを扱う
- WP plugin は並行性を扱えないが、VeriFast は並行性について検証できる [*19]
- VCC は並行性を扱えるが、分離論理を使っていない

とはいえ選択肢が豊富なことは良いことじゃなイカ。「多様性は常に善」という言葉を胸に、まずは VeriFast を使って静的コード解析の深淵なる世界に飛び込もう!っでゲッソ!

[*9] https://frama-c.com/
[*10] https://www.microsoft.com/en-us/research/project/vcc-a-verifier-for-concurrent-c/
[*11] http://qnighy.hatenablog.com/entry/2016/05/06/215000
[*12] https://link.springer.com/chapter/10.1007/978-3-642-05089-3_51
[*13] https://distrinet.cs.kuleuven.be/people/bartj
[*14] https://groups.google.com/forum/#!topic/verifast/xbUHyhPjAe4
[*15] https://frama-c.com/wp.html
[*16] 「VeriFast チュートリアル」の「6.1 導入」にある「モジュラーシンボリック実行」を参照
[*17] 「VeriFast チュートリアル」の「6.1 導入」を参照
[*18] https://en.wikipedia.org/wiki/Separation_logic
[*19] 「VeriFast チュートリアル」の「6.23 Mutex」を参照

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。

このツールを適用することからはじめましょう。

次からダウンロードできる `illegal_access.c` プログラムと一緒に `vfide.exe` を起動してください:

- `https://people.cs.kuleuven.be/~bart.jacobs/verifast/illegal_access.c`

VeriFast IDE にそのプログラムが表示されるでしょう。そのプログラムを検査するために、**Verify** メニューの **Verify program** コマンドを選択するか、**Play** ツールバーボタンを押すか、F5 キーを押してください。図 6.1 のような結果になるでしょう。

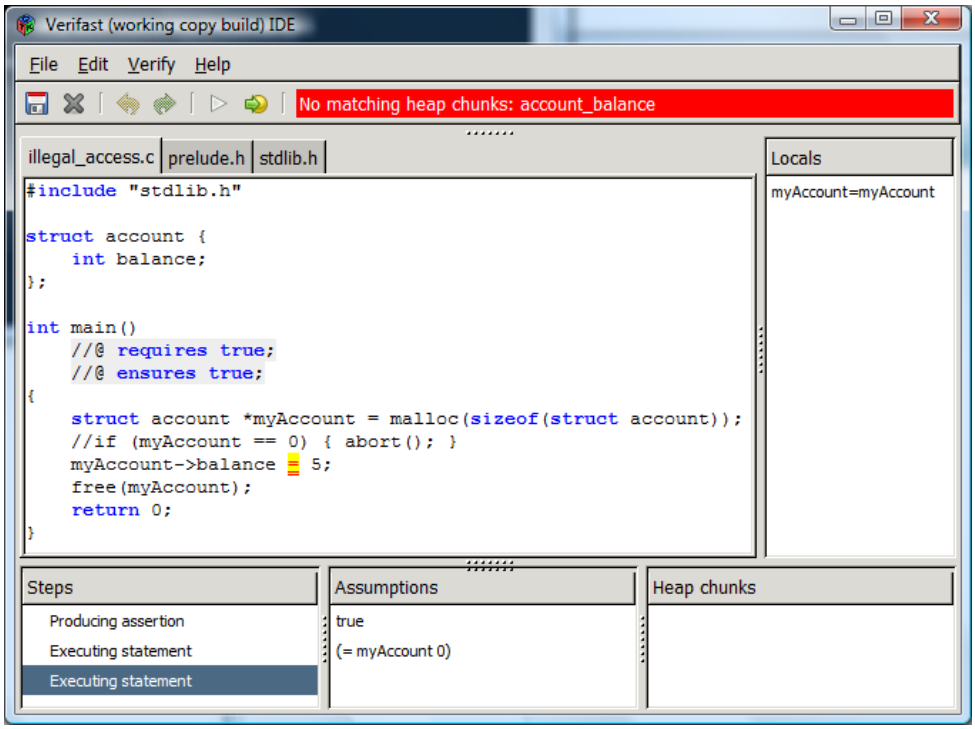


図 6.1: illegal_access.c を開いた VeriFast IDE スクリーンショット

このプログラムは *malloc* を使って確保した構造体インスタンス `myAccount` のフィールド `balance` にアクセスしようと試みます。けれども、もしメモリが不足していたら、*malloc* はゼロを返してメモリは確保されません。 VeriFast はこのような場合に起きる不正なメモリアクセスを検出します。次の GUI の要素に注意してください:

- 誤ったプログラムの要素が二重下線をともなって赤色で表示されます。
- “No matching heap chunks: account_balance” というエラーメッセージが表示されます。実際メモリが不足した場合には、プログラムがアクセスを試みるメモリ位置 (もしくは **ヒープチャンク** (*heap chunk*)) はアクセス可能ではありません。`account_balance` は、構造体 `account` インスタンスの `balance` フィールドを表わすヒープチャンクの型です。
- 代入文が黄色バックグラウンドで表示されています。これはその代入文が**現在のステップ** (*current step*) であるためです。関係するプログラム状態のシンボリック表現のトラックを保持しながら、そこへのステップで VeriFast はそれぞれの関数を検証します。VeriFast ウィンド

ウの左下コーナーの Steps ペインでステップを選択することで、それぞれのステップでのシンボリック状態を検査できます。その現在のステップに相当するプログラムの要素は黄色バックグラウンドで表示されます。シンボリック状態は、Assumptions ペインに表示された **パスコンディション** (*path condition*) と; Heap chunks ペインに表示された **シンボリックヒープ** (*symbolic heap*) と; Locals ペインに表示された**シンボリックストア** (*symbolic store*) で構成されます。

このエラーを修正するために、コメント化された文のコメントをはずしてください。そして再び F5 を押してください。バーは緑色になるでしょう; これでこのプログラムは検証されました。これは VeriFast が `main` 関数を実行する全てのパスをシンボリックに実行し、かつエラーが見つからなかったことを意味しています。

VeriFast がこの関数をどのようにシンボリックに実行するのか良く調べてみましょう。VeriFast プログラムをシンボリックに実行した後、Trees ペインにそれぞれの関数に対する **シンボリック実行木** (*symbolic execution tree*) を見ることができます。この Trees ペインはデフォルトでは隠されていますが、VeriFast ウィンドウの右端の縁を左にドラッグすることで現われます。 Trees ペインの上部は、シンボリックに実行された全ての関数のドロップダウンリストです。`main` 関数のシンボリック実行木を見るために、*Verifying function 'main'* を選択してください。

シンボリック実行木は 3 種類のノードを持ちます:

- **先頭ノード** (*top node*) はシンボリック実行の開始を表わします。先頭ノードをクリックしてください: シンボリック実行の初期状態では、ヒープチャンクは存在せず (Heap chunks ペインは空です)、ローカル変数も存在せず (Locals ペインは空です)、かつ仮定も存在しません (Assumptions ペインは空です)。
- シンボリック実行パスが 2 つのパスに分れる点に、それぞれ 1 つの**分岐ノード** (*branch node*) があります。シンボリック実行で複数の場合が必要とされる時にこの分岐が発生します; そのためこれは **場合分け** (*case split*) とも呼ばれます。`main` 関数のシンボリック実行は 1 つの場合分けを引き起こします: `malloc` 呼び出しのシンボリック実行は、メモリ不足のために `malloc` が NULL ポインタを返す分岐と、メモリに空きがあるため `malloc` が要求された量のメモリを確保して、そのポインタを返す分岐とに分かれます。(`if` 文をシンボリックに実行する時にも場合分けが起きますが、`if` 文の 2 つの場合と `malloc` 文の 2 つの場合は同一なので、別々の分岐ノードとして表示されません。)
- 関数を通るシンボジック実行パスの完了する端点に、それぞれ 1 つの**葉ノード** (*leaf node*) があります。どれか葉ノードをクリックすると、Steps ペインに完全なシンボリック実行パスが見えるでしょう。`main` 関数は 2 つのシンボリックパスを持ちます: あるパスはメモリ不足のため `abort` 呼び出しによってプログラムが終了した時に終了します; もう 1 つのパスはメモリに空きがあるためその関数が返るときに終了します。

右端の葉ノードをクリックして、`malloc` がメモリ確保に成功する実行パスを表示させましょう。このとき VeriFast が `main` 関数のコード左端の余白に矢印を表示することに注意してください。この矢印はこのパスが `malloc` 文の 2 番目の場合と `if` 文の 2 番目の場合を実行することを表わしています。

VeriFast のシンボリック実行の詳細について理解を深めるために、上からこのパスを一歩ずつ見てみましょう。Steps ペインの最初のステップを選択してください。それから ↓ 方向キーを押してください。 *Verifying function `main`* ステップはシンボリック状態に影響を与えません。*Producing assertion* ステップは仮定 *true* を Assumptions に追加します。このチュートリアルの後で詳細に表明の生成と消費を考察することにします。これで `malloc` 文に対する *Executing statement* ステップに

辿り着きました。この文はシンボリック状態に 3 つの影響を与えます:

- ヒープチャンク *account_balance(myAccount, value)* と *malloc_block_account(myAccount)* をシンボリックヒープに追加します。(これらは Heap chunks ペインに表示されます。) このとき、*myAccount* と *value* は未知の値を表わす **シンボル** (*symbols*) です。具体的には、*myAccount* は新しく確保した構造体インスタンスのメモリアドレスを表わし、*value* はその構造体インスタンスの `balance` フィールドの初期値を表わします。(Java では新しいオブジェクトのフィールドはその型のデフォルト値で初期化されますが、C 言語では新しく確保された構造体インスタンスのフィールドの初期値は不定値です。) VeriFast はシンボリック実行のステップでこれらのシンボルを**新しく採取します**。すなわち、新しい構造体インスタンスのアドレスと `balance` フィールドの初期値を表わすために、VeriFast はまだこのシンボリック実行パスで使われていないシンボルを使うのです。これを調べるために、以下に示す関数 `test` を検証してみてください。すると 1 番目の `malloc` 文をシンボリックに実行するために VeriFast はシンボル *myAccount* と *value* を採取し、次に 2 番目の `malloc` 文をシンボリックに実行ためにシンボル *myAccount0* と *value0* を採取したことに注意してください。

```
void test()
    //@ requires true;
    //@ ensures true;
{
    {
        struct account *myAccount = malloc(sizeof(struct account));
        if (myAccount == 0) { abort(); }
    }
    {
        struct account *myAccount = malloc(sizeof(struct account));
        if (myAccount == 0) { abort(); }
    }
}
```

- 仮定 *myAccount* $\neq$ *0* (たぶん異なる表記になるでしょう) をパスコンディションに追加します。(これは Assumptions ペインに表示されます。) 実際、もし `malloc` に成功したら、返り値のポインタは NULL ではありません。
- シンボリックストア (Locals ペインに表示されます) に、ローカル変数 `myAccount` からシンボリック値 *myAccount* への束縛を追加します。実際、このプログラムは (シンボル *myAccount* で表わされた) `malloc` 呼び出しの返り値をローカル変数 `myAccount` に代入します。この場合ではローカル変数とシンボルは偶然にも同じ名前を持つのは偶然で、特殊な意味を持たないことに注意してください。

図 6.3 は成功した場合の `malloc` 文のシンボリック実行を要約しています。図 6.2 は失敗した場合を要約しています。

このシンボリック実行トレースの次のステップは `if` 文のシンボリック実行です。 2 つの場合が考えられるという点で、`if` 文は `malloc` 文に似ています; そのため `if` 文に対しても、VeriFast は場

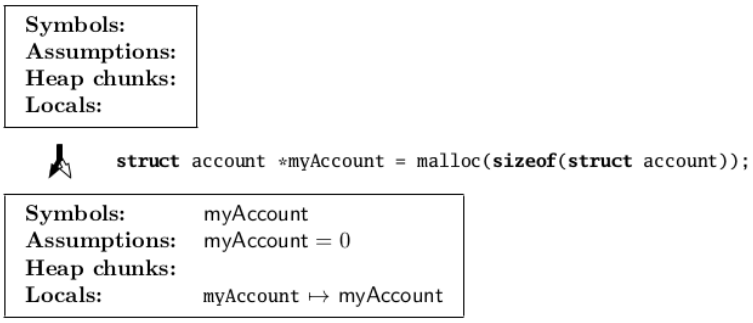


図 6.2: malloc 文のシンボリック実行 (1 つ目の場合)

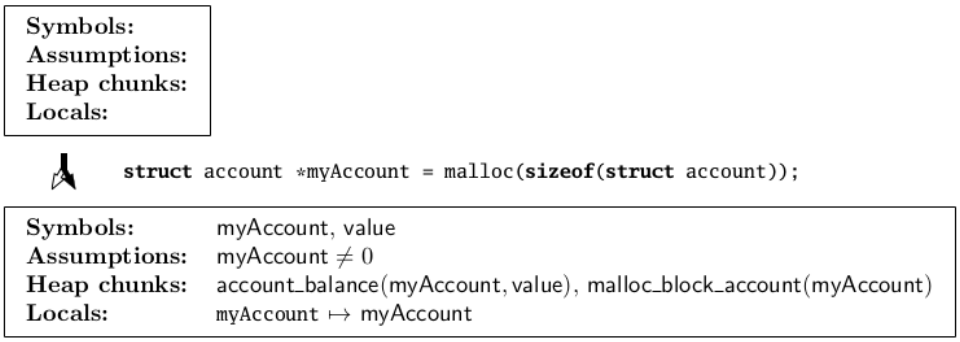


図 6.3: malloc 文のシンボリック実行 (2 つ目の場合)

合分けを行ない、シンボリック実行パスを 2 つの分岐に分けます。1 つ目の分岐では、VeriFast は `if` 文の条件は真である場合だと見なします。さらにこの場合の仮定をパスコンディションに追加し、`if` 文の *then* ブロックをシンボリックに実行します。2 つ目の分岐では、VeriFast は `if` 文の条件が偽である場合だと見なします。さらに一致する仮定をパスコンディションに追加して、*else* ブロックをシンボリックに実行します。仮定をパスコンディションに追加した後、VeriFast は結果のパスコンディションに矛盾が検出されるか常にチェックすることに注意してください; もし矛盾があったら、シンボリック実行パスはどのような実際の実行パスにも相当しません。そのためそのパスのシンボリック実行を継続せず、VeriFast はその実行を放棄します。これは `malloc` が成功した後の `if` 文の 1 つ目の分岐で起きることです; さらにそれは `malloc` に失敗した後の `if` 文の 2 番目の分岐でも発生します。

図 6.4 と図 6.5 は `if` 文のシンボリック実行の 2 つの場合を要約しています。

このシンボリック実行トレースの次のステップは、新しく確保した構造体インスタンスの `balance` フィールドに値 5 を代入する文をシンボリックに実行します。構造体インスタンスのフィールドを代入するとき、最初に VeriFast は当該構造体インスタンスのフィールドを表わすヒープチャンクがシンボリックヒープに存在するかチェックします。もし存在しなければ、検証に失敗し “No such heap chunk” とレポートします。これはそのプログラムがまだ確保されていないメモリにアクセスしようとしたことをおそらく意味しています。もしそのチャンクが存在したら、VeriFast はそのチャンクの 2 番目の引数を代入の右辺の値に置き換えます。これは図 6.6 のようになるでしょう。

最後に、`free` 文のシンボリック実行は `malloc` 文で追加された 2 つのヒープチャンク (`balance` フィールドを表わすチャンクと malloc_block チャンク) がまだシンボリックヒープに存在するか

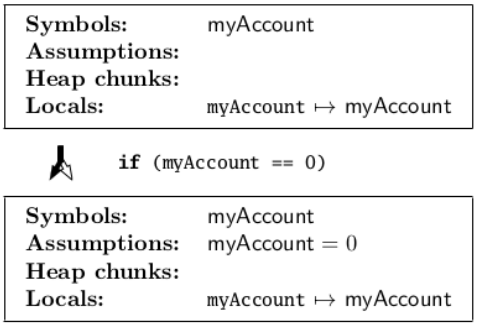


図 6.4: if 文のシンボリック実行 (1 つ目の場合)。シンボリック実行は if 文の them ブロックを実行します。

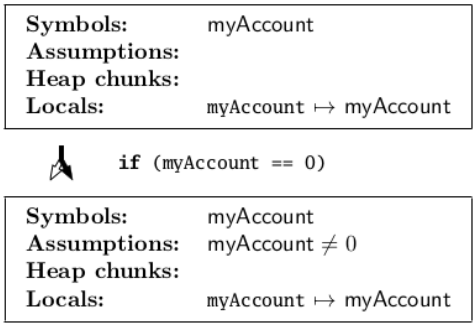


図 6.5: if 文のシンボリック実行 (2 つ目の場合)。シンボリック実行は if 文の else ブロックを実行します。

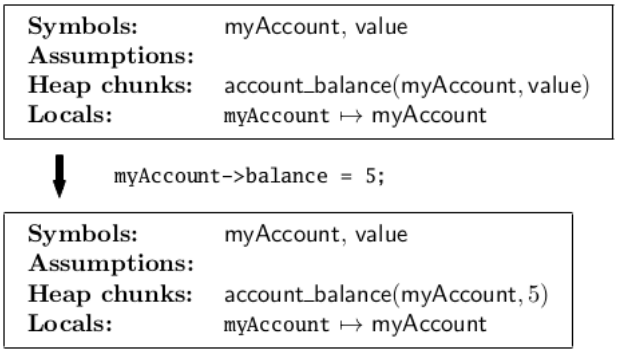


図 6.6: 構造体フィールドへの代入文のシンボリック実行

チェックします。もし存在しなければ、VeriFast は検証に失敗します; そのプログラムは既に解放済みの構造体インスタンスを解放しようとしているのかもしれません。そうでなければ、図 6.7 に示すようにそのチャンクを削除します。これはもしプログラムが構造体インスタンスを解放してからその構造体インスタンスのフィールドにアクセスしようと試みたら、そのフィールドにアクセスする文のシンボリック実行が失敗することを保証します。(なぜならそのフィールドを表わすヒープチャンクが見つからないからです。)

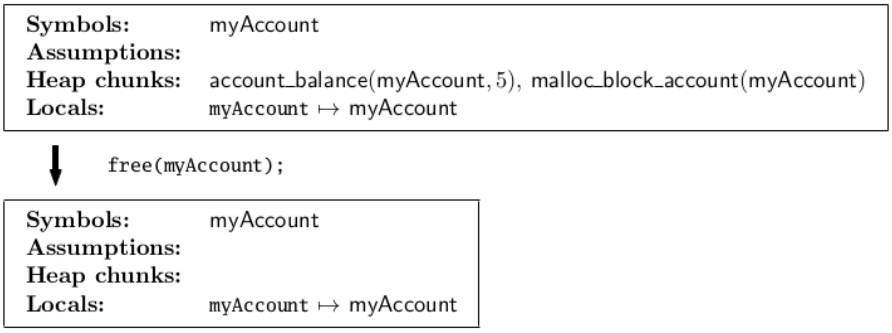


図 6.7: free 文のシンボリック実行

6.3 malloc_block チャンク

なぜ *malloc* 文が `account_balance` チャンクと `malloc_block_account` チャンクの両方を生成するのかをより理解するために、ヒープに確保される代わりにスタック上のローカル変数として構造体インスタンスが確保されるようにプログラムを変更しましょう:

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct account myAccountLocal;
    struct account *myAccount = &myAccountLocal;
    myAccount->balance = 5;
    free(myAccount);
    return 0;
}
```

はじめに、このプログラムはスタックに構造体 `account` のインスタンスを確保し、それを `myAccountLocal` と呼びます。それからこの構造体インスタンスのアドレスをポインタ値 `myAccount` に代入します。プログラムの残りは以前と同じです: プログラムは `balance` フィールドを値 5 に初期化し、それからその構造体インスタンスを解放しようと試みます。

もし VeriFast にこのプログラムを検証させると、VeriFast は *free* 文で次のエラーをレポートします:

```
No matching heap chunks: malloc block account(myAccountLocal addr)}
```

実際この *free* は正しくありません。なぜなら *free* は `malloc` でヒープに確保された構造体インスタンスにのみ適用され、ローカル変数としてスタックに確保された構造体インスタンスには適用できないからです。

VeriFast はこのエラーを次のように検出します: 1) VeriFast は *malloc* を使って確保された構造体インスタンスにのみ `malloc_block` チャンクを生成します。このチャンクはスタックに確保された構造体インスタンスを表わしません。2) *free* 文を検証するとき、その構造体が解放されるために VeriFast は `malloc_block` チャンクが存在することをチェックします。

その一方で、`account_balance` チャンクはどちらの場合も生成されることに注意してください。結果的に、構造体インスタンスがヒープに確保されたかスタックに確保されたかどうかにかかわらず、`balance` フィールドを初期化する文は検証に成功します。

6.4 関数と契約

前の章の例からはじめます。今の時点ではこの例は唯一 1 つの関数から成ります: それは main 関数です。別の関数を追加してみましょう。 `account` 構造体インスタンスのアドレスと整数の金額を取り、その金額を構造体インスタンスの `balance` フィールドに代入する関数 `account_set_balance` を書きます。そして main 関数のフィールド代入をこの関数呼び出しで置き換えます。プログラムは以下のようになります:

```
#include "stdlib.h"

struct account {
    int balance;
};

void account_set_balance(struct account *myAccount, int newBalance)
{
    myAccount->balance = newBalance;
}

int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = malloc(sizeof(struct account));
    if (myAccount == 0) { abort(); }
    account_set_balance(myAccount, 5);
    free(myAccount);
    return 0;
}
```

新しいプログラムを検査すると、VeriFast はこの新しい関数が契約を持っていないとエラーを出します。実際、VeriFast はそれぞれの関数を別々に検査します。そのため、それぞれの関数に関数呼び出しの初期状態と最終状態を表現する事前条件と事後条件が必要です。

main 関数が持つのと同じ契約を追加しましょう:

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires true;
    //@ ensures true;
```

全ての VeriFast 注釈と同様に、契約はコメント中に置かれるので、C 言語コンパイラはそれらを無視することに注意してください。(@) 記号でマークされたコメントを除いて、VeriFast もコメントを無視します。

これでもはや VeriFast は契約の欠落に関するエラーを出さなくなりました。けれども今度は、account_set_balance 本体のフィールド代入が検査できないというエラーが出ます。なぜなら、そのシンボリックヒープはこのフィールドへのアクセスを許可するヒープチャンクを含まないからです。これを修正するために、関数の事前条件で、その関数がアドレス myAccount の account 構造体インスタンスの balance フィールドへのアクセスに対する許可を要求することを明記しなければなりません。事前条件にヒープチャンクを書くことで、この修正を行ないましょう:

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires account_balance(myAccount, _);
    //@ ensures true;
```

フィールドの値がある位置にアンダースコアを使っていることに注意してください。これは、この関数が呼び出された時に、そのフィールドの古い値について関心がないことを示しています。(また VeriFast はフィールドチャンクに対してより簡潔な構文をサポートしています。例えば、account_balance(myAccount, _) は myAccount->balance |-> _ と書くこともできます。実際には一般に、後者の (フィールドチャンク固有の) 構文は前者の (総称チャンク) 構文よりも推奨されています。なぜならそれは VeriFast に、与えられたフィールドを表わすチャンクが最大 1 つあり、そのフィールドの値がその型の制限内であるというフィールドチャンク固有の情報を考慮させるからです。けれども、注釈に書かれたチャンクと VeriFast IDE で表示されるヒープチャンクが同じ見た目になるように、このチュートリアルでは初めは総称チャンク構文を使います。)

これで VeriFast は関数本体を閉じる括弧をハイライトします。これはフィールド代入の検証に成功したことを意味しています。けれども、VeriFast はこの関数がヒープチャンクをリークしているというエラーを表示します。さしあたって単純に、このヒープチャンクのリークを許すことを示す leak コマンドを挿入してこのエラーメッセージを回避しましょう。後にこの問題に戻ってくることにします。

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires account_balance(myAccount, _);
    //@ ensures true;
{
    myAccount->balance = newBalance;
    //@ leak account_balance(myAccount, _);
}
```

これで関数 account_set_balance は検査され、VeriFast は関数 main を検査しようと試みます。

この検査は account 構造体インスタンスを解放できないというエラーを出力します。なぜなら、balance フィールドへのアクセス許可を持たないからです。実際、そのシンボリックヒープは malloc_block_account チャンクを含みますが、account_balance チャンクを含みません。何が起きたのでしょうか？ シンボリック実行パスをステップ実行して調べてみましょう。2 番目のステップを選択します。 malloc 文が実行されようとしており、そのシンボリックヒープは空です。次のステップを選択します。 malloc 文が account_balance チャンクと malloc_block_account チャンクを追加しました。

if 文は作用がありません。

そうして account_set_balance を呼び出します。この実行ステップは “Consuming assertion” と “Producing assertion” という名前の 2 つの子ステップを持つことに気が付くでしょう。関数呼び出しの検証は、関数の事前条件の **消費** (*consuming*) とその後の関数の事後条件の **生成** (*producing*) から成ります。事前条件と事後条件は **表明** (*assertions*)、すなわち通常の論理に加えてヒープチャンクを含むような式です。事前条件の消費は、その関数から要求されたヒープチャンクをその関数に渡すことを意味し、従ってそれらをシンボリックヒープから削除します。事後条件の生成は、その関数が返るときにその関数から提示されたヒープチャンクを受け取ることを意味し、従ってそれらをシンボリックヒープに追加します。

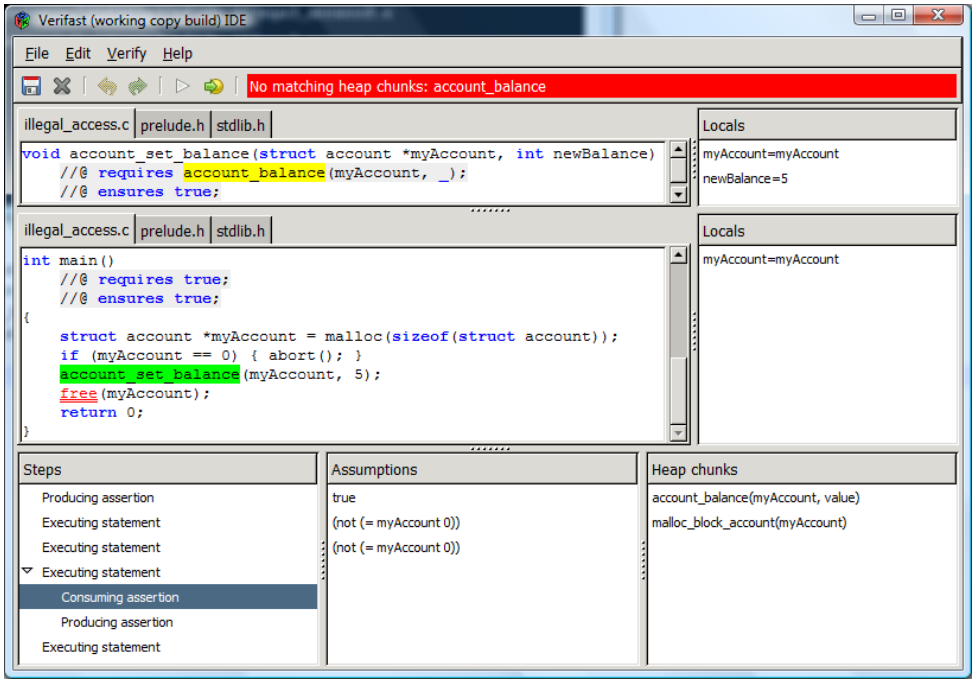


図 6.8: 関数呼び出しをステップ実行したとき、VeriFast は (下のペインに緑色で) 呼び出し元と (上のペインに黄色で) 呼び出された契約の両方を表示

“Consuming assertion” ステップを選択すると VeriFast ウィンドウのレイアントが変わります (図 6.8 を見てください)。ソースコードペインが 2 つの部分に分かれます。上の部分は呼び出された関数の契約を表示するのに使われ、下の部分は検査される関数を表示するのに使われます。(この例では呼び出される関数は検査される関数と近接しているので、上と下のパートは同じものが表示されます。) 検査される呼び出しが緑色バックグラウンドで表示されます。消費/生成される契約は黄

色バックグラウンドで表示されます。“Consuming assertion” ステップから “Producing assertion” ステップに移ると、“Consuming assertion” ステップがシンボリックヒープから account_balance チャンクを削除することに気が付くでしょう。ここでは概念上、main 関数が account_set_balance 関数の返りを待つ間、それはこの関数によって使われます。関数 account_set_balance の事後条件はヒープチャンクに言及していないので、“Producing assertion” ステップはシンボリックヒープに何も追加しません。

ここでは VeriFast が account_set_balance がヒープチャンクをリークしたとエラーを出すのかは明確です: この関数は account_balance チャンクをその呼び出し元に返さないため、そのチャンクは失われ、そのフィールドは再びアクセスすることができないのです。これは通常プログラマの意図ではないので、VeriFast はこれをエラーとみなします; さらに、より多くのメモリ位置がリークしたら、そのプログラムはメモリ不足になるでしょう。

このエラーを修正する方法も明確です: 関数 account_set_balance の事後条件にこの関数が account_balance チャンクを呼び出し元に返すことを指定すべきです。

```
void account_set_balance(struct account *myAccount, int newBalance)
    //@ requires account_balance(myAccount, _);
    //@ ensures account_balance(myAccount, newBalance);
{
    myAccount->balance = newBalance;
}
```

これはリークエラーメッセージと *free* 文のエラーを取り除きます。これでこのプログラムは検査されました。フィールドの値が属する位置で newBalance パラメータを参照していることに注意してください; これは関数が返る時のこのフィールドの値がこのパラメータの値と等しいことを意味しています。

練習問題 1 account 構造体インスタンスの生成と破棄を関数に切り出してください。その生成関数は balance をゼロで初期化すべきです。注意: 表明で複数のヒープチャンクを使うには、それらを分離積 &*& (アンパサンド-スター-アンパサンド) を使って区切ってください。また事後条件では関数の返り値を result の名前で参照できます。

6.5 パターン

現在の balance を返す関数を追加し、それを main 関数で使ってみましょう。次のコードが最初の試みです:

```
int account_get_balance(struct account *myAccount)
    //@ requires account_balance(myAccount, _);
    //@ ensures account_balance(myAccount, _);
{
    return myAccount->balance;
}

int main()
    //@ requires true;
```

```
    //@ ensures true;
{
    struct account *myAccount = create_account();
    account_set_balance(myAccount, 5);
    int b = account_get_balance(myAccount);
    assert(b == 5);
    account_dispose(myAccount);
    return 0;
}
```

新しい関数の検査は成功しますが、VeriFast は条件 `b == 5` を証明できないというエラーを返します。 VeriFast は条件のチェックを要求されると、はじめに、それぞれの変数をシンボリック値で置換することで、その条件を論理式に変換します。Locals ペインに表示されたシンボリックストアに、変数 `b` のシンボリック値が論理記号 `b` であることを見つけることができます。そのため、結果の論理式は `b == 5` です。それから VeriFast は **パスコンディション**、すなわち Assumptions ペインに表示されている式、からこの式を導出しようと試みます。この場合唯一の仮定は **true** なので、VeriFast はこの条件を証明できないのです。

この問題は関数 `account_get_balance` の事後条件はこの関数の返り値を指定していないことによります。その返り値が関数が呼ばれたときの `balance` フィールドの値と等しいと言明していないのです。これを修正するために、関数が呼ばれたときの `balance` フィールドの値に名前を代入できるようにする必要があります。これは事前条件のアンダースコアを **パターン** `?theBalance` で置換することで可能です。これで名前 `theBalance` は `balance` フィールドの値に束縛されるようになります。この名前は事後条件中で、等価条件を使う返り値を指定するために使うことができます。関数の返り値はその関数の事後条件で `result` の名前で使えます。表明における論理条件とヒープチャンクは分離積 `&*&` を使って区切ってください。

```
int account_get_balance(struct account *myAccount)
    //@ requires account_balance(myAccount, ?theBalance);
    //@ ensures account_balance(myAccount, theBalance) &*& result == theBalance;
{
    return myAccount->balance;
}
```

事後条件のフィールド値の位置で使うことで、名前 `theBalance` はその関数が `balance` フィールドの値を修正しないことを指定するのにも使えることに注意してください。

これでこのプログラムは検査されました。実際、**Run to cursor** コマンドを使って assert 文まで実行すると、Assumptions ペインに仮定 `(= b 5)` が現われるでしょう。ステップを戻すと、関数 `account_get_balance` の事後条件における等式条件が生成されたときに、その仮定が追加されたことが分かるでしょう。さらにステップを戻すと、フィールドチャンクの表明が消費されたときに、上の Locals ペインに変数 `theBalance` が追加され、その値が 5 に束縛されたことが分かるでしょう。シンボリックヒープに値が見つかったので、それは値 5 に束縛されます。関数呼び出しを検査するとき、上の Locals ペインはその呼び出される関数の契約を評価するのに使われます。初期状態ではそれはその関数のパラメータからその呼び出しで指定された引数への束縛を含んでいます; 追

加の束縛は契約にパターンを記述すると現われます。仮定 (= b 5) は、上の Locals ペインで示されたシンボリックストアで、等式条件 result == theBalance を評価して得られる論理式です。

練習問題 2 与えられた金額を account に預ける関数を追加してください。次の main 関数を検証してください。

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = create_account();
    account_set_balance(myAccount, 5);
    account_deposit(myAccount, 10);
    int b = account_get_balance(myAccount);
    assert(b == 15);
    account_dispose(myAccount);
    return 0;
}
```

注意: VeriFast は算術オーバーフローをチェックします。ここでは、**Verify** メニューでこのチェックを無効にしてください。

練習問題 3 account の最小 balance を指定するために、構造体 account にフィールド limit を追加してください。(これは一般にゼロもしくは負の数になります。) この limit は生成時に指定します。さらに与えられた金額を account から引き出す関数を追加してください。この関数は limit を配慮しなければなりません; 要求された金額の引き出しが limit に違反したら、limit の違反を除いて引き出し可能な最大の金額が引き出されます。その関数は実際に引き出された金額をその返り値として返します。C 言語の条件式 condition ? value1 : value2 を使うことになるでしょう。関数 account_set_balance を削除してください。フィールドチャンク myAccount->balance |-> value の略記を使ってください。次の main 関数を検査してください。

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct account *myAccount = create_account(-100);
    account_deposit(myAccount, 200);
    int w1 = account_withdraw(myAccount, 50);
    assert(w1 == 50);
    int b1 = account_get_balance(myAccount);
    assert(b1 == 150);
    int w2 = account_withdraw(myAccount, 300);
    assert(w2 == 250);
    int b2 = account_get_balance(myAccount);
```

```
    assert(b2 == -100);
    account_dispose(myAccount);
    return 0;
}
```

6.6 述語

練習問題 3 で得られたプログラムからはじめましょう。先の契約が成長するのを観察しました。さらに account の “クラス” と、異なるモジュールである main 関数を考えると、account モジュールの内部実装詳細が main 関数に露出されてしまっています。関数の契約に account 構造体インスタンスを表現する **述語** (*predicate*) を導入することで、情報の隠蔽とより簡潔な契約を実現できます。

```
/*@
predicate account_pred(struct account *myAccount, int theLimit, int theBalance)
  = myAccount->limit |-> theLimit &*& myAccount->balance |-> theBalance
    &*& malloc_block_account(myAccount);
@*/
```

述語は名前の付いたパラメータ化された表明です。さらにそれはヒープチャンクの新しい型を導入します。account_pred ヒープチャンクは、account_limit ヒープチャンク、account_balance ヒープチャンクと malloc_block_account ヒープチャンクを 1 つにまとめています。

この述語を使って、引き出し関数の契約を書き換えてみましょう。最初の試みは次のようになります:

```
void account_deposit(struct account *myAccount, int amount)
    //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
    //@ ensures account_pred(myAccount, limit, balance + amount);
{
    myAccount->balance += amount;
}
```

この関数は検査できません。シンボリックヒープに account_balance ヒープチャンクがないので、balance フィールドの更新は検査できないのです。そこには account_pred ヒープチャンクしかありません。 account_pred ヒープチャンクは account_balance ヒープチャンクをカプセル化していますが、VeriFast は述語 account_pred を自動的に分解しません。 **open** ゴースト文を挿入して、VeriFast に述語ヒープチャンクを分解するよう指示する必要があります:

```
void account_deposit(struct account *myAccount, int amount)
    //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
    //@ ensures account_pred(myAccount, limit, balance + amount);
{
    //@ open account_pred(myAccount, limit, balance);
```

```
    myAccount->balance += amount;
}
```

これでこの代入は検査されましたが、VeriFast は事後条件で停止します。それは関数の呼び出し元へ返す account_pred ヒープチャンクが見つからないと訴えています。 account_pred チャンクを構成するチャンクはシンボリックヒープ中に存在しますが、VeriFast は自動的にそれらを account_pred チャンクにまとめません。**close** ゴースト文を使ってそれを行なうよう VeriFast に指示する必要があります:

```
void account_deposit(struct account *myAccount, int amount)
    //@ requires account_pred(myAccount, ?limit, ?balance) &*& 0 <= amount;
    //@ ensures account_pred(myAccount, limit, balance + amount);
{
    //@ open account_pred(myAccount, limit, balance);
    myAccount->balance += amount;
    //@ close account_pred(myAccount, limit, balance + amount);
}
```

これでこの関数は検査されます。けれども、account_deposit 呼び出しが account_pred ヒープチャンクを期待しているため、main 関数は検査できません。

練習問題 4 述語 account_pred を使って残りの契約を書き換えてください。必要に応じて文 **open** と **close** を挿入してください。

6.7 再帰的な述語

前章では、簡潔さと情報の隠蔽のために、述語を導入しました。けれども、述語に対してさらなる切実な要請があります: それは VeriFast でサイズに際限のないデータ構造を表現する唯一の方法なのです。実際、述語がない場合、表明で表現されるメモリ位置の数はその表明の長さに比例します。この制限は再帰的な述語、すなわち自分自身を呼び出す述語、を使うことで克服できます。

練習問題 5 片方向リストデータ構造を使って整数のスタックを実装してください: 関数 create_stack, stack_push, stack_pop, stack_dispose を実装してください。 stack_pop の事前条件を指定できるようにするために、述語はスタック中の要素の数を指定するパラメータを持つ必要があります。関数 stack_dispose は空のスタックにのみ呼び出すことができます。スタックの中身を指定しないでください; これまで見てきた注釈ではそれは不可能です。条件付き表明 condition ? assertion1 : assertion2 を使う必要があります。注意: **open** 文中でのフィールドのデリファレンスを VeriFast は許しません。 **open** 文中でフィールドの値を使いたい場合、はじめにその値をローカル変数に保存しなければなりません。次の main 関数を検査してください:

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
```

```
    stack_push(s, 10);
    stack_push(s, 20);
    stack_pop(s);
    stack_pop(s);
    stack_dispose(s);
    return 0;
}
```

ここで、練習問題 5 の答を拡張して stack_is_empty 関数を作ってみましょう。述語定義を思い出してみましょう:

```
predicate nodes(struct node *node, int count) =
    node == 0 ?
        count == 0
    :
        0 < count
        &*& node->next |-> ?next &*& node->value |-> ?value
        &*& malloc_block_node(node) &*& nodes(next, count - 1);

predicate stack(struct stack *stack, int count) =
    stack->head |-> ?head &*& malloc_block_stack(stack) &*& 0 <= count &*&
    nodes(head, count);
```

次は stack_is_empty 関数の最初の試みです:

```
bool stack_is_empty(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == (count == 0);
{
    //@ open stack(stack, count);
    bool result = stack->head == 0;
    //@ close stack(stack, count);
    return result;
}
```

この関数は検査できません。 VeriFast は事後条件の条件式 result == (count == 0) を証明できないとエラーを吐きます。実際、Assumptions ペインの仮定を見てみると、それらの仮定はこの条件を証明するのに不十分です。この問題は head ポインタの値と nodes の数との関係が述語 nodes の中に隠されてしまっていることです。その情報を仮定に追加するために、その述語を開く必要があります。もちろん、述語 stack を閉じられるように、その後その述語を再び閉じなければなりません。

```
bool stack_is_empty(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == (count == 0);
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ open nodes(head, count);
    bool result = stack->head == 0;
    //@ close nodes(head, count);
    //@ close stack(stack, count);
    return result;
}
```

これでこの関数は検査できます。正確に起きることは次のようなものです。VeriFast が open 文を実行すると、述語 nodes の本体中の条件付き表明を生成します。これは **場合分け** (*case split*) の実行を引き起こします。これはその関数の残りが二度検証されることを意味しています: 一度は条件が真である仮定において、もう一度は条件が偽である仮定においてです。別の言い方をすると、その実行パスは 2 つの実行パス、もしくは **分岐** (*branches*) に分かれるのです。これで両方の分岐において、事後条件はたやすく証明できます: 1 番目の分岐では仮定 `head == 0` と `count == 0` を得て、2 番目の分岐では `head != 0` と `0 < count` を得ます。

練習問題 6 関数 `stack_dispose` を修正して、スタックが要素を含んでいても動くようにしてください。再帰的なヘルパー関数を使ってください。(警告: VeriFast は停止性を検証しません; 無限再帰や無限ループをエラーにしないのです。それはあなた自身の責務です。)
if 文を検証するとき、VeriFast は場合分けを行なうことに注意してください。

練習問題 7 スタックの要素の値の合計を返す関数 `stack_get_sum` を追加してください。再帰的なヘルパー関数を使ってください。(まだどうやってするのか分からないので) 契約は返り値を指定する必要はありません。

6.8 ループ

練習問題 6 では、再帰関数を使って `stack_dispose` を実装しました。けれども、これは最善の実装ではありません。データ構造がとても多くの要素を含んでいると、あまりに多くのスタックフレームを生成してコールスタックがあふれてしまうかもしれません。より良い実装はループを使った関数です。次のコードは最初の試みです:

```
void stack_dispose(struct stack *stack)
    //@ requires stack(stack, _);
    //@ ensures true;
{
    //@ open stack(stack, _);
    struct node *n = stack->head;
    while (n != 0)
```

```
    {
        //@ open nodes(n, _);
        struct node *next = n->next;
        free(n);
        n = next;
    }
    //@ open nodes(0, _);
    free(stack);
}
```

この関数は検証できません。このループはループ不変条件を指定していないので、VeriFast はこのループでエラーになります。 VeriFast がループ不変条件を必要としているのは、(最初の反復のみではなく) 任意のループ反復の開始を表わすシンボリック状態からはじめて、ループ本体を一度検証することでループ反復の任意の順序を検証できるようにするためです。

具体的には VeriFast はループを次のように検証します:

- はじめに、ループ不変条件を消費します。
- それから、ヒープから残りのヒープチャンクを削除します (しかしそれらは記憶されています)。
- それから、ループ本体で変更されたそれぞれのローカル変数に新鮮な論理記号を代入します。
- それから、ループ不変条件を生成します。
- それから、ループ条件の場合分けを行ないます:
  - もし条件が真なら:
    - ループ本体を検証し、
    - それから、ループ不変条件を消費し、
    - それから最後に、リークをチェックします。このステップの後、この実行パスは終了します。
  - もし条件が偽なら、VeriFast はステップ 2 で削除されたヒープチャンクをヒープに戻し、それからループの後から検証を続けます。

これは、ループがそのループ不変条件中で言及されたヒープチャンクのみアクセス可能であることを意味することに注意してください。

上記の関数に対する正しいループ不変条件は次のようになります:

```
void stack_dispose(struct stack *stack)
    //@ requires stack(stack, _);
    //@ ensures true;
{
    //@ open stack(stack, _);
    struct node *n = stack->head;
    while (n != 0)
        //@ invariant nodes(n, _);
    {
        //@ open nodes(n, _);
        struct node *next = n->next;
```

```
        free(n);
        n = next;
    }
    //@ open nodes(0, _);
    free(stack);
}
```

ループ本体の閉括弧にカーソルを置いて **Run to cursor** コマンドを選択することで、そのループの実行の 1 番目の分岐を調査できます。**while** 文を表わす *Executing statement* ステップを見つけてください。このステップが *Producing assertion* ステップと *Consuming assertion* ステップの前にあることに注意してください。これら 2 つのステップの間に、変数 `n` の値が `head` から `n` に変化し、全てのチャンクがシンボリックヒープから削除されることに注意してください。さらにその次のステップで仮定 `(not (= n 0))` が Assumptions ペインに追加されることに気付いてください。

関数本体の閉括弧にカーソルを置いて **Run to cursor** コマンドを選択することで、そのループの実行の 2 番目の分岐を調査できます。そのループ本体が実行されないことと、仮定 `(= n 0)` が Assumptions ペインに追加されることを除いて、1 番目の分岐と同じことが起きていることに注意してください。

練習問題 8 与えられた数の要素をスタックからポップして **void** を返す関数 `stack_popn` を実装してください。その内部から `stack_pop` を呼び出すことができます。**while** ループを使ってください。事後条件を証明するために、ループ不変条件はそのループ本体の検証を有効にするだけでなく、現在状態と初期状態の関係も維持しなければならないことに注意してください。これはしばしば関数パラメータの値を上書きしてはならないことを意味しています。なぜなら一般にループ不変条件では元の値が必要になるからです。

6.9 帰納データ型

(練習問題 5 の回答である) 最初に注釈したスタック実装に戻ってみましょう。この注釈は関数的な正しさを十分に明記していません。特に、関数 `stack_pop` の契約はその関数の返り値を指定してません。結果としてこれらの注釈を使うと、次のような main 関数を検証できません:

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
    int result1 = stack_pop(s);
    assert(result1 == 20);
    int result2 = stack_pop(s);
    assert(result2 == 10);
    stack_dispose(s);
    return 0;
```

```
}
```

この main 関数を検証するためには、スタック中の要素の数を単に追跡する代わりに、要素の値も同様に追跡する必要があります。別の言い方をすると、スタックに現状保存されている要素の正確な列を追跡する必要があるのです。**帰納データ型** (*inductive datatype*) `ints` を使うことで、整数の列を表現できます:

```
inductive ints = ints_nil | ints_cons(int, ints);
```

この宣言は 2 つの **コンストラクタ** (*constructors*) `ints_nil` と `ints_cons` を共なう型 `ints` を宣言しています。 `ints_nil` は空の列を表わします。`ints_cons` は **ヘッド** (最初の要素) と **テイル** (残りの要素) で与えられた空でない列をコンストラクトします。例えば、列 1,2,3 は次のように書けます:

```
ints_cons(1, ints_cons(2, ints_cons(3, ints_nil)))
```

練習問題 9 `nodes` の `count` パラメータと述語 `stack` を型 `ints` の `values` パラメータで置き換えて、その述語本体を更新してください。さらに関数 `create_stack`, `stack_push`, `stack_dispose` を更新してください。ここでは `stack_pop` を気にしないでください。

6.10 不動点関数

どうやって関数 `stack_pop` に対する注釈を更新すべきでしょうか？ 事後条件と close 文で現在の列のテイルを参照する必要があります。さらにその返り値を指定するために、現在の列のヘッドを参照する必要があります。これは **不動点関数** (*fixpoint functions*) を使うことで可能になります:

```
fixpoint int ints_head(ints values) {
    switch (values) {
        case ints_nil: return 0;
        case ints_cons(value, values0): return value;
    }
}

fixpoint ints ints_tail(ints values) {
    switch (values) {
        case ints_nil: return ints_nil;
        case ints_cons(value, values0): return values0;
    }
}
```

switch 文を帰納データ型に使えることに注意してください。それぞれのコンストラクタに対して、きっちり 1 回の case が必要です。パラメータを取るコンストラクタに対する case では、指定されたパラメータ名がその値がコンストラクトされたときに使われた関連するコンストラクタ引数値に束縛されます。不動点関数の本体は (**帰納パラメータ** (*inductive parameter*) と呼ばれる) 関数のパラメータの 1 つを持つ switch 文でなければなりません。さらに、それぞれの case 本体は return 文でなければなりません。

通常の C 言語関数と対照的に、不動点関数は注釈において使われる式で使うことができます。こ

れは関数 stack_pop を新しい述語定義に適合させて、その返り値を指定するのに、ints_head と ints_tail 関数を使うことができることを意味しています。

練習問題 10 上記を試してください。

これでスタック実装の関数的な正しさを明記でき、VeriFast は新しい main 関数を検証できます。

練習問題 11 与えられたスタックの要素の和を返す C 言語関数 stack_get_sum を追加してください。その関数を再帰的なヘルパー関数 nodes_get_sum を使って実装してください。再帰的な不動点関数 ints_sum を使ってその新しい C 言語関数を明記してください。**Verify** メニューを確認して、算術オーバーフローを無効にしてください。次の main 関数を検証してください:

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
    int sum = stack_get_sum(s);
    assert(sum == 30);
    int result1 = stack_pop(s);
    assert(result1 == 20);
    int result2 = stack_pop(s);
    assert(result2 == 10);
    stack_dispose(s);
    return 0;
}
```

VeriFast は再帰的な不動点関数をサポートしています。 VeriFast は直接の再帰のみを許可し、再帰呼び出しの帰納パラメータの値がその呼び出し元の帰納パラメータの値のコンストラクタ引数であることを要求することで、その不動点関数が常に停止することを強制します。

6.11 補題

注意: 章 6.25 から章 6.29 にかけて別に再帰的な述語を導入しています。まだ再帰的な述語に馴染みがないのであれば、この章をはじめる前にそれらの章を参照してください。

(練習問題 5 の答である) 最初に注釈したスタックの実装に戻りましょう。スタック中の要素の数を返す、ループを使って実装された stack_get_count 関数を追加しましょう:

```
int stack_get_count(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == count;
{
    //@ open stack(stack, count);
```

```
    struct node *head = stack->head;
    struct node *n = head;
    int i = 0;
    while (n != 0)
        //@ invariant true;
    {
        n = n->next;
        i++;
    }
    //@ close stack(stack, count);
    return i;
}
```

ループ不変条件 **true** は有意な働きをしないのは明確です。それはどうあるべきでしょうか？ `n` が nodes 列中のどこかにあることを表わす必要があります。これを行なう 1 つの方法は**連結リストセグメント** (*linked list segments*) (もしくは略して **リストセグメント** (*list segments*)) と連携することです。 `head` から `n` へのリストセグメントと `n` から `0` への別のリストセグメントを含むループ不変条件を表明しましょう:

```
//@ invariant lseg(head, n, i) &*& lseg(n, 0, count - i);
```

次のように述語 `lseg` が定義できます:

```
predicate lseg(struct node *first, struct node *last, int count) =
    first == last ?
        count == 0
    :
        0 < count &*& first != 0 &*&
        first->value |-> _ &*& first->next |-> ?next &*&
        malloc_block_node(first) &*& lseg(next, last, count - 1);
```

まだ終わりではありません。最初にループに入るときのループ不変条件を証明する必要があります。つまり、次の式を証明する必要があります。

```
lseg(head, head, 0) &*& lseg(head, 0, count)
```

1 つ目の項は簡単です: それは空で、単に閉じることができます。 2 つ目の項は `nodes(head, count)` チャンクを等価な `lseg(head, 0, count)` チャンクへの書き換えることを要求します。それは静的に制限された数の open と close 操作では行なえないことに注意してください。ループもしくは再帰関数のどちらかを使う必要があるのです。 VeriFast は注釈中でのループを許さないので、再帰関数を使うことになります。C 言語関数は実行時に意味を持つので、C 言語関数は使いません; さらに、不動点関数は open もしくは close 文を含むことができないので、不動点関数は使えません。 VeriFast は 3 種類目の関数 **補題関数** (*lemma functions*) をサポートしています。この関数は通常の C 言語関数に似ていますが、この関数はフィールド代入や通常の関数呼び出しを行なわず、必ず停止します。その関数の呼び出しは実行時の作用がありません。この関数の目的はなんらかの

ヒープチャンクを等価なヒープチャンクの集合、すなわち同じ実メモリの値を表現する異なるヒープチャンクに変換することです。不動点関数と対照的に、この関数は式の中からではなく、別の呼び出し文からのみ呼び出されます。

VeriFast は直接の再帰と各再帰呼び出しを以下のように検査することで補題関数の停止性を検査します: 最初に、もし再帰呼び出しの事前条件を消費した後シンボリックヒープにフィールドチャンクが残っていたら、その呼び出しは許可されます。これはヒープサイズについての帰納法です。そうでなければ、もし補題関数の本体が型が帰納データ型であるパラメータに対する switch 文であったなら、再帰呼び出しにおけるこのパラメータを表わす引数は同じパラメータを表わす呼び出し元の引数のコンストラクタ引数でなければなりません。これは帰納パラメータについての帰納法です。最後に、もし補題関数の本体がそのような switch 文でなかったら、1 つ以上の open 操作を通じて呼び出し元の事前条件によって消費される最初のヒープチャンクから呼び出し先の事前条件で消費される最初のヒープチャンクを獲得しなければなりません。これは事前条件の項の導出についての帰納法です。

次の補題関数を使って nodes チャンクを lseg チャンクに変換できます:

```
lemma void nodes_to_lseg_lemma(struct node *first)
    requires nodes(first, ?count);
    ensures lseg(first, 0, count);
{
    open nodes(first, count);
    if (first != 0) {
        nodes_to_lseg_lemma(first->next);
    }
    close lseg(first, 0, count);
}
```

この補題はヒープサイズについての帰納法を実行するため認められることに注意してください。

練習問題 12 同様に逆の操作が必要になります。 0 で終わる lseg チャンクを取って、それを nodes チャンクに変換する補題関数 lseg_to_nodes_lemma を書いてください。

これで stack_get_count 関数は次のようになります:

```
int stack_get_count(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == count;
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ nodes_to_lseg_lemma(head);
    struct node *n = head;
    int i = 0;
    //@ close lseg(head, head, 0);
    while (n != 0)
        //@ invariant lseg(head, n, i) &*& lseg(n, 0, count - i);
```

```
    {
        //@ open lseg(n, 0, count - i);
        n = n->next;
        i++;
        // Error!
    }
    //@ open lseg(0, 0, _);
    //@ lseg_to_nodes_lemma(head);
    //@ close stack(stack, count);
    return i;
}
```

ほとんど完了しました。残りはループ本体の終わりで起きるエラーを修正するだけです。そのポイントは、次のようなチャンクを持っていることです:

```
lseg(head, ?old_n, i - 1) &*& old_n->value |-> _ &*& old_n->next |-> n &*&
malloc_block_node(old_n) &*& lseg(n, 0, count - i)
```

これらのチャンクを次のように変換する必要があります:

```
lseg(head, n, i) &*& lseg(n, 0, count - i)
```

つまり、head ではじまるリストセグメントの最後に、n の古い値の node を追加する必要があるのです。このためには再び補題関数を使う必要があるでしょう。次が最初の試みです:

```
lemma void lseg_add_lemma(struct node *first)
    requires
        lseg(first, ?last, ?count) &*& last != 0 &*& last->value |-> _ &*&
        last->next |-> ?next &*& malloc_block_node(last);
    ensures lseg(first, next, count + 1);
{
    open lseg(first, last, count);
    if (first == last) {
        close lseg(next, next, 0);
    } else {
        lseg_add_lemma(first->next);
    }
    close lseg(first, next, count + 1);
}
```

VeriFast は、first と last が等しいパスで、最後の close 操作をしようとしてエラーを吐きます。first と next が等しいことを仮定してしまい、count + 1 とゼロが等しいことを証明できません。first は常にスタックの 1 つの node を指し、next は別の node を指すか 0 に等しいので、このシナリオでは first と next は異なります。けれども、今回の補題関数の事前条件はこれを表わしていません。この情報を含めるために、同様に next から 0 へのリストセグメントを要求する必要があります。最後の close 操作の前にこのリストセグメントを開いて閉じることで、first と next が異

なるという情報を得ることができます。具体的には、VeriFast があるフィールドチャンクを生成し、同じフィールドを表わす別のフィールドチャンクが既にシンボリックヒープ中に存在しても、そのフィールドチャンクが異なる構造体インスタンスに属することを示す仮定を追加します。

次のような lseg_add_lemma と stack_get_count 関数が得られます:

```
/*@
lemma void lseg_add_lemma(struct node *first)
    requires
        lseg(first, ?last, ?count) &*& last != 0 &*& last->value |-> _ &*&
        last->next |-> ?next &*& malloc_block_node(last) &*&
        lseg(next, 0, ?count0);
    ensures lseg(first, next, count + 1) &*& lseg(next, 0, count0);
{
    open lseg(first, last, count);
    if (first == last) {
        close lseg(next, next, 0);
    } else {
        lseg_add_lemma(first->next);
    }
    open lseg(next, 0, count0);
    close lseg(next, 0, count0);
    close lseg(first, next, count + 1);
}
@*/

int stack_get_count(struct stack *stack)
    //@ requires stack(stack, ?count);
    //@ ensures stack(stack, count) &*& result == count;
{
    //@ open stack(stack, count);
    struct node *head = stack->head;
    //@ nodes_to_lseg_lemma(head);
    struct node *n = head;
    int i = 0;
    //@ close lseg(head, head, 0);
    while (n != 0)
        //@ invariant lseg(head, n, i) &*& lseg(n, 0, count - i);
    {
        //@ open lseg(n, 0, count - i);
        n = n->next;
        i++;
        //@ lseg_add_lemma(head);
    }
    //@ open lseg(0, 0, _);
```

```
    //@ lseg_to_nodes_lemma(head);
    //@ close stack(stack, count);
    return i;
}
```

これで検証できました。

練習問題 13 次の関数を検証してください。追加の補題が必要になるでしょう。

```
void stack_push_all(struct stack *stack, struct stack *other)
    //@ requires stack(stack, ?count) &*& stack(other, ?count0);
    //@ ensures stack(stack, count0 + count);
{
    struct node *head0 = other->head;
    free(other);
    struct node *n = head0;
    if (n != 0) {
        while (n->next != 0)
        {
            n = n->next;
        }
        n->next = stack->head;
        stack->head = head0;
    }
}
```

練習問題 14 インプレースで、すなわちメモリ確保せずにスタックを逆転する関数 stack_reverse を実装/明記/検証してください。(6.9 章を見て) 関数的な正しさを検証してください。追加の不動点と補題を定義することになるでしょう。

6.12 関数ポインタ

スタックを取って与えられた述語を満たさない要素を削除する関数を書いてみましょう:

```
typedef bool int_predicate(int x);

struct node *nodes_filter(struct node *n, int_predicate *p)
    //@ requires nodes(n, _);
    //@ ensures nodes(result, _);
{
    if (n == 0) {
        return 0;
    } else {
        //@ open nodes(n, _);
```

```
        bool keep = p(n->value);
        if (keep) {
            struct node *next = nodes_filter(n->next, p);
            //@ open nodes(next, ?count);
            //@ close nodes(next, count);
            n->next = next;
            //@ close nodes(n, count + 1);
            return n;
        } else {
            struct node *next = n->next;
            free(n);
            struct node *result = nodes_filter(next, p);
            return result;
        }
    }
}

void stack_filter(struct stack *stack, int_predicate *p)
    //@ requires stack(stack, _);
    //@ ensures stack(stack, _);
{
    //@ open stack(stack, _);
    struct node *head = nodes_filter(stack->head, p);
    //@ assert nodes(head, ?count);
    stack->head = head;
    //@ open nodes(head, count);
    //@ close nodes(head, count);
    //@ close stack(stack, count);
}

bool neq_20(int x)
    //@ requires true;
    //@ ensures true;
{
    return x != 20;
}

int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
```

```
    stack_push(s, 30);
    stack_filter(s, neq_20);
    stack_dispose(s);
    return 0;
}
```

このプログラムは検証できません。関数 `nodes_filter` 中での `p` の呼び出し検証するために使う契約を、VeriFast は知らないのです。 VeriFast はそれぞれの関数が契約を持つことを要求します:

```
typedef bool int_predicate(int x);
    //@ requires true;
    //@ ensures true;
```

けれども、これは十分ではありません。 `p` は型 `int_predicate *` ですが、これはそのポインタが `int_predicate` 関数の型シグニチャと契約を共なう関数を指すことを保証していません。実際、どのような整数もポインタにキャストでき、どのようなポインタも関数ポインタにキャストできます。そのため、VeriFast は、プログラムで宣言したそれぞれの関数の型 `T` に対して純粋なブール関数 `is_T` を導入します。型 `T *` の関数ポインタ `p` の呼び出しを検証するとき、VeriFast は `is_T(p)` が真であることをチェックします。与えられた関数 `f` に対して事実 `is_T(f)` を生成するために、`f` のヘッダは関数の型の実装節を含まなければなりません:

```
bool neq_20(int x) //@ : int_predicate
    //@ requires true;
    //@ ensures true;
```

さらにこれは VeriFast に、`neq_20` の契約が `int_predicate` の契約を包含することをチェックさせます。

最後に、事実 `is_int_predicate` をクライアントから呼び出し先に渡します:

```
struct node *nodes_filter(struct node *n, int_predicate *p)
    //@ requires nodes(n, _) &*& is_int_predicate(p) == true;
    //@ ensures nodes(result, _);

void stack_filter(struct stack *stack, int_predicate *p)
    //@ requires stack(stack, _) &*& is_int_predicate(p) == true;
    //@ ensures stack(stack, _);
```

`is_int_predicate(p)` の代わりに `is_int_predicate(p) == true` と書くことに注意してください。VeriFast は後者の形式を述語表明としてパースし、かつそのような述語は存在しないため、これを拒絶します。

練習問題 15 全てを統合してください。

`stack_filter` 関数の注記は 2 つの意味で十分ではありません: 1 つ目は、その事前条件はどのよ

うなヒープチャンクも要求しないので、int_predicate 関数はどのようなメモリ位置も読み出せないことです; 例えば、それによって指定した値より大きい全ての要素をフィルタできないのです。これは 6.15 章での述語族を使うことで解決できます。2 つ目は、少しの要素が削除されただけでも、この実装はそれぞれの next ポインタを再代入してしまうことです。これは 6.13 章での参照によるパラメータを使うことで解決できます。

6.13 参照によるパラメータ

ここでは 6.12 章での stack_filter 関数の別実装を見てみます。それぞれの next ポインタを再代入する代わりに、ポインタを next ポインタに渡して、変化したポインタのみ再代入します。

```
void nodes_filter(struct node **n, int_predicate *p) {
    if (*n != 0) {
        bool keep = p((*n)->value);
        if (keep) {
            nodes_filter(&(*n)->next, p);
        } else {
            struct node *next = (*n)->next;
            free(*n);
            *n = next;
            nodes_filter(n, p);
        }
    }
}

void stack_filter(struct stack *stack, int_predicate *p) {
    nodes_filter(&stack->head, p);
}
```

このプログラムでは、参照を用いて現在の node へのポインタを関数 nodes_filter に渡しています。関数 nodes_filter 中では、n をデリファレンスして現在の node へのポインタを得ます。VeriFast はポインタのデリファレンスをフィールドのデリファレンスと同じように扱います。けれども、シンボリックヒープにフィールドチャンクが見つかる代わりに、それは一般的な変数チャンクに見つかります; この場合、デリファレンスされるポインタはポインタを保持する変数を指しているので、それは pointer チャンクであると期待されます。述語 pointer は prelude.h で次のように定義されています:

```
predicate pointer(void **pp; void *p);
```

(後にセミコロンの意味について議論します; ここでは、単にコンマのように読んでみてください。) フィールドチャンクの場合と同様に、1 番目の引数は変数のアドレスで、2 番目の引数はその変数の現在の値です。

次は関数 nodes_filter に対する正当な契約です:

```
void nodes_filter(struct node **n, int_predicate *p)
    /*@ requires
            pointer(n, ?node) &*& nodes(node, _) &*&
            is_int_predicate(p) == true;
      @*/
    //@ ensures pointer(n, ?node0) &*& nodes(node0, _);
```

stack_filter から nodes_filter を呼び出しできるようにするために、ポインタチャンクを生成する必要があります。具体的には、stack_head チャンクをポインタチャンクに変換する必要があります。これは単に stack_head チャンクを開くだけです。そのポインタチャンクを stack_head チャンクに戻すには、再びその stack_head チャンクを閉じるだけです:

```
void stack_filter(struct stack *stack, int_predicate *p)
    //@ requires stack(stack, _) &*& is_int_predicate(p) == true;
    //@ ensures stack(stack, _);
{
    //@ open stack(stack, _);
    //@ open stack_head(stack, _);
    nodes_filter(&stack->head, p);
    //@ assert pointer(&stack->head, ?head) &*& nodes(head, ?count);
    //@ close stack_head(stack, head);
    //@ open nodes(head, count);
    //@ close nodes(head, count);
    //@ close stack(stack, count);
}
```

close 文ではパターンを使えないので、stack_head チャンクを閉じる前に head フィールドの値を変数 head で束縛する必要があることに注意してください。

練習問題 16 関数 nodes_filter を検証してください。

注意: 現在 VeriFast は、ポインタへのポインタ、整数へのポインタ、文字へのポインタのためにデリファレンス演算子 (*) をサポートしています。後者の場合、prelude.h で定義された次の述語が使えます:

```
predicate integer(int *p; int v);
predicate character(char *p; char v);
```

まだ VeriFast はローカル変数のアドレスの受け取りをサポートしていません。

6.14 算術オーバーフロー

次のようなプログラムを考えます:

```
int main()
    //@ requires true;
    //@ ensures true;
{
    int x = 2000000000 + 2000000000;
    assert(0 <= x);
    return 0;
}
```

このプログラムの挙動は C 言語で一意に定義されません。具体的には、**int** 型は有限範囲の整数のみをサポートしていて、さらに C 言語はこの範囲の限界を指定していないのです。この限界は **実装依存** です。 C 言語は型 **int** の最小値と最大値をマクロ INT_MIN と INT_MAX によって指定しています。それぞれ INT_MIN は -32767 以下、INT_MAX は 32767 以上です。さらに、もし型 **int** の算術演算の結果、型 **int** の限界の範囲外の値を得たら何が起きるかを C 言語は指定していません。

VeriFast は任意の C 言語実装に準拠するか (もしくは任意の規格に準拠するか) を気にしません。具体的には INT_MIN が -2^31 に等しく、INT_MAX が 2^31 - 1 に等しいことを仮定します。

(注釈中ではなく) C 言語コード中において算術演算をシンボリックに評価するとき、**Verify** メニューの Checking arithmetic overflow がオフになっていなければ、VeriFast は演算の結果が結果の型の範囲に収まるかどうかチェックします。結果として、VeriFast は上記のプログラムを拒否します。また次のプログラムも拒否します:

```
void int_add(int *x, int *y)
    //@ requires integer(x, ?vx) &*& integer(y, ?vy);
    //@ ensures integer(x, vx+vy) &*& integer(y, vy);
{
    int x_deref = *x;
    int y_deref = *y;
    *x = x_deref + y_deref;
}
```

32-bit マシン上で正しく動作するように、この関数を修正してみましょう。次は最初の試みです:

```
#include "stdlib.h"
void int_add(int *x_ptr, int *y_ptr)
    //@ requires integer(x_ptr, ?x_value) &*& integer(y_ptr, ?y_value);
    //@ ensures integer(x_ptr, x_value+y_value) &*& integer(y_ptr, y_value);
{
    int x = *x_ptr;
    int y = *y_ptr;
    if (0 <= x) {
        if (INT_MAX - x < y) abort();
    } else {
```

```
        if (y < INT_MIN - x) abort();
    }
    *x_ptr = x + y;
}
```

この関数は 32-bit マシンで正しく動作するにもかかわらず、VeriFast はこの関数を受理しません。これは算術操作の結果がその型の範囲内かどうかチェックするけれども、元の値がその型の範囲内かどうか仮定しないからです。検証のパフォーマンスのために、これらの仮定は自動的には生成されません。これらの仮定を生成するためには、produce_limits ゴーストコマンドをこのコードに挿入する必要があります。produce_limits コマンドの引数は (注釈中で宣言されたローカル変数でなく) C 言語ローカル変数の名前でなければなりません。次のプログラムは検証できます。

```
#include "stdlib.h"
void int_add(int *x_ptr, int *y_ptr)
    //@ requires integer(x_ptr, ?x_value) &*& integer(y_ptr, ?y_value);
    //@ ensures integer(x_ptr, x_value+y_value) &*& integer(y_ptr, y_value);
{
    int x = *x_ptr;
    int y = *y_ptr;
    //@ produce_limits(x);
    //@ produce_limits(y);
    if (0 <= x) {
        if (INT_MAX - x < y) abort();
    } else {
        if (y < INT_MIN - x) abort();
    }
    *x_ptr = x + y;
}
```

注意: 上記のオーバーフローチェックされた整数に追加した実装は、パフォーマンスの点で最適なものではありません。例えば、x86 命令セットは、オーバーフローフラグがセットされるとソフトウェア割り込みを発生させる INTO (Interrupt on Overflow) 命令を含んでいます。この命令を使った実装はより良いパフォーマンスになるでしょう。

注意: integer、character もしくは pointer チャンクがヒープにある場合、それぞれ補題 integer_limits、character_limits と pointer_limits を使うことができます。この補題の契約はファイル prelude.h にあります。

6.15 述語族

6.12 章の stack_filter 関数に戻りましょう。スタックから、与えられた値を全て削除したくなったとします。

```
typedef bool int_predicate(void *data, int x);
```

```
struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
{
    if (n == 0) {
        return 0;
    } else {
        bool keep = p(data, n->value);
        if (keep) {
            struct node *next = nodes_filter(n->next, p, data);
            n->next = next;
            return n;
        } else {
            struct node *next = n->next;
            free(n);
            struct node *result = nodes_filter(next, p, data);
            return result;
        }
    }
}

void stack_filter(struct stack *stack, int_predicate *p, void *data)
{
    struct node *head = nodes_filter(stack->head, p, data);
    stack->head = head;
}

struct neq_a_data {
    int a;
};

bool neq_a(struct neq_a_data *data, int x)
{
    bool result = x != data->a;
    return result;
}

int read_int();

int main()
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
    stack_push(s, 30);
    int a = read_int();
```

```
    struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
    if (data == 0) abort();
    data->a = a;
    stack_filter(s, neq_a, data);
    free(data);
    stack_dispose(s);
    return 0;
}
```

どうやってスタックモジュールを指定すればいいでしょうか？ 次はその試みです:

```
//@ predicate int_predicate_data(void *data) = ??

typedef bool int_predicate(void *data, int x);
    //@ requires int_predicate_data(data);
    //@ ensures int_predicate_data(data);

struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
    /*@ requires
            nodes(n, _) &*& is_int_predicate(p) == true &*&
            int_predicate_data(data);
      @*/
    //@ ensures nodes(result, _) &*& int_predicate_data(data);
{ ... }

void stack_filter(struct stack *stack, int_predicate *p, void *data)
    /*@ requires
            stack(stack, _) &*& is_int_predicate(p) == true &*&
            int_predicate_data(data);
      @*/
    //@ ensures stack(stack, _) &*& int_predicate_data(data);
{ ... }
```

問題は述語 int_predicate_data の定義です。 data ポインタが指すデータ構造を予測するようなスタックモジュールを表わす方法がありません。述語の定義を選択することができれば、たやすく検証できます:

```
//@ predicate int_predicate_data(void *data) = neq_a_data_a(data, _);

bool neq_a(struct neq_a_data *data, int x) //@ : int_predicate
    //@ requires int_predicate_data(data);
    //@ ensures int_predicate_data(data);
{
```

```
    //@ open int_predicate_data(data);
    bool result = x != data->a;
    //@ close int_predicate_data(data);
    return result;
}

int read_int();
    //@ requires true;
    //@ ensures true;

int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
    stack_push(s, 30);
    int a = read_int();
    struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
    if (data == 0) abort();
    data->a = a;
    //@ close int_predicate_data(data);
    stack_filter(s, neq_a, data);
    //@ open int_predicate_data(data);
    free(data);
    stack_dispose(s);
    return 0;
}
```

けれども、これが実現可能でないことは明確です。結局、スタックモジュールは 1 つ以上のクライアントがあるので、同じ述語を表わす複数の定義を持つことになります。具体的には、型 int_predicate のそれぞれの関数について int_predicate_data の 1 つの定義を持つことになります。もし型 int_predicate で与えられた関数に関連した int_predicate_data の定義を具体的に指すことができれば、この問題は解決できます。これはまさに **述語族** (*predicate families*) で可能になります。述語族は通常の述語に似ていますが、それぞれの定義が異なる**インデックス** (*index*) と関連付けられた複数の定義を持つことができる点で異なります。述語族のインデックスは関数ポインタでなければなりません。

述語族を適用すると、スタックモジュールを表わす次のような定義が得られます:

```
//@ predicate_family int_predicate_data(void *p)(void *data);

typedef bool int_predicate(void *data, int x);
```

```
    //@ requires int_predicate_data(this)(data);
    //@ ensures int_predicate_data(this)(data);

struct node *nodes_filter(struct node *n, int_predicate *p, void *data)
    /*@ requires
            nodes(n, _) &*& is_int_predicate(p) == true &*&
            int_predicate_data(p)(data);
      @*/
    //@ ensures nodes(result, _) &*& int_predicate_data(p)(data);

void stack_filter(struct stack *stack, int_predicate *p, void *data)
    /*@ requires
            stack(stack, _) &*& is_int_predicate(p) == true &*&
            int_predicate_data(p)(data);
      @*/
    //@ ensures stack(stack, _) &*& int_predicate_data(p)(data);
```

関数の型の契約中で、関数ポインタを this で参照できることに注意してください。
クライアントは次のように検証できます:

```
struct neq_a_data {
    int a;
};

/*@
predicate_family_instance int_predicate_data(neq_a)(void *data) =
    neq_a_data_a(data, _);
@*/

bool neq_a(struct neq_a_data *data, int x) //@ : int_predicate
    //@ requires int_predicate_data(neq_a)(data);
    //@ ensures int_predicate_data(neq_a)(data);
{
    //@ open int_predicate_data(neq_a)(data);
    bool result = x != data->a;
    //@ close int_predicate_data(neq_a)(data);
    return result;
}

int read_int();
    //@ requires true;
    //@ ensures true;
```

```
int main()
    //@ requires true;
    //@ ensures true;
{
    struct stack *s = create_stack();
    stack_push(s, 10);
    stack_push(s, 20);
    stack_push(s, 30);
    int a = read_int();
    struct neq_a_data *data = malloc(sizeof(struct neq_a_data));
    if (data == 0) abort();
    data->a = a;
    //@ close int_predicate_data(neq_a)(data);
    stack_filter(s, neq_a, data);
    //@ open int_predicate_data(neq_a)(data);
    free(data);
    stack_dispose(s);
    return 0;
}
```

練習問題 17 int を取り int を返す関数 f を取る関数 stack_map を追加してください。 stack_map はスタックのそれぞれの要素の値を、その値に f を適用した結果で置き換えます。値 10, 20, 30 を含むスタックを作るクライアントプログラムを書いてください; それからユーザから値を読み; そして stack_map を使ってそのスタックのそれぞれの要素にその値を加えてください。完成したプログラムのメモリ安全性を検証してください。

6.16 ジェネリクス

整数のスタックに対する stack_reverse 関数の関数的な正しさを検証した練習問題 14 の答を考えてみましょう。ints 帰納データ型、不動点関数 append と reverse、そして補題 append_nil と append_assoc を使いました。ここで、ポインタのスタックに対して同じ機能が必要になったと想定してみましょう。明確に、C 言語はジェネリクスをサポートしていないので、C 言語コードをコピーアンドペーストして、至るところで **int** を __void *__ に置き換える必要があります。けれども幸運なことに、VeriFast は帰納データ型、不動点関数、補題、そして述語に対するジェネリクスをサポートしてます。そのためそれらをコピーアンドペーストする代わりに、要素の型を使ってそれらをパラメータ化できます。次はパラメータ化した ints、append、そして append_nil です:

```
inductive list<t> = nil | cons(t, list<t>);

fixpoint list<t> append<t>(list<t> xs, list<t> ys) {
    switch (xs) {
        case nil: return ys;
        case cons(x, xs0): return cons<t>(x, append<t>(xs0, ys));
    }
```

```
}

lemma void append_nil<t>(list<t> xs)
    requires true;
    ensures append<t>(xs, nil<t>) == xs;
{
    switch (xs) {
        case nil:
        case cons(x, xs0):
            append_nil<t>(xs0);
    }
}
```

見たように、帰納データ型の定義、不動点関数の定義、もしくは補題の定義は、山括弧で囲まれた **型パラメータ** (*type parameters*) のリストである **型パラメータリスト** (*type parameter list*) を任意に受け取ります。その定義の中では、この型パラメータは他の型のように使うことができます。型パラメータ化されたデータ型、不動点、補題、述語、もしくは型パラメータ化されたデータ型のコンストラクタを使うときは、山括弧で囲まれた型のリストで**型引数リスト** (*type argument list*) が指定されなければなりません。

次はポインタのスタックに対する述語 `nodes` と `stack` と関数 `stack_reverse` です:

```
predicate nodes(struct node *node, list<void *> values) =
    node == 0 ?
        values == nil<void *>
    :
        node->next |-> ?next &*& node->value |-> ?value &*&
        malloc_block_node(node) &*& nodes(next, ?values0) &*&
        values == cons<void *>(value, values0);

predicate stack(struct stack *stack, list<void *> values) =
    stack->head |-> ?head &*& malloc_block_stack(stack) &*& nodes(head, values);
```

```
void stack_reverse(struct stack *stack)
    //@ requires stack(stack, ?values);
    //@ ensures stack(stack, reverse<void *>(values));
{
    //@ open stack(stack, values);
    struct node *n = stack->head;
    struct node *m = 0;
    //@ close nodes(m, nil<void *>);
    //@ append_nil<void *>(reverse<void *>(values));
    while (n != 0)
        /*@
        invariant
```

```
            nodes(m, ?values1) &*& nodes(n, ?values2) &*&
            reverse<void *>(values) ==
                append<void *>(reverse<void *>(values2), values1);
        @*/
    {
        //@ open nodes(n, values2);
        struct node *next = n->next;
        //@ assert nodes(next, ?values2tail) &*& n->value |-> ?value;
        n->next = m;
        m = n;
        n = next;
        //@ close nodes(m, cons<void *>(value, values1));
        /*@ append_assoc<void *>(reverse<void *>(values2tail),
                cons<void *>(value, nil<void *>), values1);
          @*/
    }
    //@ open nodes(n, _);
    stack->head = m;
    //@ close stack(stack, reverse<void *>(values));
}
```

ジェネリクスのおかげで、int のスタックとポインタのスタックの両方を表わす同じデータ型、不動点、そして補題を再利用できます。けれども、いたるところに型引数リストを挿入する必要があることを考えると、このアプローチは多くの構文上のオーバーヘッドを導入するように思えます。幸運にも VeriFast は **型引数推論** (*type argument inference*) を行ないます。もし VeriFast が式中にある型パラメータ化された要素を発見し、型引数リストが指定されていなかった場合、VeriFast は型引数リストを推論します。 VeriFast の型システムはサブタイピングを持たないので、単純なユニフィケーションに基づく推論アプローチで十分です。その結果、式中で明確に型引数リストを与える必要はほとんどありません。特に、この例では式中の全ての型引数リストは省略できます。注意: VeriFast は型中での型引数リストを推論しません; つまり型パラメータ化された帰納データ型を使うときには、常に型引数を明示的に与える必要があります。

もちろん `list` データ型は単なる整数のスタックやポインタのスタックよりも一般的に有用です。実際、自明でないプログラムの検証ではリストの使用が要求されます。このため、VeriFast はリストデータ型、この例で使った不動点と補題、さらに他の機能を含むヘッダファイル `list.h` を備えています。このヘッダファイルは VeriFast によって検証されるそれぞれのファイルから暗黙的にインクルードされるので、それを明示的にインクルードする必要がありません。注意: これはまた、名前の衝突を引き起こすため、独自の `nil` や `cons` もしくは `list.h` で提供されるその他の要素を定義することができないことを意味しています。

6.17 述語値

前章では、ポインタのスタックを手に入れました。この章では、このスタックを使ってオブジェクトの集合の記録をつけてみます。

次のプログラムを検証してみましょう。これは 2D ベクトルに対するスタックに基いた電卓です。これは前章でのスタックを使っています。ユーザはベクトルをスタックにプッシュし、上位 2 つの

ベクトルをそれらの和で置換し、印字するために上位のベクタをポップします。

```
struct vector {
    int x;
    int y;
};

struct vector *create_vector(int x, int y)
{
    struct vector *result = malloc(sizeof(struct vector));
    if (result == 0) abort();
    result->x = x;
    result->y = y;
    return result;
}

int main()
{
    struct stack *s = create_stack();
    while (true)
    {
        char c = input_char();
        if (c == ' p' ) {
            int x = input_int();
            int y = input_int();
            struct vector *v = create_vector(x, y);
            stack_push(s, v);
        } else if (c == ' +' ) {
            bool empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v1 = stack_pop(s);
            empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v2 = stack_pop(s);
            struct vector *sum = create_vector(v1->x + v2->x, v1->y + v2->y);
            free(v1);
            free(v2);
            stack_push(s, sum);
        } else if (c == ' =' ) {
            bool empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v = stack_pop(s);
            output_int(v->x);
            output_int(v->y);
```

```
            free(v);
        } else {
            abort();
        }
    }
}
```

create_vector の仕様は簡単です:

```
/*@ predicate vector(struct vector *v) =
        v->x |-> _ &*& v->y |-> _ &*& malloc_block_vector(v);
@*/

struct vector *create_vector(int x, int y)
    //@ requires true;
    //@ ensures vector(result);
```

トリッキーな部分は main のループに対するループ不変条件です。そのループ不変条件は s にスタックを持つことを表明すべきで、さらに s のそれぞれの要素にベクトルを持つことを表明しなければなりません。表明 stack(s, ?values) を使ってその 1 番目を表わすことができ、その 2 番目を表わすために再帰的な述語を容易に書くことができます:

```
predicate vectors(list<struct vector *> vs) =
    switch (vs) {
        case nil: return true;
        case cons(v, vs0): return vector(v) &*& vectors(vs0);
    };
```

これはうまく動作します。けれども、不幸にも、与えられた述語でリストのそれぞれの要素を表わしたいときは新しい述語を定義しなければなりません。これを解決するために VeriFast は **述語値** (*predicate values*) をサポートしています。つまり、述語を引数として別の述語に渡せるのです。これで上記の述語 vectors を次のように一般化できます:

```
predicate foreach(list<void *> vs, predicate(void *) p) =
    switch (vs) {
        case nil: return true;
        case cons(v, vs0): return p(v) &*& foreach(vs0, p);
    };
```

ジェネリクスを使うことで、この述語をさらに一般化して、任意の値のリストを取れるようにさえできます:

```
predicate foreach<t>(list<t> vs, predicate(t) p) =
    switch (vs) {
        case nil: return true;
```

```
        case cons(v, vs0): return p(v) &*& foreach(vs0, p);
    };
```

型引数の推論のおかげで再帰 `foreach` 呼び出しの型引数を省略できることに注意してください。

この述語は一般に使用できるので、それは `list.h` からインクルードしています; 結果として、それはそれぞれのファイルで自動的に有効で、自分で定義する必要はないのです。

練習問題 18 `foreach` を使ってこのプログラムを検証してください。

6.18 述語コンストラクタ

前章のプログラムにひねりを加えてみましょう:

```
struct vector *create_vector(int limit, int x, int y)
{
    if (x * x + y * y > limit * limit) abort();
    struct vector *result = malloc(sizeof(struct vector));
    if (result == 0) abort();
    result->x = x;
    result->y = y;
    return result;
}

int main()
{
    int limit = input_int();
    struct stack *s = create_stack();
    while (true)
    {
        char c = input_char();
        if (c == 'p') {
            int x = input_int();
            int y = input_int();
            struct vector *v = create_vector(limit, x, y);
            stack_push(s, v);
        } else if (c == '+') {
            bool empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v1 = stack_pop(s);
            empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v2 = stack_pop(s);
            struct vector *sum = create_vector(limit, v1->x + v2->x,
                                               v1->y + v2->y);
            free(v1);
```

```
            free(v2);
            stack_push(s, sum);
        } else if (c == '=') {
            bool empty = stack_is_empty(s);
            if (empty) abort();
            struct vector *v = stack_pop(s);
            int x = v->x;
            int y = v->y;
            free(v);
            assert(x * x + y * y <= limit * limit);
            output_int(x);
            output_int(y);
        } else {
            abort();
        }
    }
}
```

これでこのプログラムはベクトルのサイズの境界 (リミット) をユーザにたずねてから開始するようになりました。ベクトルを生成するとき、このプログラムはそのサイズがリミットを超えていないことをチェックします; そうでなければプログラムは中断します。ベクトルを印字するとき、このプログラムはそのベクトルがサイズリミットを満たすことを表明します。

どうすればこの assert 文を検証できるでしょうか？

ベクトルのサイズがリミットの範囲内であるという情報を含むように、述語 `vector` を拡張する必要があります。けれども、このリミットはローカル変数で、述語定義中ではスコープの範囲外です。そこで追加の引数を渡す必要があります。しかし、述語 `foreach` はもはや使えません。それは唯一 1 つのパラメータを取る述語であると期待されているからです。これを解決するために、VeriFast は **述語コンストラクタ** (*predicate constructors*) の形で、**部分適用された述語** (*partially applied predicates*) をサポートしています。このプログラム例を検証するために、次のような述語コンストラクタ `vector` を定義できます:

```
/*@
predicate_ctor vector(int limit)(struct vector *v) =
    v->x |-> ?x &*& v->y |-> ?y &*& malloc_block_vector(v) &*&
    x * x + y * y <= limit * limit;
@*/
```

リスト `values` のそれぞれの要素について、次のようなリミット `limit` を満たすベクトルを持つことを表わすことができます。

```
foreach(values, vector(limit))
```

つまり、述語の名前を使うときはいつでも、引数リストに適用される述語コンストラクタも使うことができます。それは、open 文中、close 文中、そして表明の中で使えます。

注意: 現時点では VeriFast は述語コンストラクタの引数位置におけるパターンをサポートしていません。

練習問題 19 このプログラムを検証してください。

6.19 マルチスレッド

整数の二分木を走査し、それぞれのノードの値の階乗を計算し、その結果を合計し、その合計をコンソールに印字するような次のようなプログラムを考えてみましょう。

```
int rand();

int fac(int x)
{
    int result = 1;
    while (x > 1)
    {
        result = result * x;
        x = x - 1;
    }
    return result;
}

struct tree {
    struct tree *left;
    struct tree *right;
    int value;
};

struct tree *make_tree(int depth)
{
    if (depth == 0) {
        return 0;
    } else {
        struct tree *left = make_tree(depth - 1);
        struct tree *right = make_tree(depth - 1);
        int value = rand();
        struct tree *t = malloc(sizeof(struct tree));
        if (t == 0) abort();
        t->left = left;
        t->right = right;
        t->value = value % 2000;
        return t;
    }
}
```

```
int tree_compute_sum_facs(struct tree *tree)
{
    if (tree == 0) {
        return 1;
    } else {
        int leftSum = tree_compute_sum_facs(tree->left);
        int rightSum = tree_compute_sum_facs(tree->right);
        int f = fac(tree->value);
        return leftSum + rightSum + f;
    }
}

int main()
{
    struct tree *tree = make_tree(22);
    int sum = tree_compute_sum_facs(tree);
    printf("%i", sum);
    return 0;
}
```

練習問題 20 このプログラムのメモリ安全性を検証してください。合計を計算した後 (6.4 章で見たように) 木がリークする可能性があります。

このプログラムの実行には筆者のマシンで 14 秒かかります。けれども、筆者のマシンはデュアルコアなので、両コアで同時にオペレーティングシステムでスケジュールされうる 2 つのスレッドに分散させれば、スピードアップできるかもしれません。

不幸にも、C 言語ではマルチスレッドのプログラムを書くための標準的な方法が定義されていません。(最新の C 言語標準 C11 はマルチスレッドのサポートを導入していますが、まだ広範囲にはサポートされていません。) Windows 向けのプログラムは Windows API を使うことができ、Unix 系オペレーティングシステム向けプログラムは一般に POSIX スレッド API を一般に使うことができます。けれども、VeriFast 配付版はサポートする全てのプラットフォーム横断する同一のインターフェイスを提供するこれらの API へのラッパーを含んでいます。これは `threading.h` で定義され、`threading.c` で実装されています; この両ファイルは VeriFast の `bin` ディレクトリにあります。 VeriFast は `bin` ディレクトリ中のヘッダファイルを自動的に見つけるので、単純にファイルの先頭に次の行を追加するだけで、これらの API を使えます:

```
#include "threading.h"
```

この章では、次のように `threading.h` で定義された関数 `thread_start_joinable` と `thread_join` を使います:

```
typedef void thread_run_joinable(void *data);

struct thread;

struct thread *thread_start_joinable(void *run, void *data);

void thread_join(struct thread *thread);
```

関数 thread_start_joinable は run 関数へのポインタを取り、新しいスレッドでこの関数を実行します。run 関数が要求するどのようなデータも thread_start_joinable の data パラメータを通して渡されます。このパラメータは単純に run 関数へ渡されます。 thread_start_joinable は、関数 thread_join を使って生成済みスレッドの合流のために使うことのできるスレッドハンドルを返します。関数 thread_join は指定したスレッドが完了するのを待ちます。

これらの関数を使って、次のようにこのプログラム例をスピードアップできます:

```
struct sum_data {
    struct thread *thread;
    struct tree *tree;
    int sum;
};

void summator(struct sum_data *data)
{
    int sum = tree_compute_sum_facs(data->tree);
    data->sum = sum;
}

struct sum_data *start_sum_thread(struct tree *tree)
{
    struct sum_data *data = malloc(sizeof(struct sum_data));
    struct thread *t = 0;
    if (data == 0) abort();
    data->tree = tree;
    t = thread_start_joinable(summator, data);
    data->thread = t;
    return data;
}

int join_sum_thread(struct sum_data *data)
{
    thread_join(data->thread);
    return data->sum;
```

```
}

int main()
{
    struct tree *tree = make_tree(22);
    struct sum_data *leftData = start_sum_thread(tree->left);
    struct sum_data *rightData = start_sum_thread(tree->right);
    int sumLeft = join_sum_thread(leftData);
    int sumRight = join_sum_thread(rightData);
    int f = fac(tree->value);
    printf("%i", sumLeft + sumRight + f);
    return 0;
}
```

main プログラムは深さ 22 の木を生成します。そして 2 つのスレッドが開始されます。1 つ目のスレッドは左のサブ木の値の階乗の合計を計算し、2 つ目のスレッドは右のサブ木の値の階乗の合計を計算します。それからメインスレッドは両スレッドの完了を待ち合わせ、最後に両スレッドの結果とルートノードの値の階乗を合計します。筆者のマシンでは、このプログラムの実行には 7 秒かかります; 2 倍スピードアップしました!

ここで、このプログラムのメモリ安全性を検証してみましょう。このプログラムの検証には新しい手法は必要ありません; 単純に、関数ポインタと述語族について 6.12 章と 6.15 章で見た手法を適用します。つまり、thread_run_joinable 関数ポインタ型について契約を指定し、それぞれのプログラムが必要に応じてこの契約を具体化できるように、述語族を使います。threading.h における thread_start_joinable と thread_join の仕様は次のようになります:

```
/*@

predicate_family thread_run_pre(void *thread_run)(void *data, any info);
predicate_family thread_run_post(void *thread_run)(void *data, any info);

@*/

typedef void thread_run_joinable(void *data);
    //@ requires thread_run_pre(this)(data, ?info);
    //@ ensures thread_run_post(this)(data, info);

struct thread;

/*@ predicate thread(
        struct thread *thread, void *thread_run, void *data, any info); @*/

struct thread *thread_start_joinable(void *run, void *data);
    /*@ requires
```

```
            is_thread_run_joinable(run) == true &*&
            thread_run_pre(run)(data, ?info);
      @*/
    //@ ensures thread(result, run, data, info);

void thread_join(struct thread *thread);
    //@ requires thread(thread, ?run, ?data, ?info);
    //@ ensures thread_run_post(run)(data, info);
```

関数 thread_start_joinable は、パラメータ run の値が関数ポインタ型 thread_run_joinable の契約を満たす関数ポインタであることを要求しています; さらにそれは、述語族インスタンス thread_run_pre(run) が示す、run 関数自身が要求するリソースを要求します。事前条件がこのポインタが指すデータ構造を表現できるように、この述語は data ポインタを引数として取ります。 thread_run_pre と thread_run_post の info パラメータはより高度なシナリオで使われますが、ここでは説明を省略します。関数 thread_start_joinable は、thread_join 呼び出しを検証するために要求される全ての情報を含む述語 thread を返します。関数 thread_join はこの述語 thread を取り、述語族インスタンス thread_run_post(run) で表現された run 関数が返すリソースを返します。

練習問題 21 このプログラムのメモリ安全性を検証してください。メモリの解放については気にする必要はありません; 不要となったチャンクを単にリークさせてください。

6.20 分割所有パーミッション

ここで、前章のプログラムを次のようにしたいと仮定してみましょう: 単に階乗の和を計算する代わりに、階乗の和と階乗の積の両方を計算したいのです。筆者のマシンは 2 つのコアを持つので、2 つのスレッドを走らせたいのです: 1 つは木のノードの値の階乗の和を計算し、もう 1 つは木のノードの値の階乗の積を計算します。

これを解決するために、はじめに先の tree_compute_sum_facs 関数を、引数としてノードの値を刈り取る関数を取る tree_fold 関数に一般化しましょう:

```
typedef int fold_function(int acc, int x);

int tree_fold(struct tree *tree, fold_function *f, int acc)
{
    if (tree == 0) {
        return acc;
    } else {
        acc = tree_fold(tree->left, f, acc);
        acc = tree_fold(tree->right, f, acc);
        acc = f(acc, tree->value);
        return acc;
    }
}
```

末端からの順序で木 t の要素が e1, e2, e3, e4 であった場合、tree_fold (t, f, a) は次を返します:

```
f(f(f(f(a,e1),e2),e3),e4)
```

同様に先の和を計算するスレッドは fold スレッドに一般化できます:

```
struct fold_data {
    struct thread *thread;
    struct tree *tree;
    fold_function *f;
    int acc;
};

void folder(struct fold_data *data)
{
    int acc = tree_fold(data->tree, data->f, data->acc);
    data->acc = acc;
}

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f,
    int acc)
{
    struct fold_data *data = malloc(sizeof(struct fold_data));
    struct thread *t = 0;
    if (data == 0) abort();
    data->tree = tree;
    data->f = f;
    data->acc = acc;
    t = thread_start_joinable(folder, data);
    data->thread = t;
    return data;
}

int join_fold_thread(struct fold_data *data)
{
    thread_join(data->thread);
    return data->acc;
}
```

次のように、fold スレッドを使うことで前章のプログラムを再実装できます:

```
int sum_function(int acc, int x)
{
```

```
    int f = fac(x);
    return acc + f;
}

int main()
    //@ requires true;
    //@ ensures true;
{
    struct tree *tree = make_tree(22);
    //@ open tree(tree, _);
    struct fold_data *leftData =
        start_fold_thread(tree->left, sum_function, 0);
    struct fold_data *rightData =
        start_fold_thread(tree->right, sum_function, 0);
    int sumLeft = join_fold_thread(leftData);
    int sumRight = join_fold_thread(rightData);
    int f = fac(tree->value);
    /*@ leak tree->left |-> _ &*& tree->right |-> _ &*& tree->value |-> _ &*&
            malloc_block_tree(tree); @*/
    printf("%i", sumLeft + sumRight + f);
    return 0;
}
```

練習問題 22 このプログラムのメモリ安全性を検証してください。

これで、1 つのスレッドで和を計算し、もう 1 つのスレッドで積を計算し、積と和で異なる結果を返すプログラムを書くのは簡単です:

```
int sum_function(int acc, int x)
{
    int f = fac(x);
    return acc + f;
}

int product_function(int acc, int x)
{
    int f = fac(x);
    return acc * f;
}

int main()
{
    struct tree *tree = make_tree(21);
```

```
    struct fold_data *sumData = start_fold_thread(tree, sum_function, 0);
    struct fold_data *productData =
        start_fold_thread(tree, product_function, 1);
    int sum = join_fold_thread(sumData);
    int product = join_fold_thread(productData);
    printf("%i", product - sum);
    return 0;
}
```

けれども、これまで見てきた手法を使ってこのプログラムを検証することはできません。1 度目の `start_fold_thread` 呼び出しは `tree` チャンクを消費し、その結果 2 度目の呼び出しにおける事前条件を満たせないので VeriFast はエラーになってしまいます。このシステムでは説明した通り、一度に 1 つのスレッドだけが特定のメモリチャンクを所有できます。したがって、もし和を計算するスレッドが木を所有していると、積を計算するスレッドはそれを同時にアクセスすることはできないのです。けれども、両スレッドが木を読むだけでその木を変更しないのであれば、実際にはこれは安全です。

VeriFast は **分割所有パーミッション** (*fractional permissions*) を使うことで、読み出し専用の共有メモリチャンクをサポートしています。VeriFast のシンボリックヒープ中のそれぞれのチャンクは、ゼロより大きく 1 以下の実数であるような、1 つの **係数** (*coefficient*) を持っています。デフォルトの係数は 1 で、見えません。もしチャンクの係数が 1 でなければ、それは角括弧で囲われてチャンクの左に表示されます。係数 1 のチャンクは **フルパーミッション** (*full permission*) を表わし、それはすなわち **読み書きパーミッション** (*read-write permission*) です。係数が 1 より小さいチャンクは **分割所有パーミッション** (*fractional permissions*) を表わし、それはすなわち **読み出し専用パーミッション** (*read-only permission*) です。

`tree_fold` 関数は木を変更しないので、`tree` チャンクの断片のみを要求します。

```
int tree_fold(struct tree *tree, fold_function *f, int acc)
    //@ requires [?frac]tree(tree, ?depth) &*& is_fold_function(f) == true;
    //@ ensures [frac]tree(tree, depth);
{
    if (tree == 0) {
        return acc;
    } else {
        //@ open [frac]tree(tree, depth);
        acc = tree_fold(tree->left, f, acc);
        acc = tree_fold(tree->right, f, acc);
        acc = f(acc, tree->value);
        return acc;
        //@ close [frac]tree(tree, depth);
    }
}
```

`tree` チャンクの任意の小さな断片に関数を適用できるように、係数の位置においてパターンを

使用していることに注意してください。また、**open** と **close** 文で断片を指定していることにも注意してください。シンボリックヒープ中にある断片はデフォルトでどんな断片であっても開くので **open** 文は必要ではありませんが、デフォルトではチャンクを係数 1 で閉じようとするので **close** 文は必要です。

練習問題 23 このプログラムのメモリ安全性を検証してください。木のチャンクの 1/2 の断片のみを要求するように、関数 `start_fold_thread` を修正してください。注意: 少数表記はサポートされていません; 代わりに分数表記を使ってください。

6.21 正確な述語

前章のプログラムは大量にメモリをリークしています。このリークはプログラムが終了する前にのみ起きるだけなので、これは構わないでしょう。けれども、このプログラムがもっと大きいプログラムの一部で、実行時間が長いと仮定してみましょう; この場合、このようなリークの除去は重要になります。そこでこの章では、前章のプログラムから **leak** コマンドを除去してみましょう。

はじめに、全ての動的確保したメモリをきっちり解放するように、C 言語プログラムを修正しなければなりません。関数 `dispose_tree` を導入し、関数 `join_fold_thread` と `main` を修正する必要があります:

```
void dispose_tree(struct tree *tree)
{
    if (tree != 0) {
        dispose_tree(tree->left);
        dispose_tree(tree->right);
        free(tree);
    }
}

int join_fold_thread(struct fold_data *data)
{
    thread_join(data->thread);
    int result = data->acc;
    free(data);
    return result;
}

int main()
{
    struct tree *tree = make_tree(21);
    struct fold_data *sumData = start_fold_thread(tree, sum_function, 0);
    struct fold_data *productData =
         start_fold_thread(tree, product_function, 1);
    int sum = join_fold_thread(sumData);
    int product = join_fold_thread(productData);
    dispose_tree(tree);
```

```
    printf("%i", product - sum);
    return 0;
}
```

注釈の修正は少しトリッキーです。はじめに簡単な部分を見てみましょう: 関数 join_fold_thread 中の free 文の検証です。それは既に要求されたパーミッションのほとんどを持っています; 唯一 malloc_block_fold_data チャンクが欠落しています。このチャンクは関数 start_fold_thread 中でリークされています。**leak** 文を削除し、start_fold_thread の **ensures** 節と join_fold_thread の **requires** 節にそのチャンクを追加することでこの問題を解決できます; これで free 文は検証されました。

これで残りは関数 main 中の dispose_tree 呼び出しの検証になりました。木に対するフルパーミッション、すなわち tree(tree, _) チャンクが必要です。それぞれの join_fold_thread 呼び出しは [1/2]tree(tree, _) チャンクをリークしていることに注意してください。このチャンクをリークする代わりに呼び出し元に戻すために、関数 join_fold_thread の注釈を修正する必要があります。けれども現状では、join_fold_thread の契約中でその木を特定する方法がありません。また main において、そのチャンクが処分する特定の木に関連することを知らないので、[1/2]tree(_, _) チャンクは役に立ちません。

この問題を解決するために、その木が関数 join_fold_thread の事前状態中でフィールド data->tree によって指されていることに注意してください。けれども、このフィールドに対するフルパーミッションを folder スレッドに渡してしまったので、関数 join_fold_thread の事前条件中でこのフィールドに言及できません。

解決策は、フィールド data->tree を表わすパーミッションの断片のみを fold スレッドに渡し、main スレッド中で残りの断片を保有することです。これを行なうために、述語族インスタンス thread_run_pre と thread_run_post、関数 start_fold_thread と join_fold_thread の契約を修正しましょう:

```
/*@

predicate_family_instance
    thread_run_pre(folder)(struct fold_data *data, any info) =
    [1/2]data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
    data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;
predicate_family_instance
    thread_run_post(folder)(struct fold_data *data, any info) =
    [1/2]data->tree |-> ?tree &*& [1/2]tree(tree, _) &*&
    data->f |-> ?f &*& is_fold_function(f) == true &*& data->acc |-> _;

@*/

struct fold_data *start_fold_thread(struct tree *tree, fold_function *f,
                                    int acc)
    //@ requires [1/2]tree(tree, _) &*& is_fold_function(f) == true;
    /*@
```

```
    ensures
        [1/2]result->tree |-> tree &*& result->thread |-> ?t &*&
        thread(t, folder, result, _) &*& malloc_block_fold_data(result);
    @*/
int join_fold_thread(struct fold_data *data)
    /*@
    requires
        [1/2]data->tree |-> ?tree &*& data->thread |-> ?t &*&
        thread(t, folder, data, _) &*& malloc_block_fold_data(data);
    @*/
    //@ ensures [1/2]tree(tree, _);
```

これで関数 join_fold_thread の事後条件中で [1/2]tree(tree, _) チャンクを指定できます。

ここで関数 main は dispose_tree 呼び出しにおいて、2 つの [1/2]tree(tree, _) チャンクを持つことに注意してください。2 つの半分のチャンクを持ち、必要なのは 1 つのフルチャンクです; なぜ VeriFast は単純にこの 2 つの半分チャンクを結合してくれないのでしょうか？

その理由は 2 つの断片チャンクを 1 つのチャンクに結合するのが常に健全 (すなわち安全) であるとは限らないからです。例えば、次のプログラムは検証できます:

```
/*@

predicate foo(bool b) = true;

predicate bar(int *x, int *y) = foo(?b) &*& b ? integer(x, _) : integer(y, _);

lemma void merge_bar() // This lemma is false!!
    requires [?f1]bar(?x, ?y) &*& [?f2]bar(x, y);
    ensures [f1 + f2]bar(x, y);
{
    assume(false);
}

@*/

int main()
    //@ requires true;
    //@ ensures true;
{
    int x, y;
    //@ close [1/2]foo(true);
    //@ close [1/2]bar(&x, &y);
    //@ close [1/2]foo(false);
    //@ close [1/2]bar(&x, &y);
    //@ merge_bar();
```

```
    //@ open bar(&x, &y);
    //@ assert foo(?b);
    //@ if (b) integer_unique(&x); else integer_unique(&y);
    assert(false);
}
```

このプログラムは、与えられたチャンクの 2 つの断片は 1 つのチャンクに結合できるという仮定が、不健全性を引き起こすことを示しています (すなわち実行時に表明に失敗するプログラムの検証に成功してしまいます)。このプログラムは prelude.h の次の補題を使っています:

```
lemma void integer_unique(int *p);
    requires [?f]integer(p, ?v);
    ensures [f]integer(p, v) &*& f <= 1;
```

この問題は 2 つの [1/2]bar(&x, &y) チャンクが同じメモリ領域を表わさないことに由来します: 1 つは [1/2]integer(&x, _) を表わし、もう 1 つは [1/2]integer(&y, _) を表わします。これらの結合は integer(&x, _) も integer(&y, _) も生み出しません。

けれども、両方の断片が同じメモリ領域を表わすのであれば、2 つの断片を結合するのは健全です。これをサポートするために、VeriFast は **正確な述語** (*precise predicates*) の表記をサポートしています: 述語のパラメータリスト中の**入力パラメータ** (*input parameters*) と **出力パラメータ** (*output parameters*) の間のコンマの代わりにセミコロンを書くことで、述語を **正確** (*precise*) に宣言できます。このような述語に対して、VeriFast は、同じ入力引数をともなう 2 つのチャンクが同じメモリ領域を表わし、常に同じ出力引数を持つことをチェックします。VeriFast は同じ入力引数を持つような正確な述語の断片を自動的に結合します。

例えば、述語 integer 自身は正確な述語です; それは prelude.h において次のように宣言されています:

```
predicate integer(int *p; int v);
```

この宣言は述語 integer が正確で、1 つの入力パラメータ p と 1 つの出力パラメータ v を持つことを指定しています。 VeriFast が 2 つのチャンク [f1]integer(p1, v1) と [f2]integer(p2, v2) を見つけて、p1 と p2 が等しいことを証明できると、それらのチャンクを 1 つのチャンク [f1 + f2]integer(p1, v1) に結合し、v1 と v2 が等しい仮定を生成します。

結論として、このプログラム例を検証するためには、述語 tree を正確に宣言する必要があります:

```
predicate tree(struct tree *t; int depth) =
    t == 0 ?
        depth == 0
    :
        t->left |-> ?left &*& t->right |-> ?right &*& t->value |-> _ &*&
        malloc_block_tree(t) &*& tree(left, ?childDepth) &*&
        tree(right, childDepth) &*& depth == childDepth + 1;
```

VeriFast の正確さ解析によって受け入れられるように、述語の本体を少し書き換えたことに注意してください。これでこのプログラムは検証できます。

正確さの解析は、述語の本体が入力パラメータについて正確であり、出力パラメータが固定であることをチェックします。固定の値 *X* の集合において様々な形式の表明が正確であるための規則は次のようになります:

- もし述語が正確でその入力引数が *X* 固定であれば、その述語の表明は *X* において正確です; 出力引数として現われるどのような値も固定になります。例えば、表明 `p(x + y, z)` が正確であると考えられれば、`p` は正確な述語です。さらに、それは 1 つの入力パラメータを持つと仮定します。すると `x` と `y` は *X* 中で、その表明は `z` 固定になります。
- もし係数が *X* で固定か、その係数がダミーのパターンで、本体が *X* で正確であるなら、その断片の表明は正確です; その本体で固定化されているどのような変数も固定になります。
- ブール表明はどのような *X* においても常に正確です。それが等式で、その左辺が変数で、その右辺が *X* で固定されていない限り、どのような変数も固定にはなりません; そのような例外においては、その左辺の変数は固定になります。
- もし条件が *X* で固定されていて、それぞれの分岐が *X* において正確であるなら、その条件付き表明は *X* において正確です; その固定の変数は全ての分岐によって固定の変数です。
- 同様に、もしオペランドが *X* で固定で、それぞれの分岐が *X* の和集合とコンストラクタ引数において正確なら、その **switch** 表明は正確です; その固定の変数は全ての分岐によって固定の変数です。
- 1 番目のオペランドが *X* において正確で、2 番目のオペランドが *X* の和集合において正確で、変数が 1 番目のオペランドで固定なら、その分離積は正確です。どちらのオペランドでも変数は固定です。

つまりこの解析は、固定の変数の集合を追跡するために、表明を左から右に横断します。入れ子になった述語の入力引数と分岐条件は、すでに固定である変数にのみ依存しなければなりません; 入れ子になった述語の表明の出力引数としてか、もしくは右辺が固定であるような等式の左辺に現われる変数は、固定の変数の集合に追加されます。

6.22 自動 open/close

これまでの章では、同じチャンクの断片を結合できるようにするために、正確な述語に対する VeriFast の機能を紹介しました。けれども、述語を正確に宣言するのには別の利点があります: 自動的にその述語を開いたり閉じたりする VeriFast の論理が有効になるのです。これは明示的に書かなければならない **open** と **close** コマンドの数を削減します。

特に VeriFast が述語の表明を消費していて、シンボリックヒープにマッチするチャンクが無く、けれどもその述語が正確で、全ての入力引数がその表明で指定されていたら、自動 open と自動 close が試行されます。述語の本体に現れるチャンクがシンボリックヒープに見つかったら、自動 close が行なわれます; 所望のチャンクがシンボリックヒープに有るチャンクの本体に現われたら、自動 open が行なわれます。

例えば、これまでの章でのプログラムにおいて、`tree` チャンクを開いたり閉じたりする全てのゴーストコマンドは削除できます。

6.23 Mutex

1 秒に一度、2 つのセンサーをモニタし、両方のセンサーで検出したパルスの数の合計を印字するようなプログラムを書きたいと仮定しましょう。API 関数 `wait_for_pulse` を使って、与えられたセンサーからのパルスを待ち合わせることができます。パルスは両方のセンサーから同時にやって

くるので、別々のスレッドでそれぞれのセンサーからのパルスを待ち合わせる必要があります。パルスがセンサーから届いたときはいつでも、一致するスレッドがスレッド間で共有された 1 つのカウンタをインクリメントします。main スレッドは毎秒に 1 回、そのカウンタの値を印字します。

この例のカウンタのように、複数のスレッドが同時に同じ変数にアクセスすると、それらのアクセスを同期させる必要があります; さもなければ、2 つの並列アクセスは干渉し、それらのアクセスが交互に行なわれた際の結果とは異なる結果を引き起こすかもしれません。ほとんどのプログラミングプラットフォームは様々な方法でスレッドを同期させるために、様々な同期機構を提供しています。

おそらく最も一般的な同期機構は、ロックもしくは mutex としても知られる、相互排他ロックです。どの時点においても、mutex は 2 つの状態の 1 つを取ります; それはフリーかなんらかのスレッドによって保持されるかどちらかです。 mutex は 1 つ以上のスレッドによって保持されることはありません。 mutex は 2 つの操作を提供します; 獲得と解放です。スレッドが mutex を獲得しようと試みた場合、2 つの場合があります。

- もしその mutex がフリーなら、その試みは成功し、スレッドは解放するまでその mutex を保持します。
- もしその mutex が別のスレッドによって保持されていたら、その試みを行なったスレッドはその mutex がフリーになるまでブロックします。
- もしその mutex が獲得を試みるスレッドによって既に保持されていたら、その結果は mutex がリエントラントか非リエントラントかに依存します。もしその mutex がリエントラントなら、その試みは成功し、スレッドはさらにそのロックを保持します。もしその mutex が非リエントラントなら、その試みは失敗します。これは、スレッドが永遠にブロックするか、エラーが発生してプログラムが終了するかどちらかであることを意味しています。

C 言語はそれ自身ではどのような同期機構も提供しません; オペレーティングシステムの責務です。例えば、Windows API はクリティカルセクションと呼ばれる mutex 機構を提供し、Linux API は mutex と呼ばれる mutex 機構を提供します。全てのプラットフォーム横断の同一インターフェイスのために、VeriFast は 2 つの同期機構、mutex と lock、を提供するモジュール `threading.c` を持っています。これら 2 つの唯一の違いは mutex は非リエントラントで、lock はリエントラントであることです。 mutex の方が使いやすいので、この例では mutex を使います。

次はプログラム例のソースコードです:

```
#include "stdlib.h"
#include "threading.h"

void wait_for_pulse(int source);
void sleep(int millis);
void print_int(int n);

struct counter {
    int count;
    struct mutex *mutex;
};

struct count_pulses_data {
```

```
    struct counter *counter;
    int source;
};

void count_pulses(struct count_pulses_data *data) {
    struct counter *counter = data->counter;
    int source = data->source;
    free(data);

    struct mutex *mutex = counter->mutex;

    while (true) {
        wait_for_pulse(source);
        mutex_acquire(mutex);
        counter->count++;
        mutex_release(mutex);
    }
}

void count_pulses_async(struct counter *counter, int source) {
    struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
    if (data == 0) abort();
    data->counter = counter;
    data->source = source;
    thread_start(count_pulses, data);
}

int main() {
    struct counter *counter = malloc(sizeof(struct counter));
    if (counter == 0) abort();
    counter->count = 0;
    struct mutex *mutex = create_mutex();
    counter->mutex = mutex;

    count_pulses_async(counter, 1);
    count_pulses_async(counter, 2);

    while (true) {
        sleep(1000);
        mutex_acquire(mutex);
        print_int(counter->count);
        mutex_release(mutex);
    }
}
```

このプログラムでは、スレッドを待ち合わせないので、thread_start_joinable の代わりにスレッド API 関数 thread_start を使います。関数 thread_start の仕様は、6.19 章で議論した関数 thread_start_joinable の仕様に似ています; それはヘッダファイル threading.h で次のように与えられます:

```
//@ predicate_family thread_run_data(void *thread_run)(void *data);

typedef void thread_run(void *data);
    //@ requires thread_run_data(this)(data);
    //@ ensures true;

void thread_start(void *run, void *data);
    //@ requires is_thread_run(run) == true &*& thread_run_data(run)(data);
    //@ ensures true;
```

mutex を使ってプログラミングすると、ミスを犯しがちです。最も悪い問題は、保護されるべきデータ構造にアクセスする前に mutex を獲得することをプログラマが忘れることです。その結果として誤った結果が得られ、原因を調べるのは困難になるかもしれません。

幸運にも、VeriFast はこれらのトリッキーなバグの捕捉を助けてくれます。これまでの章から、VeriFast によるチェックの結果として、それぞれのスレッドは所有するメモリ位置にのみアクセスでき、2 つのスレッドが同時に同じメモリ位置を (フルに) 所有することはできないことを思い出してください。これは並列アクセスによる干渉を防止します。けれども、それならどうやってスレッドは可変の変数を共有できるのでしょうか？ もちろんその答は mutex を使うことです。 mutex が生成されると、それは**ロック不変条件** (*lock invariant*) によって指定された、メモリ位置の集合の所有権を取ります。スレッドが mutex を獲得すると、その mutex によって所有されたメモリ位置は、スレッドがその mutex を解放するまで、そのスレッドによって所有されるようになります。その mutex が解放されると、そのメモリ位置は再びその mutex の所有になります。この方法で mutex を共有することで、スレッドはうまく同期して間接的に任意のメモリ位置を共有出来ます。

threading.c によって提供される mutex 関数は threading.h で次のように指定されます:

```
struct mutex;

/*@
predicate mutex(struct mutex *mutex; predicate() p);
predicate mutex_held(struct mutex *mutex, predicate() p, int threadId,
    real frac);
predicate create_mutex_ghost_arg(predicate() p) = true;
@*/

struct mutex *create_mutex();
    //@ requires create_mutex_ghost_arg(?p) &*& p();
    //@ ensures mutex(result, p);

void mutex_acquire(struct mutex *mutex);
```

```
    //@ requires [?f]mutex(mutex, ?p);
    //@ ensures mutex_held(mutex, p, currentThread, f) &*& p();

void mutex_release(struct mutex *mutex);
    //@ requires mutex_held(mutex, ?p, currentThread, ?f) &*& p();
    //@ ensures [f]mutex(mutex, p);

void mutex_dispose(struct mutex *mutex);
    //@ requires mutex(mutex, ?p);
    //@ ensures p();
```

mutex を生成するとき、その mutex によって所有されるメモリ位置を**ロック不変条件** (*lock invariant*) で指定する必要があります。それは **述語値** (*predicate value*) (6.17 章を見てください) を指定することで行ないます。実関数 `create_mutex` は実引数としてその述語値を取れないので、この関数はその述語値を述語 `create_mutex_ghost_arg` の引数の形でゴースト引数として取ります。この述語はこの目的のためにだけ存在します。つまり、`create_mutex` を呼び出す前に、`create_mutex_ghost_arg` チャンクを閉じる必要があり、その引数は新しい mutex のロック不変条件を指定する述語の名前です。 `create_mutex` 呼び出しはそのゴースト引数チャンクとそのロック不変条件チャンクを消費し、その mutex を現わす `mutex` チャンクを生成します。このチャンクはその第二引数としてロック不変条件を指示します。

複数のスレッドに mutex の共有を許すために、mutex チャンクは複数の断片に分割できます。mutex を獲得するために、その mutex チャンクのたった 1 つの断片が要求されます。 mutex が獲得されると、`mutex` チャンク断片は `mutex_held` チャンクに変化します。この `mutex_held` チャンクは mutex とロック不変条件を指定するだけでなく、mutex を獲得したスレッドと、mutex の獲得に使用された mutex チャンク断片の係数も指定します。`mutex_acquire` 呼び出しは、mutex によって所有されたメモリ位置へのアクセスをスレッドに与える、ロック不変条件も追加で生成します。

`mutex_release` 呼び出しは、ロック不変条件と現在のスレッドに対する `mutex_held` チャンクを消費し、ロックを獲得するために使われた元の `mutex` チャンクを生成します。

必要なら、プログラムが mutex を使い終わると、全ての mutex チャンク断片を集めて mutex を処分することができます; これによって mutex を処分したスレッドにロック不変条件の所有権が返ります。

練習問題 24 このプログラム例を検証してください。

6.24 リークとダミー断片

前章のプログラム例からはじめましょう。このプログラムでは、パルスの発信源の数は固定でした。ここで、発信源が動的に接続されたり切断されたりすると仮定してみましょう。初期状態では発信源はありません。接続される新しい発信源を待ち合わせて、その識別子を返す API 関数 `wait_for_source` があるとします。さらに API 関数 `wait_for_pulse` がブール値を返すとします。その結果が偽なら、新しいパルスが検出されたことを意味します; その結果が真なら、その発信源が切断されたことを意味します。プログラムの目的は、全ての発信源からのパルスの合計数を数えて、毎秒に 1 度それを印字することのままです。次のプログラムはこれを実装しています:

```
#include "stdlib.h"
#include "threading.h"

int wait_for_source();
bool wait_for_pulse(int source); // true means the sensor has been removed.
void sleep(int millis);
void print_int(int n);

struct counter {
    int count;
    struct mutex *mutex;
};

struct count_pulses_data {
    struct counter *counter;
    int source;
};

void count_pulses(struct count_pulses_data *data) {
    struct counter *counter = data->counter;
    int source = data->source;
    free(data);

    struct mutex *mutex = counter->mutex;
    bool done = false;
    while (!done) {
        done = wait_for_pulse(source);
        if (!done) {
            mutex_acquire(mutex);
            counter->count++;
            mutex_release(mutex);
        }
    }
}

void count_pulses_async(struct counter *counter, int source) {
    struct count_pulses_data *data = malloc(sizeof(struct count_pulses_data));
    if (data == 0) abort();
    data->counter = counter;
    data->source = source;
    thread_start(count_pulses, data);
}
```

```
void print_count(struct counter *counter) {
    struct mutex *mutex = counter->mutex;
    while (true) {
        sleep(1000);
        mutex_acquire(mutex);
        print_int(counter->count);
        mutex_release(mutex);
    }
}

int main() {
    struct counter *counter = malloc(sizeof(struct counter));
    if (counter == 0) abort();
    counter->count = 0;
    struct mutex *mutex = create_mutex();
    counter->mutex = mutex;

    thread_start(print_count, counter);

    while (true) {
        int source = wait_for_source();
        count_pulses_async(counter, source);
    }
}
```

ここでは mutex は任意の多くのスレッド間でいつでも共有されうる一方で、前章のプログラム例では mutex は 3 つのスレッド間で共有されていたことに注意してください。これは検証に対して影響を及ぼします: 前章では mutex チャンクの 3 分の 1 をそれぞれのスレッドに与えれば済む一方で、この例での断片の分割はより複雑です。

さらに別の問題もあります: `count_pulses` スレッドが終了したとき、それはまだ mutex の断片を所有しています。VeriFast はリークチェックでこれをエラーにします。一般に、mutex のリークはプログラムのメモリ不足を最終的に引き起こす可能性があります。けれども、このプログラムの場合は、1 つの mutex のリークは問題にはなりません。そして mutex チャンク断片はどのみちリークされ、その mutex を破棄するために再構築されることはありません。そのためその断片の係数を慎重に追跡する必要はないのです。

VeriFast は、多くのスレッド間で共有され、決して再構築されないチャンクの取り扱いを簡単にする**ダミー断片** (*dummy fractions*) と呼ばれる機能を持っています具体的には、**leak** ゴーストコマンドをチャンクに適用すると、そのチャンクを削除せずに、そのチャンクの係数を**ダミー断片係数シンボル** (*dummy fraction coefficient symbol*) で単純に置換します。係数がダミー断片係数シンボルであるようなチャンクは **ダミー断片** (*dummy fraction*) と呼ばれます。 VeriFast はそれぞれの関数の最後でリークチェックを行なうとき、ダミー断片でないチャンクのみをエラーにします。

ダミー断片は `[_]chunk(args)` のようなダミーパターンを使った表明で表わされます。ダミー係数をともなう表明を消費すると、そのマッチしたチャンクはダミー断片であるべきで、そうでなければそのチャンクのマッチはそのチャンクを暗黙的にリークするので VeriFast はエラーを報告しま

す。ダミー係数をともなう表明を生成すると、生成されたチャンクはダミー断片になります。

任意のダミー断片の共有を簡単にするために、ダミー断片表明を消費すると、VeriFast はマッチしたチャンクを自動的に 2 つに分割します: その 1 つは消費され、もう 1 つはシンボリックヒープ中に残ります。

練習問題 25 このプログラム例を検証してください。mutex チャンクを生成した後、それを直接リークしてください。ダミーパターンを使って、表明中で mutex チャンクの断片の係数を表わしてください。

6.25 文字配列

標準入力から固定数の文字を読み、それらを二度表示するプログラムを検証してみましょう。次は 5 文字読むバージョンです:

```
char getchar();
void putchar(char c);

int main()
{
    char c1 = getchar();
    char c2 = getchar();
    char c3 = getchar();
    char c4 = getchar();
    char c5 = getchar();
    for (int i = 0; i < 2; i++) {
        putchar(c1);
        putchar(c2);
        putchar(c3);
        putchar(c4);
        putchar(c5);
    }
    return 0;
}
```

このプログラムは動作します。 (getchar の返り値の型を少し単純化しています; その本当の関数は int を返します。けれどもこのバージョンは正常にコンパイルされ、上手く動作するべきです。) このプログラムを実行して Hello と入力すると、HelloHello が返ります。

けれども明確に、このアプローチはより多い文字数に対して実用的ではありません。より多い文字数に対して有効なアプローチの 1 つは文字の配列を使うことです。標準 C 言語関数 malloc を使って配列を確保し、それから再帰関数を使って文字を読み書きしましょう。差し当たり、プログラムの最後でその配列を解放することについては気にしません。最初に 5 文字を読むバージョンを再び作ってみましょう。

```
char getchar();
void putchar(char c);
```

```
char *malloc(int count);

void getchars(char *start, int count) {
    if (count > 0) {
        char c = getchar();
        *start = c;
        getchars(start + 1, count - 1);
    }
}

void putchars(char *start, int count) {
    if (count > 0) {
        char c = *start;
        putchar(c);
        putchars(start + 1, count - 1);
    }
}

int main() {
    char *array = malloc(5);
    getchars(array, 5);
    putchars(array, 5);
    putchars(array, 5);
    return 0;
}
```

このプログラムは動作します。(ここでは `malloc` の失敗を無視しています。) このプログラムを検証してみましょう。

プログラムに対して単純に VeriFast を実行して、エラー箇所を見るのは常に良いアプローチです。上記のプログラムを検証すると、VeriFast は関数群が契約を持たないことをエラーにします。それぞれの関数に最も単純な実行可能な契約を与えてみましょう:

```
//@ requires true;
//@ ensures true;
```

これで VeriFast は関数 `getchars` の次の行でエラーになります:

```
    *start = c;
```

この文は文字 `c` をアドレス `start` のメモリ位置に書き込んでいます。VeriFast はこのような文を見つけると、その関数がその位置への書き込みパーミッションを持つかチェックします。具体的には、`character(start, _)` にマッチするチャンクがシンボリックヒープ中に存在するかチェックします。この場合、`getchars` の requires 節にそのチャンクが書かれておらず、この関数はどこか

別の所からそれを獲得もしていないので、そのようなチャンクがシンボリックヒープ中に存在しません。

この問題を解決するためには、少なくともパラメータ count がゼロより大きいなら、関数 getchars を呼び出すどんな関数もアドレス start の位置への書き込みパーミッションを与えなければならないことを、関数 getchars の requires 節で指定する必要があります。良いマナーとして、ensures 節に指定することで、使い終わったらそのパーミッションを呼び出し元に戻します:

```
//@ requires count > 0 ? character(start, _) : true;
//@ ensures count > 0 ? character(start, _) : true;
```

再び VeriFast を実行すると、VeriFast が *start への代入を受け付けます。けれども今度は、getchars の再帰呼び出しがエラーになります。実際、count が 1 より大きいと、この再帰呼び出しにはアドレス start + 1 の位置にアクセスするためのパーミッションが必要になります。契約を拡張してこれを反映してみましょう:

```
/*@ requires count > 0 ?
        character(start, _) &*& (count > 1 ? character(start + 1, _) : true)
    :
        true;
  @*/
/*@ ensures count > 0 ?
        character(start, _) &*& (count > 1 ? character(start + 1, _): true)
    :
        true;
  @*/
```

悲しいかな、再び VeriFast を実行すると、当該の再帰呼び出しで再び VeriFast はエラーになります。 (Verify メニューで arithmetic overflow checking が無効にすることを思い出してください。) 実際、count が 2 より大きいと、この再帰呼び出しは、アドレス start + 2 の位置へアクセスするためのパーミッションを必要とします。

再度、契約を修正することはできますが、どこまで行なえば済むのでしょうか？ 戻って getchars が行なっていることを考えると、呼び出し getchars(start, count) が start から start + count - 1 の領域の位置へのアクセスするためのパーミッションを要求していることに気が付きます。どうすればこれを表現できるでしょうか？ もし 5 文字までのみをサポートするなら、次のような事前条件を使えます:

```
/*@
requires
    count <= 0 ? true :
        character(start, _) &*&
        (count - 1 <= 0 ? true :
            character(start + 1, _) &*&
            (count - 2 <= 0 ? true :
```

```
                character(start + 2, _) &*&
                (count - 3 <= 0 ? true :
                    character(start + 3, _) &*&
                    (count - 4 <= 0 ? true :
                        character(start + 4, _) &*&
                        (count - 5 <= 0 ? true :
                            false)))));
@*/
```

事後条件でも同じ表明を使うと、関数 getchars は検証できます。さらに putchars に対して同じ契約を使うと、その関数も検証できます。そして変数名 start を result で置き換えることで得られる、malloc の事後条件として同じ表明を使えます。これで残る問題は、関数 main の最後での 5 つのメモリ位置をリークしている VeriFast のエラーだけです。関数の最後に次のゴースト文を挿入することで、そのリークを許容することを VeriFast に指示できます:

```
/*@
leak
character(array, _) &*& character(array + 1, _) &*& character(array + 2, _) &*&
character(array + 3, _) &*& character(array + 4, _);
@*/
```

これでこのプログラムは検証できます。素晴しい!

練習問題 26 100 文字を読めるように、このプログラムを修正してください。

この練習問題は終わりましたか？ まだでしょうか？ あなたが悪いのではありません。もちろん大量にこれらの契約を書くのは困難です。

解決策は **再帰的な述語** (*recursive predicates*) を使うことです。 getchars の事前条件は再帰的な構造を持っていることに注意してください。表明に名前をつけ、その表明自身の中でこの名前を使うことで、無限に続く表明を作ることができます:

```
predicate characters(char *start, int count) =
    count <= 0 ?
        true
    :
        character(start, _) &*& characters(start + 1, count - 1);
```

この述語を 5 回展開、すなわち characters の出現をその定義で 5 回置換することで、getchars の事前条件にとても近いものを得ることができます。唯一の違いは **false** の代わりに characters(start + 5, count - 5) が得られることです。つまり characters は 5 文字では止まりません; 無限に続くのです。

契約中で characters を使うことで、練習問題 26 を解いてみましょう:

```
char getchar(); /*@ requires true; @*/ /*@ ensures true; @*/
void putchar(char c); /*@ requires true; @*/ /*@ ensures true; @*/
```

```
/*@
predicate characters(char *start, int count) =
    count <= 0 ?
        true
    :
        character(start, _) &*& characters(start + 1, count - 1);
@*/

char *malloc(int count);
    //@ requires true;
    //@ ensures characters(result, count);

void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    if (count > 0) {
        char c = getchar();
        *start = c;
        getchars(start + 1, count - 1);
    }
}

void putchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    if (count > 0) {
        char c = *start;
        putchar(c);
        putchars(start + 1, count - 1);
    }
}

int main() /*@ requires true; @*/ /*@ ensures true; @*/
{
    char *array = malloc(100);
    getchars(array, 100);
    putchars(array, 100);
    putchars(array, 100);
    //@ leak characters(array, 100);
    return 0;
}
```

これはまだ完全には検証できません。このプログラムでは、characters チャンクをその定義で、もしくはその逆を VeriFast は自動的に置換しないのです。 **open** と **close** ゴーストコマンドを挿入することで、明示的にそれを行なう必要があります。例えば、上記のプログラムに VeriFast を実行すると、チャンクへのマッチ character(start, _) が見つからないので、VeriFast は関数 getchars 中の *start への代入でエラーになります。このチャンクは実際には存在します; それは characters チャンクの内側に隠れているのです。 VeriFast が内側の character チャンクを見つけられるように、その characters チャンクを開く必要があります。関数 getchars の次のバージョンは検証できます:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    if (count > 0) {
        //@ open characters(start, count);
        char c = getchar();
        *start = c;
        getchars(start + 1, count - 1);
        //@ close characters(start, count);
    }
}
```

関数 putchars に同じコマンドを挿入することで、練習問題 26 への正しい回答が得られます。

練習問題 27 単純な暗号/復号プログラムを実装して検証してください。このプログラムは標準入力から 10 文字の 2 つの配列を読み込みます。それから 1 つ目の配列のそれぞれの文字を、その文字と 2 つ目の配列の関連する文字の XOR で置き換えます。これを行なう再帰関数を作ってください。この関数はその 1 つ目の配列を標準出力に書き込みます。 2 つの文字 c1 と c2 の XOR は C 言語では (char)(c1 ^ c2) のように書けます。

6.26 配列に対するループ

標準入力から 100 文字の列を読み、それを標準出力に二度書く前章のプログラム例は正しいものです。けれども、一千万文字を読もうとすると、恐らく動作しないでしょう。なぜなら関数 getchars と putchars は一千万回の再帰呼び出しを実行してしまうからです。これではコールスタックを使いつくしてしまうかもしれません。(これは C 言語コンパイラが末尾再帰最適化を行なわなかった場合です。しかし、C 言語標準はコンパイラにそれを要求していません。) そこで、これらの関数が再帰の代わりにループを使えるように、これらをより安全に書き直します。関数 getchars を書き直してみましょう:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    for (int i = 0; i < count; i++) {
```

```
        char c = getchar();
        *(start + i) = c;
    }
}
```

このプログラムを検証しようとすると、VeriFast はループ不変条件が必要だというエラーになります。ループの任意の反復の開始を表わすシンボリック状態から開始して、ループ本体を一度で検証するために、ループ不変条件が必要です。ループ不変条件はこのシンボリック状態を表わします。具体的には、シンボリックヒープの中身と、ループによって変更されるローカル変数の値に関して要求される情報をそれは表わします。(ループに関する情報詳細は 6.8 章を見てください。)

この例では、それぞれのループ反復の開始において、シンボリックヒープは `characters` チャンクを含み、変数 `i` の値は非負です。これを次のようにエンコードしましょう:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    for (int i = 0; i < count; i++)
        //@ invariant characters(start, count) &*& 0 <= i;
    {
        char c = getchar();
        *(start + i) = c;
    }
}
```

今度は VeriFast はシンボリックヒープ中のアドレス `start + i` に文字を書くためのパーミッションが見つからないというエラーになります。実際にはこのパーミッションは存在しますが、`characters(start, count)` チャンクの内側に隠れています。簡単な場合なら、内側のパーミッションを見せるためには、単純にチャンクを開けば十分です。けれどもこの場合、パーミッションは述語 `characters` の多層のレイヤーの中に隠れています。実際、パーミッションはきっちり `i + 1` 層のレイヤーの中に隠れています。パーミッションを見せるために、`i + 1` 回の open 操作を要求してしまいます。何回操作を書けば良いかわからないので、プログラムテキスト中にこのような操作を直接書くことはできません。

解決策は、VeriFast が探すパーミッションが表面に来るような方法で、`characters(start, count)` チャンクをチャンクの同等な集合に書き換えることです。`characters(start, count)` チャンクは次のチャンクのペアと同じメモリパーミッションの集合を表わすことに注意してください:

```
characters(start, i) &*& characters(start + i, count - i)
```

もしこのチャンクをこのような形式に書き換えられたら、`characters(start + i, count - i)` チャンクを単純に開けば `character(start + i, _)` パーミッションが得られます。`i` 文字を `characters(start, i)` チャンクに結合するために、はじめに `i` 回 open 操作を行ない、最初の `i` 文字を剥き出しにし、そして `i + 1` 回 close 操作を行なうことで、この書き換えができます。これ

らの操作を行なうために、ヘルパー関数を書くことができます。ゴースト操作をのみ行なう役目を持つヘルパー関数は **補題関数** (*lemma function*) として書くのが良いでしょう。補題関数は通常の C 言語関数と似ていますが、**lemma** キーワードで開始し、注釈の中に書きます:

```
/*@
lemma void split_characters_chunk(char *start, int i)
    requires characters(start, ?count) &*& 0 <= i &*& i <= count;
    ensures characters(start, i) &*& characters(start + i, count - i);
{
    if (i == 0) {
        close characters(start, 0);
    } else {
        open characters(start, count);
        split_characters_chunk(start + 1, i - 1);
        close characters(start, i);
    }
}
@*/
```

通常の関数と同じく、補題関数は契約と本体を持ちます。上記の補題関数の契約 split_characters_chunk は、この関数が 1 つの文字チャンクを要求し、値 i がゼロ以上 count 以下であり、2 つのチャンクを戻すことを表明しています: 1 つは最初の i 文字を含み、もう 1 つは残りの count - i 文字を含みます。

この関数の実装では最初に i がゼロに等しいかチェックします。もし等しければ、入力のチャンクは返されるべき 2 つ目のチャンクに相当します。この関数は 1 つ目のチャンクを生成するだけです。しかし i はゼロと等しいので、このチャンクは空のチャンクで、単純に閉じるだけで生成できます。

i がゼロと等しくなかった場合、この関数ははじめに入力チャンクを開きます。これは (start にある 1 つの) 最初の文字と start + 1 から最後までの characters チャンクを剥き出しにします。それからこの関数は、後者のチャンクを最初の i - 1 文字を含む部分と残りの count - i 文字を含む部分とに分割します。この後者のチャンクは返されるべき 2 番目のチャンクです。最後に、返されるべき 1 番目のチャンクを得るために、この関数は start にある character チャンクと、再帰呼び出しによって返された characters(start + 1, i - 1) チャンクをまとめます。

関数 getchars 中の *(start + i) への代入を検証するために、ここではこの補題関数を呼び出して、それからチャンク characters(start + i, count - i) を開く必要があります:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    for (int i = 0; i < count; i++)
        //@ invariant characters(start, count) &*& 0 <= i;
    {
        char c = getchar();
```

```
        //@ split_characters_chunk(start, i);
        //@ open characters(start + i, count - i);
        *(start + i) = c;
    }
}
```

これで VeriFast はこの代入を受け付けます。今度はループ本体の終了でのループ不変条件をチェックしようとして VeriFast はエラーになります。チャンク `characters(start, count)` が見つからないためにエラーとなるのです。これには再びわずかにシンボリックヒープを書き換えるだけであることに注意してください: `characters(start, count)` で表わされた全てのメモリパーミッションは実際にはシンボリックヒープ中に存在し、VeriFast が望む形になっていないだけです。VeriFast を満足させるために、はじめに `characters(start + i, count - i)` チャンクを再び閉じて、それから `characters(start, i)` と `characters(start + i, count - i)` チャンクを 1 つの `characters(start, count)` チャンクに戻す補題関数を呼び出す必要があります。

練習問題 28 補題関数 `merge_characters_chunks` を書いてください。それから `getchars` を検証できるように、この関数の呼び出しを `getchars` に挿入してください。さらに再帰の代わりにループを使うように `putchars` を書き換えて、結果得られたプログラムを検証してください。

6.27 再帰的なループの証明

前章での重要な観察結果は、関数 `getchars` の再帰バージョンの検証はループを使ったバージョンの検証よりはるかに簡単であるということです。実際、ループを検証するには、そのループによって使われるパーミッション (すなわちヒープチャンク) 全てを表現するようなループ不変条件を管理することが要求されます。対照的に、再帰関数の契約はその関数の特定の呼び出しにより使われるパーミッションのみを表現します; 再帰呼び出しの時点において、もし呼び出し元で使われるパーミッションのいくつかが呼び出される側に要求されない場合、これらのパーミッションは、再帰呼び出しの間、単に呼び出し元のシンボリックヒープに留まります。

これは実行時間のパフォーマンスと検証時間の利便性は二者択一であることを意味するのでしょうか？ 幸運にも、そうではありません! 最近 Thomas Tuerk という研究者が、相当する再帰関数を検証するのと同等の利便性でループを検証するアプローチを示しました。具体的には、Tuerk はどのようなループも等価な再帰関数に書き直せると指摘しています。次のようなループを考えてみましょう:

```
while (condition) {
    ... body ...
}
```

このループは次の呼び出しと等価です

```
iter();
```

このとき関数 `iter` は次のように定義されます

```
void iter() {
    if (condition) {
        ... body ...
        iter();
    }
}
```

具体例として、前章での関数 getchars を考えてみましょう。**for** ループを **while** ループで書き換えれば、次のコードが得られます:

```
void getchars(char *start, int count) {
    int i = 0;
    while (i < count) {
        char c = getchar();
        *(start + i) = c;
        i++;
    }
}
```

これで上記の変換を適用して、このループを等価なローカル再帰関数で置換できます:

```
void getchars(char *start, int count) {
    int i = 0;
    void iter() {
        if (i < count) {
            char c = getchar();
            *(start + i) = c;
            i++;
            iter();
        }
    }
    iter();
}
```

この変換ではローカル関数を使っていることに注意してください。C 言語標準はローカル関数をサポートしませんが、GNU C コンパイラ gcc はサポートしています。 gcc は上記の関数を正しくコンパイル/実行します。

それぞれのループが等価な再帰関数に書き換えできることを示した後、Tuerk はこれが再帰関数を検証するのと同じ方法でループを検証できることを意味することを示しました: その再帰関数に対して契約を提供するのです。

VeriFast はこのアプローチをサポートしています。ループを検証するとき、ループ不変条件を指定する代わりに、事前条件と事後条件から成るループ契約を指定できます。すると VeriFast は、あ

たかもそのループがローカル再帰関数を使って書かれているかのように、そのループを検証します。これがどのように働くか見るために、最初にローカル再帰関数 `iter` を使う `getchars` 関数のバージョンを検証してみましょう:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    int i = 0;
    void iter()
        //@ requires characters(start + i, count - i);
        //@ ensures characters(start + old_i, count - old_i);
    {
        if (i < count) {
            //@ open characters(start + i, count - i);
            char c = getchar();
            *(start + i) = c;
            i++;
            iter();
            //@ close characters(start + old_i, count - old_i);
        }
    }
    iter();
}
```

関数呼び出しが開始時点での変数の値を参照するために、`old_` を接頭辞につけた変数名を使っていることに注意してください。 VeriFast は現時点ではローカル関数をサポートしていませんが、もしサポートしていたらこの関数の検証は成功するでしょう。

これで **while** ループを使う `getchars` のバージョンを検証するために必要な残りは、再帰関数に挿入した注釈をループバージョンに移植することだけです:

```
void getchars(char *start, int count)
    //@ requires characters(start, count);
    //@ ensures characters(start, count);
{
    int i = 0;
    while (i < count)
        //@ requires characters(start + i, count - i);
        //@ ensures characters(start + old_i, count - old_i);
    {
        //@ open characters(start + i, count - i);
        char c = getchar();
        *(start + i) = c;
```

```
        i++;
        //@ recursive_call();
        //@ close characters(start + old_i, count - old_i);
    }
}
```

recursive_call() を挿入したことに注意してください; このゴースト文は仮想的な再帰呼び出しが起きる位置を示しています。VeriFast はこの関数の検証に成功します。

練習問題 29 while ループを使う putchars 関数のバージョンを書いて、それをループ契約を使って検証してください。

練習問題 30 ループ契約を使って 6.11 章での stack_get_count 関数を検証してください。この証明には述語 lseg やどのような補題も必要ではありません! けれども **while** ループを **for (;;)** ループに (もしくは同等な **while (true)** ループに) 書き換える必要があることに注意してください。ループから抜ける前にゴーストコマンドを実行する必要があるためです。

6.28 配列の内容物を追跡する

1 つの文字配列の内容物を別の配列にコピーする memcpy 関数の次の実装の関数的な正しさを検証したいと仮定してみましょう。

```
void memcpy(char *dest, char *src, int count) {
    for (int i = 0; i < count; i++) {
        dest[i] = src[i];
    }
}
```

関数的な正しさを明記するために、この関数が返るときに dest 配列の内容物は src 配列の内容物と等しいことを、memcpy を表わす契約は表現しなければなりません。これは、これまでの章での述語 characters は文字配列のサイズのみを明記していて、その内容物に言及していないので、この述語を使うことはできないことを意味しています。配列によって保持される文字のリストを明記する述語パラメータを追加すべきです:

```
predicate chars(char *array, int count; list<char> cs) =
    count == 0 ?
        cs == nil
    :
        character(array, ?c) &*& chars(array + 1, count - 1, ?cs0) &*&
        cs == cons(c, cs0);
```

この述語を使うことで、次のように memcpy 関数の挙動を完全に指示できます:

```
void memcpy(char *dest, char *src, int count)
    //@ requires chars(dest, count, _) &*& [?f]chars(src, count, ?cs);
    //@ ensures chars(dest, count, cs) &*& [f]chars(src, count, cs);
```

実際、この述語は VeriFast において “公式な” 文字配列の述語です: それは多数の便利な補題と一緒に prelude.h で宣言され、いくらかの目的に VeriFast で使われます。例えば、次の関数は memcpy に対して上記の契約を検証しています:

```
void test()
    //@ requires true;
    //@ ensures true;
{
    char buffer1[7] = "Hello!";
    char buffer2[7];
    memcpy(buffer2, buffer1, 7);
    assert(buffer2[5] == '!');
}
```

実際、ローカルに文字配列を宣言すると VeriFast は chars チャンクを生成します。

述語 chars の定義におけるセミコロンに注意してください: これは 2 つの入力パラメータ (array と count) と 1 つの出力パラメータ (cs) が正確に宣言されていることを意味しています。6.21 章と 6.22 章で解説した通り、これで VeriFast は断片 chars を結合し、ある環境下では chars チャンクに対して自動的に **open** と **close** を呼び出します。

VeriFast は、chars チャンクに対する表明の可読性を向上させるために、**配列スライス構文** (*array slice syntax*) をサポートしています。 a の型が char * であるとき、表記 a[i..n] |-> ?vs は chars(a + i, n - i, ?vs) と等価です。配列スライス構文を使うことで、次のようにより若干可読性の高い関数 memcpy の契約を書くことができます:

```
void memcpy(char *dest, char *src, int count)
    //@ requires dest[0..count] |-> _ &*& [?f]src[0..count] |-> ?cs;
    //@ ensures dest[0..count] |-> cs &*& [f]src[0..count] |-> cs;
```

これでこの実装を検証できます。配列を扱う際によくあることですが、((6.27 章で見たように) ループ不変条件の代わりにループ契約を使うのが効果的です。

```
void memcpy(char *dest, char *src, int count)
    //@ requires dest[0..count] |-> _ &*& [?f]src[0..count] |-> ?cs;
    //@ ensures dest[0..count] |-> cs &*& [f]src[0..count] |-> cs;
{
    for (int i = 0; ; i++)
        //@ requires dest[i..count] |-> _ &*& [f]src[i..count] |-> ?cs0;
        //@ ensures dest[old_i..count] |-> cs0 &*& [f]src[old_i..count] |-> cs0;
```

```
    {
        //@ open chars(dest + i, _, _);
        //@ open chars(src + i, _, _);
        if (i == count) {
            break;
        }
        dest[i] = src[i];
    }
}
```

ループ契約を使う際によくあることですが、ループを抜ける (すなわち **break** 文の) 前にゴーストコマンドを挿入するために、ループ条件をループ本体内に移動させなければならなかったことに注意してください。

練習問題 31 2 つの配列が同じ内容物を持つなら、関数 `memcmp` の次の実装がゼロを返すことを明記して検証してください。注意: 表明中のブール式が括弧ではじまる場合、そのブール式に接頭辞 `true ==` を付けなければならなりません、例えば `true == ((a == b) == (c == d))` です; これは VeriFast パーサの制限です。

```
int memcmp(char *p1, char *p2, int count) {
    int result = 0;
    for (int i = 0; ; i++) {
        if (i == count) {
            break;
        }
        if (p1[i] < p2[i]) {
            result = -1; break;
        }
        if (p1[i] > p2[i]) {
            result = 1; break;
        }
    }
    return result;
}
```

6.29 文字列

C 言語プログラムでは、慣習的に文字列は **ゼロ終端** (*zero-terminated*) された文字配列としてメモリに保存されます。例えば、文字列 `"Hello"` は`'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `0` のように保存されます。これは文字列の長さを算出する関数は次のように書けることを意味しています:

```
int strlen(char *s) {
    int i = 0;
    for (; s[i] != 0; i++);
```

```
    return i;
}
```

次はこの関数を使う単純な main プログラムです:

```
int main() {
    int n = strlen("Hello, world!");
    assert(n == 13);
    return 0;
}
```

この関数と main プログラムはどうすれば検証できるでしょうか？

関数 strlen の事前条件は、アドレス s から値がゼロにになるはじめの位置までの全ての位置へのアクセスを充足するパーミッションを、呼び出し元が供給することを明記すべきです。前章で導入した述語 chars を使ってこれに対処できますが、この目的のために新しい述語を定義するのがよりエレガントです:

```
predicate string(char *s; list<char> cs) =
    character(s, ?c) &*&
    c == 0 ?
        cs == nil
    :
        string(s + 1, ?cs0) &*& cs == cons(c, cs0);
```

この述語を使って、次のように関数 strlen を明記できます:

```
int strlen(char *s)
    //@ requires string(s, ?cs);
    //@ ensures string(s, cs) &*& result == length(cs);
```

この仕様が与えれると、この関数は簡単に検証できます。

実際には、述語 string は VeriFast でのゼロ終端された文字列を表わす “公式な” 述語です: それは多数の便利な補題と一緒に prelude.h で宣言され、文字列リテラル式が評価されると VeriFast は string チャンクを生成します。

ここで main プログラムを検証してみましょう。不幸にも、この検証は失敗します。その理由は、関数 strlen の上記の事前条件は文字列に対するフルパーミッション、すなわち文字列の文字を読み書きするパーミッション、を要求するからです。けれども、C 言語文字列リテラルは **不変** (*immutable*) です; プログラムはそれを変更してはなりません。(6.20 章で見たように) 文字列リテラルに対する **分割所有パーミッション** (*fractional permission*) のみを生成することで、VeriFast はこれを反映します。

けれども幸運にも、関数 strlen はそれが操作する文字列を変更しません。そのため、分割所有パーミッションのみを要求するように、その契約を弱めることができます:

```
int strlen(char *s)
    //@ requires [?f]string(s, ?cs);
    //@ ensures [f]string(s, cs) &*& result == length(cs);
```

練習問題 32 関数 strlen と main プログラムを検証してください。ループを検証するために、(6.27 章で見た) ループ契約を使ってください。ループ条件をループ本体に移動する必要があることに注意してください。

6.30 ポインタの配列

前章では文字の配列に対するサポートを VeriFast に導入しました。符号無し文字、整数、符号無し整数、そしてポインタ、の配列に対して同様のサポートがあります。この章では、クラス内の生徒の名前を追跡する単純なアプリケーションのメモリ安全性を検証するために、ポインタの配列に対する VeriFast サポートを使います。次のプログラムは生徒の名前のリストを読み、それからユーザに一定時間で k 番目の生徒の名前を検索することを許します。ユーザが無効な生徒の番号を入力したら、このプログラムは全ての確保したメモリを解放してから終了します。

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int read_int() {
    int x;
    int scanf_result = scanf("%i", &x);
    if (scanf_result < 1) abort();
    return x;
}

char *read_line() {
    char buffer[100];
    int scanf_result = scanf(" %99[^\n]", buffer);
    if (scanf_result < 1) abort();
    char *result = strdup(buffer);
    if (result == 0) abort();
    return result;
}

int main() {
    printf("How many students do you have? ");
    int n = read_int();
    if (n < 0 || 0x20000000 <= n) abort();
    char **names = malloc(n * sizeof(char **));
    if (names == 0) abort();
```

```
    for (int i = 0; i != n; i++) {
        printf("Please enter the name of student number %d: ", i + 1);
        char *name = read_line();
        printf("Adding ' %s' ...\n", name);
        names[i] = name;
    }

    for (;;) {
        printf("Please enter a student number: ");
        int k = read_int();
        if (k < 1 || n < k) {
            printf("Student number out of range. Terminating...\n");
            break;
        } else {
            char *name = names[k - 1];
            printf("Student number %d is called %s.\n", k, name);
        }
    }

    for (int i = 0; i != n; i++) {
        free(names[i]);
    }
    free(names);

    return 0;
}
```

ヘルパー関数 `read_int` は、標準入力から (10 進数表記もしくは 16 進数表記の) 整数値を読み、それを変数 `x` に保存するために、(ヘッダファイル `stdio.h` で宣言された) 標準 C 言語関数 `scanf` を使います。同様にヘルパー関数 `read_line` は、改行文字ではない最大 99 文字の列、すなわち最大 99 文字の 1 行を読み、それを (スタックに確保された) 文字配列 `buffer` に保存するために、`scanf` を使います。 `scanf` の返り値は読み出しに成功した要素の数を示します。それから、`read_line` は (ヘッダファイル `string.h` で宣言された POSIX 標準の) ライブラリ関数 `strdup` を使って、`buffer` 中のゼロ終端された文字列を新たにヒープに確保したメモリブロックにコピーします。メモリ不足のためにこの操作に失敗した場合、`strdup` は 0 を返します。

関数 `read_int` の検証は自明です。関数 `read_line` の検証には困難が 1 つあります: `scanf` 呼び出しの後、配列 `buffer` は `chars` チャンクで表わされますが、関数 `strdup` は `string` チャンクを要求します。幸運にも、`prelude.h` で宣言された補題 `chars_separate_string` を使って、ゼロ文字を含むどのような文字配列も `string` チャンクに展開できます。関数 `read_line` が返るとその配列は `chars` チャンクとして再び有効になるべきなので、`string` チャンクを `chars` チャンクに結合するために `strdup` 呼び出しの後で補題 `chars_unseparate_string` を呼び出すべきです。

この main 関数では、検証は 3 つのループそれぞれに注釈を要求します。最初のループに注目しましょう。このループは `names` 配列を前から後へ辿っているので、ループ契約を使います。このループはその配列へのアクセスを要求します; どうやればこれを明記できるでしょうか？ どうやっ

て VeriFast は malloc 文で確保された配列を表わすのでしょうか？ 実際には、もし malloc 呼び出しが T **x = malloc(n * sizeof(T *)) の形なら、VeriFast は pointers(x, n, _) チャンクを生成します。述語 pointers は prelude.h で定義され、(6.28 章で見た) 述語 chars に似ています:

```
predicate pointers(void **pp, int count; list<void *> ps) =
    count == 0 ?
        ps == nil
    :
        pointer(pp, ?p) &*& pointers(pp + 1, count - 1, ?ps0) &*&
        ps == cons(p, ps0);
```

chars チャンクと同様に、VeriFast は pointers チャンクに対する配列スライス構文をサポートしています: a が型 T ** であるとき、表記 a[i..n] |-> ?ps は pointers(a + i, n - i, ?ps) と等価です。したがって、関数 main 中の最初のループに対する適切な事前条件は次のようになります:

```
//@ requires names[i..n] |-> _;
```

このループは、ゼロ終端された文字列を含むヒープに確保されたメモリブロックへのポインタを、配列 names のそれぞれの要素に保管します。したがって、このループの事後条件はその配列自身だけでなくこれらのメモリブロックも同時に明記すべきです。6.17 章で議論した通り、述語 foreach を使ってリストのそれぞれの要素を表わすチャンクの存在を明記できます。したがって、このループに対する妥当な事後条件は次のようになるでしょう:

```
//@ ensures names[old_i..n] |-> ?ps &*& foreach(ps, student);
```

このとき述語 student は次のように定義されます:

```
predicate student(char *name) =
    string(name, ?cs) &*& malloc_block_chars(name, length(cs) + 1);
```

(注意: n 個の文字の配列 a の確保は malloc_block_chars(a, n) チャンクを生成します; 同様に、n 個のポインタの配列 a の確保は malloc_block_pointers(a, n) チャンクを生成します。) 上記の事後条件は動作しますが、(6.22 章で見たように) 正確なバリアントを使うことで、これらのチャンクと動作するのに必要な **open** と **close** 文の数を減らすことができます。次のように正確な述語 student を宣言できます:

```
predicate student(char *name;) =
    string(name, ?cs) &*& malloc_block_chars(name, length(cs) + 1);
```

VeriFast は **ゴーストヘッダファイル** (*ghost header file*) を使います (ゴーストインクルード命令//@ #include <listex.gh> を使ってインクルードされます)。これは foreachp と名付けられた述語 foreach の正確なバリアントを宣言しています:

```
predicate foreachp<t>(list<t> xs, predicate(t;) p;) =
    xs == nil ?
```

```
        emp
    :
        xs == cons(head(xs), tail(xs)) &*& p(head(xs)) &*&
              foreachp(tail(xs), p);
```

これらの正確な述語を使うことで、最初のループに対して次の事後条件が得られます:

```
//@ ensures names[old_i..n] |-> ?ps &*& foreachp(ps, student);
```

これで最初のループの本体は、ほとんどどのような注釈もなしに検証できます。VeriFast は `i == n` なら `ps == nil` であることを理解しません。このため、このループから抜けるパスの検証に失敗します。ループ条件をループ本体に移動し、本体の先頭に **open** 文を挿入することで、場合分けを強制する必要があります:

```
for (int i = 0; ; i++)
    //@ requires names[i..n] |-> _;
    //@ ensures names[old_i..n] |-> ?ps &*& foreachp(ps, student);
{
    //@ open pointers(_, _, _);
    if (i == n) {
        break;
    }
    printf("Please enter the name of student number %d: ", i + 1);
    char *name = read_line();
    printf("Adding '%s' ...\n", name);
    names[i] = name;
}
```

これで最初のループは検証できます。

2 番目のループは、そのループを直線的に辿らない点で、最初のループと異なります; どちらかといえばランダムなアクセスを行ないます。したがって、通常のループ不変条件を使います:

```
//@ invariant names[0..n] |-> ?ps &*& foreachp(ps, student);
```

最初のループでは、配列要素アクセス `names[i]` は `pointers(names + i, n - i, _)` チャンクの自動 open を引き起こし、(6.13 章で見たように) その要素アクセスで要求された `pointer(names + i, _)` チャンクをむき出しにしました。 2 番目のループでは、要求された `pointer` チャンクは `pointers` チャンクの前方ではなく中にあるので、`names[k - 1]` アクセスはこの方法では検証できません; それをむき出しにするには `k` 回の **open** 操作が必要で、自動 open 機能はこれを扱えません。けれども、VeriFast はこのアクセスをランダムアクセスとして認識して、それを特別に扱うのでこのアクセスの検証は成功します。特に、もし `a[i]` の形の配列アクセスを評価するとき VeriFast が `0 <= i < n` であるようなチャンク `pointers(a, n, ps)` を見つけると、そのアクセスを有効だと見なして、そのアクセスの結果として `nth(i, ps)` を返します。不動点関数 `nth` はヘッダファイル

list.h で宣言されています; nth(i, ps) はリスト ps の i 番目の要素を返します。

これは配列アクセス自身を解決しますが、別の問題があります: printf 呼び出しは要素 k - 1 によって指された文字列を表わす string チャンクを要求します。このチャンクは foreachp チャンクの内側にあります。ゴーストヘッダ listex.gh で宣言された補題 foreachp_remove_nth を使って、これを展開できます:

```
lemma void foreachp_remove_nth<t>(int n);
    requires foreachp<t>(?xs, ?p) &*& 0 <= n &*& n < length(xs);
    ensures foreachp<t>(remove_nth(n, xs), p) &*& p(nth(n, xs));
```

これは list.h で宣言された不動点関数 remove_nth を使います。

要素 k - 1 に対する student チャンクが有効になると、そのチャンクは自動 open され、printf 呼び出しは検証できます。 printf 呼び出しの後、補題 foreachp_unremove_nth を使って、student チャンクを foreachp チャンクに結合する必要があります:

```
lemma void foreachp_unremove_nth<t>(list<t> xs, int n);
    requires
        foreachp<t>(remove_nth(n, xs), ?p) &*& 0 <= n &*& n < length(xs) &*&
        p(nth(n, xs));
    ensures foreachp<t>(xs, p);
```

これら 2 つの補題呼び出しは、2 番目のループ本体を検証するために要求される唯一の注釈です。

3 番目のループは再び前から後ろへそのループを辿ります; ループの仕様は示されています。ループは生徒の名前を保持するメモリブロックを解放するので、foreachp チャンクは事後条件から消えます:

```
//@ requires names[i..n] |-> ?ps &*& foreachp(ps, student);
//@ ensures names[old_i..n] |-> _;
```

ループ本体の検証には少しの注釈が必要です。 1 番目として、VeriFast は i == n が ps == nil を意味すると解釈しないので、ループから抜けるパスでリークエラーになります。最初のループと同様に、ループ条件をループ本体に移動し、pointers チャンクの明示的な **open** を挿入する必要があります。

2 番目の問題は free 呼び出しが満されないことです: a の型が char * であるような free(a) 呼び出しは malloc_block_chars(a, n) チャンクと chars(a, n, _) チャンクを探します。その malloc_block_chars チャンクは foreachp チャンクの内側にありますが、chars チャンクはありません; そのメモリブロックは代わりに string チャンクによって表わされています。したがって、string チャンクを chars チャンクに変換する必要があります。ヘッダ prelude.h の補題 string_to_chars がこの目的に有効です。この補題の呼び出しを free 呼び出しの手前に挿入しましょう。

3 番目の問題は string_to_chars 呼び出しが満されないことです。それが探す string チャンクは foreachp チャンクを開いた後に得られた student チャンクを開くことで得られます。しかし student チャンクは foreachp チャンクの内側に **静的に** (*statically*) 存在しないので、自動 open 機能はこれを見つけません; 正確には、2 番目の引数が述語 student の名前であったときのみ、そのチャンクは foreachp チャンクの内側に存在します。将来、自動 open 機能はこのようなシナリオを

サポートするかもしれませんが、現時点では foreachp チャンクを明示的に開く必要があります。3 番目のループの最終的な証明は次のようになります:

```
for (int i = 0; ; i++)
    //@ requires names[i..n] |-> ?ps &*& foreachp(ps, student);
    //@ ensures names[old_i..n] |-> _;
{
    //@ open pointers(_, _, _);
    if (i == n) {
        break;
    }
    //@ open foreachp(_, _);
    //@ string_to_chars(names[i]);
    free(names[i]);
}
```

これでこのプログラムは検証できます。

練習問題 33 (VeriFast 配付版の tutorial ディレクトリの students.c ファイルである) このプログラムを検証してください。

6.31 参考文献

- VeriFast 公式サイト[*9]
- VeriFast の GitHub リポジトリ[*10]
- VeriFast メーリングリスト[*11]
- “The VeriFast Program Verifier: A Tutorial” [*12]
- “プログラム検証器 VeriFast: チュートリアル” [*13]
- “Verification of usbkbd” [*14]
- “TPPMark2016 を解きながら学ぶ VeriFast” [*15]
- “VeriFast Termination Checking Introduction(α)” [*16]

[*9] https://people.cs.kuleuven.be/~bart.jacobs/verifast/
[*10] https://github.com/verifast/verifast
[*11] https://groups.google.com/forum/#!forum/verifast
[*12] https://people.cs.kuleuven.be/~bart.jacobs/verifast/tutorial.pdf
[*13] https://github.com/jverifast-ug/translate/blob/master/Manual/Tutorial/Tutorial.md
[*14] https://people.cs.kuleuven.be/~willem.penninckx/usbkbd/
[*15] https://speakerdeck.com/eldesh/tppmark2016-wojie-kinagaraxue-bu-verifast
[*16] https://speakerdeck.com/eldesh/verifast-termination-checking-introduction-a

ぬしおさんを偲ぶ会議事録

— 有志

まえおき

本誌の原稿締切が近づいたある日、参照透明な海を守る会の創立メンバーであり@nushio 名義で以前のλカ娘 (1,2,4,5,7,8 の各巻) にも記事を書いている「ぬしお」さんこと村主崇行さんの訃報が飛び込んで来ました。「[tennet:15311] 訃報：村主崇行氏 (理研計算科学研究機構) [*17]」と簡潔に発表されたそのニュースを見て我々は呆然とするしかありませんでした。将来を嘱望された科学者の、33 歳でのあまりに早すぎる死でした。Twitter のタイムラインで村主さんの訃報が色々な人にツイートされているのを眺めながら、これが現実に起こったことなのだとようやく実感しつつあるところです。

村主さんは宇宙物理学にとって、また計算機科学にとってかけがえのない人物であったことはもちろんですが、当サークルのわたしたちにとっても本当にかけがえのないメンバーでした。村主さんの事を鮮明に思い出せるうちに（どんな記憶もやがて薄れて消えてしまいますから)、わたしたちから見た村主さんについて記録しておこうと思います。

村主さんと関わった人たちで Skype ルームでチャットを行い、それを整理しました。整理、とは言うものの、気持ちの整理がつかない状態で思い思いに語った事ですので結局雑然としたものになってしまいました。非凡な村主さんに贈る手向けの言葉としては甚だつたないものですが、これがいまのわたしたちに用意できる精一杯のものです。甘えすぎかもしれませんが、わたしたちにいつも優しく接してくれた村主さんならきっと快く受け取ってくれるのではないかとも思うのです。

村主さん、あなたはわたしたちの、最高の友達でした。

偲ぶ会議事録

`@master_q` えーと、とりあえず立ててみました。皆で生前のぬしおさんの思い出を語って、サークルとしてのご供養とできなかなぁと考えています。

`@master_q` というか、たったいま思い出したけどλカ娘の原稿の Git サーバとサークルの Web ページ [*18] はぬしおさんのポケットマネーで運用されていたんですよね。長い間おせわになりました。。。

`@master_q` はじめて僕とぬしおさんが出会ったのはスタート Haskell [*19] というイベントの第何回目かで「すごい Haskell たのしく学ぼう！」本 [*20] にサインしてもらったときでした。

`@dif_engine` 僕はリアルでお会いしたのはぬしおさんの結婚式が初めてだったんだよな.

[*17] `http://www.asj.or.jp/tennet/archives/msg07622.html`
[*18] `http://www.paraiso-lang.org/ikmsm/`
[*19] `https://atnd.org/events/17468`
[*20] `https://estore.ohmsha.co.jp/titles/978427406885P`

@master_q そういえば『@master_q 君、写真選ぶのうまいから、自営業として結婚式写真スライド作るのやってよー』とお願いされたことがあるのですが、責任重大すぎて遠慮しました。。。

@dif_engine Haskell 関係で Twitter でフォローしていて，簡約!? λカ娘 (二期) [*21] のときに表紙募集していたあたりで手を上げたのがこのサークルに関わるきっかけだったな.

@dif_engine もちろん@tanakh さんもフォローしていたから，どちらのツイートをみて応募したか，までは思い出せない…

@dif_engine 二期の表紙のときは個人的には博士論文で忙しくしていて，現実逃避で絵を描いた面もある.

@master_q あああ。。。そうそう。FPGA 合宿なるものを xhl サン/ぬしおサン/僕の 3 人で開催したのに誰も FPGA ボード持ってこなくて Idris [*22] が吐く実行バイナリのパフォーマンス解析する合宿になってしまったこともありましたね。。。 たしか@tanakh さんが合宿にはリモートで参加してくれていて、みるみるうちにパフォーマンスのボトルネックが解析された記憶が。

@dif_engine そのあと「すご H」翻訳のお手伝いで鹿野さんの出版システムのチャットで色々議論する機会があった．僕にとっては村主さんと田中さんはいつもペアだったんだよな.

@dif_engine そのあと簡約!? λカ娘 (算) [*23] の表紙も続けてやりたいですみたいなことを言ったら，村主さんが『どうせなら記事も書きませんか』とか言ってくれた.

@master_q ぬしおさんには『今 Idris のパフォーマンス向上に貢献したヒーローになれるよ！』とアドバイスされたのだけれど辞退して、ATS [*24] プログラマになって、そして ATS プログラマを脱落しつつあります。。。あの日ぬしおさんのアドバイスを聞いていたら僕にはどんな未来があったのだろうか。。。

@master_q 『すご H』のチャットは貴重な経験ですね。「すご H」本はみんながよくわからなかった Haskell 言語について明確に説明した日本語文献として今後も輝き続けるのではないでしょうか。

@dif_engine このサークルの起源としてはやはりきつねさんのイカ娘ツイートが発端ですよね. 僕の知る限り.

@master_q そうですね。触手で LISP を書くんでしたっけ？ なのに蓋をあけてみたら Haskell 勢が書く同人誌が！

@dif_engine もともと原作でイカ娘は数学が得意で，TV ではキーボードをカチャカチャ高速でやってるシーンがあって… それできつねさん (xhl) がそれに触発されて連ツイした

@master_q たしか TV の 1 コマに写っている本の背表紙が『これは絶対ドラゴン本だ』とか盛

[*21] http://www.paraiso-lang.org/ikmsm/books/c81.html
[*22] https://www.idris-lang.org/
[*23] http://www.paraiso-lang.org/ikmsm/books/c82.html
[*24] http://www.ats-lang.org/

り上がった TL もあった気がしました[*25]。

@dif_engine ソレ言い出したのもきつねさんでしたよねたしか

@dif_engine 第一巻のメンバーってきつねさん，ぬしおさん，田中さん，そして岡部さんなんだけど，最初三人はプログラミングコンテスト勢？

@dif_engine 気分を害されたら申し訳ないが，岡部さんだけ一巻のメンバーでやや毛色が違う気がする．

@master_q そうですね。3 人はたぶん交友関係があったと思うのですが、僕は勝手に Twitter 経由で飛び込んできた感じです。

@dif_engine なるほどなるほど．そんな感じもしてましたがいままでハッキリしてなかったので．(僕の中では)

@master_q 当時の僕の文章の稚拙なことといったら、、、このサークルでだいぶ自然言語をトレーニングしてもらいました。ありがたや。

@ark_golgo 私は確か簡約!? λカ娘　Go! [*26] からまぜてもらったんだよなあ…。git 操作間違えたのを村主さんに直して頂いた。

@ark_golgo あと、村主さんが『原稿には囲碁のルールも入れた方がいい』と主張されてて、結果的にそれのお陰で良い原稿になった気がする

@ark_golgo 学振、論文についてもアドバイス頂いたなあ

@ark_golgo 直接お会いしたのは小籠包を一緒に食べたときだけなのかな…？ その時岡部さんとも初めて出会った

@ark_golgo ICPC 関連でお会いしてた可能性もあるけどどうも思い出せない…。

@ark_golgo まあマジで凄い人でした

@ark_golgo あー肝心なことを忘れてた

@ark_golgo 「嫌われる勇気」って本を勧めてくれたのも村主さんだ

@ark_golgo あの本読んでなかったら多分もっと精神病んでたな…。そーいえば、英語論文の書き方の本 [*27] を紹介してくれたのもぬしおさんだったなぁ

@ark_golgo 英語力凄かったらしいですからねえ

@ark_golgo 岡野原さん [*28] っていう、私が今まで見たなかでもトップクラスの化け物の人が、村主さんの訃報に触れて「才能の塊」って表現してた

@golden_lucky すごい Haskell のときは、制作中は村主さんは京都にいらしたので、お互いに顔を見ないままでの作業だったんですよね。それでも、訳文やメールでのやりとりなど

[*25] xhl メモ @ hatena: たけるの通う小学校に Dragon Book や SystemC 等の専門書籍がてんこもりでヤバイでゲソ! http://d.hatena.ne.jp/xiaohuli3/20111201

[*26] http://www.paraiso-lang.org/ikmsm/books/c84.html

[*27] http://www.springer.com/in/book/9781441979223

[*28] https://twitter.com/hillbig

から人物像はかなりはっきりイメージできていて、刊行後にはじめて直接お会いしたときにもそのイメージのままだったので、すごいブレがない人だーと思ったのでした。

@golden_lucky 翻訳中、LaTeX 原稿を svn サーバ (当時) に commit するとビルドサーバ上で PDF を自動生成するようになってたんですが、LaTeX のエラーが起きるとエラーメッセージがメーリングリストに飛ぶようになってたんです。で、ある朝会社についてメールを開けると、かなりの数のエラーメッセージがメールで飛んできていて、全部むらぬしさんの commit によって引き起こされたものでした。commit メッセージによると、深夜に自分の commit でエラーになったからといって、作業中のファイルを二分探索してエラー箇所を特定しようとしてたみたいなんですね。そんな、(いい意味で異常な) 責任感のある著者は、はじめてでした (いまだにいない)。

@golden_lucky あとはやっぱり、自分の退職時なんですが、いろんな方に退職の挨拶を出して、それに『いつか本を書くからそのときはよろしく』という返答をはっきりとくれて、その後にも連絡をくれた数少ない著者の一人なんですよね。。村主さんから Lipovaca 先生に続編の相談とかもしてたり (レスポンスはこなかったけど)、そのへんはやっぱりとても心残りです。

@master_q 貴重な体験談ありがとうございました。 そしてぬしおさんは京都大学から理研 AICS [*29] に移り、本格的に Haskell DSL を使ったスーパーコンピューティングへの道に進んだんですよね。

@ark_golgo 村主さん失ったせいで日本の科学は何年か遅れたんじゃないだろうか。

@dif_engine 流れはわすれたけど参照透明な海を守る会のチャットルームで村主さん本人に，どんな印象かと聞かれて，村主さんのページ [*30] は以前見ていたので，「アイデアを形にすることにこだわりのある方と思います」と言ったら，いつもそれを意識しているのでそのような感想は嬉しいとの返事を頂いたことがある．

@ark_golgo なるほどです

@master_q ポスト京コンピュータプロジェクトは明確に打撃を受けたでしょうね、、、ぬしおさん不在によって、、、

@ark_golgo 村主さんに「人間は将来 SF の世界をどこまで実現できるか」教えてもらえば良かったなあ

@dif_engine あとさっきの村主さんのページ見ていて思うのが，たとえば”王の智慧をあなたの掌に Riemann / Riemann 球面に複素関数をプロットし、紙風船のような展開図にあらわします。印刷して切り抜いて組み立てればおいらの夢もあなたのもの。”という辺りの言語感覚の独特さ．

@dif_engine 『王の智慧』っていったいどこからひねり出してきたの？ という不思議さがある．

@golden_lucky その言語感覚の独特さ、ありますね

[*29] http://www.aics.riken.jp/

[*30] http://www.geocities.jp/takascience/index_ja.html

@golden_lucky すごい Haskell ではそれをスポイルしないように編集したつもりですが、アマゾンのレビューで「後半の翻訳がはしゃぎすぎ」みたいながあって、ぐぬぬって憤りが。

@master_q そうして僕が Ajhc [*31] という組み込み向けコンパイラを作るプロジェクトをはじめたら、ぬしおさんが「論文を書かなきゃダメだよ。論文を出していないプロジェクトは研究者から相手にしてもらえないんだ」とアドバイスを受けて論文を書く決心をしました。ここから僕の草の根研究者としてのキャリアがスタートしました。

@master_q ぬしおさんとスタート Haskell を立ち上げたトラビスさん [*32] のサポートのおかげで無事論文は受理 [*33] されて、僕は Haskell シンポジウム [*34] で発表することができました。なにもかもぬしおさんのおかげです。しかもぬしおさんはこの 2014 年の Haskell シンポジウムに 2 本も論文出してたんだよなぁ。スゴイ。

@master_q Haskell シンポジウム併設の ICFP の学会期間中、ぬしおさんは「岡部さんがこんなに色々なプラットフォームで Haskell 動かしているんだったらデモブースを占有して持つべきだ。主催者に相談するよ」といってくれて、なぜか Ajhc Haskell コンパイラと ATS が動くマイコンボードと、Ajhc Haskell コード入り NetBSD が音楽を流す謎のブースが ICFP 期間中にオープンしました。様々な人に研究開発の成果を紹介できる機会得られました。ありがたや。

@master_q そしてなんとスウェーデンで開催される Haskell シンポジウムに共著論文を出したよしみで、僕は RIKEN AICS の研究嘱託という謎の身分を得ることができ、そして牧野先生 [*35] にこの 8 月に会う予定になっていて、その時にぬしおさんにもろもろのお礼を言おうと思っていた矢先に、、、このような事態になってしまい。ぽっかりと心に穴が抜けています。。。

@xhl_kogitsune ぬしお先生とは少なくとも 2009 年には遭遇している

@xhl_kogitsune 2011 年夏がﾚカ娘　1 巻。

@xhl_kogitsune xhl 同年の ICFPc でもご一緒しました [*36]

@xhl_kogitsune サークルの最初の頃については私もあいまいですねー。私が関数型イカ娘で本作るぞ、と意味分からんこと言って募集したらなぜかあの四人が集まった、という説明も何もできないような事態だったし。

@master_q 強いフォースに引き寄せられたんだぜぇ。

@xhl_kogitsune サークル成立の前史については 1 巻 0 章に詳しい。

@master_q これは買わないとですね！ (1 巻入手できるのか)

@xhl_kogitsune 具体的に何にどのように引き寄せられたんですか……?

@master_q いやぁ単純に「頭おかしい」「おもしろそう」「圧倒的成長できそう」でしょうか。

[*31] http://ajhc.metasepi.org/
[*32] https://github.com/TravisCardwell
[*33] http://dl.acm.org/citation.cfm?id=2633370
[*34] https://www.haskell.org/haskell-symposium/2014/
[*35] https://twitter.com/jun_makino
[*36] https://github.com/tanakh/ICFP2011

結果全てが成立しました。

@xhl_kogitsune わかる <頭おかしい

@master_q 僕は 2011 年夏といえば、某コピー機メーカから転職を考えていた期間ですね。その時に『あ、これおもしろいやんけ！』とネタがあったからすぐにとびついたんでしょうね。結果ぬしおさんとも遭遇できた。お得すぎる。

@ark_golgo 私もそんな感じで引き寄せられましたね。5 巻から参加。当時『λ探索』を調べていたのがきっかけになったのもある。

@master_q 頭おかしい」「おもしろい」「すごい」という 3 点においてはぬしおさんは抜群の人材でしたね。。。

@xhl_kogitsune うむー

@xhl_kogitsune そして他の人のことをめっちゃ気にかけてくれてた

@ark_golgo ですねえ。

@master_q そうそう、一度だけぬしおさんぼくの家に泊まったんですよ。リビングに 2 つ布団をしいて、1 つはぬしおさん。もう 1 つは僕で隣あわせで寝ることになったんです。そうしたら夜中僕のいびきがひどかったらしく、ぬしおさんは家の中にあったリラックスチェアを玄関まで持っていって、その椅子の上で一夜をあかしたようです。

@master_q 朝になって玄関で寝ているぬしおさんを発見したので、深く謝罪したのですが、ぬしおさんは「いやぁ面白かったからぜんぜんいいよー」とにこにこしておっしゃっていました。

@ark_golgo 聖人だ

@xhl_kogitsune ぬしおさんは、λカ娘では最初になぜか釣れた人の一人で、魔法少女まどか☆マギカ[*37]のイラストを挿絵としてたくさん描いてました。0 章冒頭に QB がいるのも確か彼の~~せい~~おかげで、まどか☆マギカが全 12 話なのでλカ娘も 12 巻まで出る、みたいな計画が立ったのも彼の~~せい~~おかげだと思っています。

@xhl_kogitsune そういう意味では、その計画の完遂や 10 話ほむほむ回の前にいなくなってしまうとは……とも思いました。

@master_q 自分に絵心がないのでぬしおさんの記事がはなやかなのがうらやましかったなぁ。

@xhl_kogitsune 同じく。絵心いいなぁと思った。

@xhl_kogitsune ぬしおさんは私が元気がなかった時に「食うかい?」ってお菓子くれたのすごく覚えてる

@master_q xhl さんとぬしおさんと僕の三人で FPGA 温泉合宿あった思い出があるんですが、あれどういう経緯で開催されたんでしたっけ？ 覚えてますかー？

@xhl_kogitsune いえ……どうせ私が温泉行きたいとか言い出したんじゃないですかね? (てきとー

[*37] http://www.madoka-magica.com/

@master_q なんとー。いまや 3 人とも簡単には会えなくなってしまいましたね。。。

@master_q ぬしおさん org-mode [*38] 使いだったのかー [*39]

@xhl_kogitsune nushio 先生とはコンパイラつながりで分野が近かったので、コード生成とか最適化とかの話をしていた印象が強い

@dif_engine 以前『ぬしおさんの筑波大集中講義スライドを元ネタに Haskell 入門かきたさ is ある』と投げたら『うーんたしかにあの筑波大講義もあのままオープんで放置しておくのはいかにも勿体無いんだよなあ・・・共著で書きますか（何』と返事が来て，そのうちやりたいなんて思ってたんだけど．少なくとも一緒にやる，という機会は失われてしまったな

@master_q ぬしおさんへの思い、そろそろ語りつくした感じでしょうかー。

@ark_golgo すみませんあと一言忘れてました

@ark_golgo ぬしおさん、囲碁も少しやってみたことがあるらしく、囲碁オフやるべきだったなと思いました。まる。

@tanakh せっかくなんでなんか書いておきたい気もするのですが、なかなか何を書くべきか難しいものでしてねえ

@tanakh たぶんこの中じゃ僕が一番長く付き合いがあって、今回の情況に関してもおそらく一番把握しているとは思うんですけれども、だからこそ何を書いて何を書かざるべきかというのが本当に難しくてつらい

@master_q ぬしおさんと tanakh さんは本当に「心の友」というか兄弟のように僕には見えていました。。。

@master_q 僕には「親友」と呼べるような人がいないので本当にうらやましかった。

@tanakh ご存じかも知れないですけど、僕はつらかった時期に彼に親切にされて救われたようなところがありまして、なんというか、逆のような立場があったならば、それは必ず借りを返さなければならないという思いもあったわけですが

@tanakh それが結局なされることが叶わなくなったのが、酷く残念でして。

@tanakh Formula [*40] を PEZY-SC [*41] に実装してゴードンベル賞をという話が来年あるはずで、今年はとりあえず TaihuLight で性能出す実装を作ってみるって言うのを、7 月にやる予定だったんですけど、

@tanakh んで 7 月はそのために僕がかり出される予定だったんですけど、なんかそれも直近の予定としてはなくなって、ただ Formula 自体は僕も後を引き継いで、彼の遺志を継いでいくというコトになりそうではあります。

@tanakh 後は、僕が、そうですねえ…ユニークな情報を提供できるとしたら、高校時代の話

[*38] http://orgmode.org/
[*39] https://twitter.com/nushio/status/816300850502995968
[*40] https://github.com/nushio3/formura
[*41] http://pezy.co.jp/products/pezy-sc.html

でしょうかね。僕が彼と出会ったのは僕が高校 2 年生の頃、彼は 1 年後輩でしたから、そこで高校の部活で一緒になったのが始まりですね。その頃から彼はあからさまに変わっていて、ノートにびっしりと変な図を書き込みながらクリックアンドプレイというゲーム作成ツールでひたすら変なゲームを作っていたのだけど、僕はそんなんじゃなくてもっと本格的なプログラミングをしなよと言ったりしても、相変わらず楽しそうに変なゲーム作ってて、まあとにかく変なヤツだなあと思っていたのですが、

@tanakh そのイメージがちょっと変わったのが翌年の夏で、僕は SuperCon というプログラミングコンテストに出るために東京に行っていたのですが、その宿泊先のホテルに突然彼からフロッピーディスクが届いて、何だろうって思ったら、SuperCon の課題を解いてみたっていうんですよね。動かしてみると確かにうにょうにょ奇妙な動きをしながらそれっぽい答えを出すプログラムができてて、いや、なんでわざわざフロッピー送ってくるの？ とか、これ VB で書いてあるけど、C 言語じゃないとスパコンで動かないんだけど、とか、そもそもどうやって宿泊先のホテル分かったの？ とか、いろいろ突っ込みどころはありましたが、この件で、ただのやばいヤツではなくて、実力もあるんだなというのが分かったのでした。

@tanakh おそらくこれがなければ大学に入ってから ICPC に誘うこともなかっただろうし、そうすればたぶん以降のつきあいも会ったか分からないので、彼を Haskell の道に引きずり込むこともなかっただろうし、もっとまっとうな（？）物理学の道を邁進できたのかなと思うこともあります。僕が変な道に引きずりこんでしまったようなものなので、研究が行き詰まっているなどと聞く度に、その責任の一端が自分にあるのではないかと感じたりすることもありました、本人は最後まで僕のことを師と呼んでくれていたし、研究者として後悔がなければよかったのですが。

@tanakh という感じでしょうか。白眉以降の彼の活躍は周知の通りだろうし、僕よりも語れる人がいそうですしね。なんかすごいとりとめの無い感じになってしまって、メチャクチャなかんじですが、これ上手くまとまりますかね…？

@master_q まとまります。大丈夫です。

@master_q そろそろオンラインでの思い出を語る会を終わろうと思います。皆様ご協力ありがとうございました。

会員名簿じゃないカ?

@myuon_myon (1 章)

圏論を使わない String Diagram をやろうとしたけれどあまり言えることがないというどうしようもない結論になりました. あと TikZ で図を描くの疲れました.

@public_ai000ya (2 章)

わかるぞ…Haskell が…みんなの力を。みんなの力が Haskell に…！！

@master_q (3, 5, 6 章)

jhc から ATS2 に乗り換えたら、第三の選択肢 VeriFast に出会ってしまいどの手法を採用すべきか悩みなのか嬉しいのかわからない状況に落ちています... 明日はどっちなんだぜー

@dif_engine (4 章)

猫の玩具にと思ってハンドスピナーを買ったのですがまったく遊んでくれませんでした.

@eldesh (6 章)

関数型言語の勉強をしていたはずなのにいつの間にか VeriFast を勉強していた件。

「簡約(かんやく)！？ λ(らむだ) カ娘(か むすめ) 10」

2017 年 8 月 11 日 コミックマーケット 92 初版発行

原作：安部真弘「侵略！ イカ娘」より

発行：参照透明な海を守る会

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

ikmsm-authors@googlegroups.com

著者：@dif_engine, @master_q, @myuon_myon, @public_ai000ya, @eldesh

表紙：@dif_engine

印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/