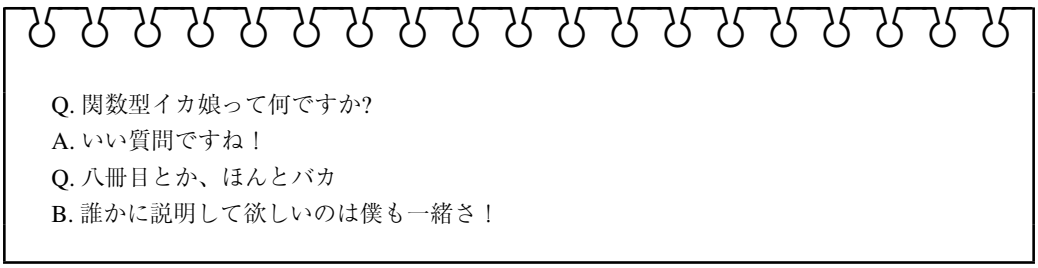
簡約!? 入力娘 8

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. 八冊目とか、ほんとバカ
B. 誰かに説明して欲しいのは僕も一緒さ！

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。

この本は八冊目の関数型イカ娘の本でゲソ。シリーズも後半に突入したので、少し雰囲気を変えていくでゲソ! アニメ 3 期の放映が近いことを祈りつつ、関数型言語で地上を侵略しなイカ!

この本の構成について

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

第1章

エルエフ島漂流記

— @master_q

1.1 不時着

今日は何時でここは何処だろう。とにかくこの小さな島は僕が住んでいた島と陸続きではないようだ。僕等はまだ見ぬ新しい技術を手に入れるために祖国をはなれて船出したのだ。しかしあのひどい嵐に飲まれてこの島に不時着してしまった。計画にない出来事に僕の頭は混乱していた。
「で、この後どうするんでゲソ？」
船を調べてみたところ舵が完全に壊れてしまっている。この状況では今日や明日に船を出すのは不可能だ。どうやらこの島である程度の期間、生活する必要があるようだった。

僕等の船のソフトウェア実装にはできうるかぎり Coq [*1] で証明が付けてあった。船はクリティカルなシステムなので、何らかの方法で安全性を担保する必要があるのだ。ところが、この島では Coq ではなく ATS/LF という証明器しか使えないらしい。そんなわけで僕は ATS/LF について少し調べてみることにした。

1.2 ATS/LF とは

ATS/LF の概要を知るために、まずはそのマニュアルである「ATS プログラミング入門 [*2][*3]」を読んでみた。

それによると、ATS/LF は ATS 言語 [*4] の持つ定理証明サブシステムの名前で、全域関数を使って証明を構築する。証明に使うこの全域関数は証明関数という実行される実体を持たない特殊な関数で、通常の関数と異なりコンパイル時に型検査されるためだけに存在する。カリー＝ハワード同型対応によると計算機プログラムと証明との間には直接的な対応関係があり、「プログラム＝証明」「型＝命題」のように対応づけられるが、まさしくこの対応によると ATS 言語においては関数と型は表 1.1 のような対応を持っていることになる。

動的	型 :=	`fun` キーワードで宣言される関数シグニチャ
	プログラム :=	`implement` キーワードで定義される関数本体
静的	命題 :=	`prfun` キーワードで宣言される証明関数シグニチャ
	証明 :=	`primplement` キーワードで定義される証明関数本体

表 1.1: ATS におけるカリー＝ハワード同型対応

[*1] `https://coq.inria.fr/`
[*2] 英語原著: `http://ats-lang.sourceforge.net/DOCUMENT/INT2PROGINATS/HTML/`
[*3] 日本語訳: `http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/`
[*4] `http://www.ats-lang.org/`

このままでは証明とプログラムは完全に分離されていて関与することができないが、ATS/LF ではプログラムと証明を混ぜて書き下すことができる。関数インターフェイスの意味として関数シグニチャに適切な命題を付記し、関数本体で型と命題が同じ意味になるようにプログラムと証明を混ぜてプログラミングをすることで、証明とプログラムの性質が一致することを期待できる。

少しわかりにくいので整理しよう。結局のところ ATS における関数は次の 3 種類に分類できる (図 1.1)。

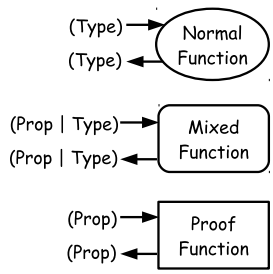


図 1.1: 関数の種類

通常の関数　引数と返り値は型で表わされる。これは他の関数型言語の関数と同様のもの。

混合の関数　引数と返り値は型と命題のタプルで表わされる。これは他の関数型言語の関数に命題が付記されたもの。

証明関数　引数と返り値は命題で表わされる。

これら 3 種類の関数はお互いを呼び出すことができるが、証明関数だけは通常の関数や混合の関数を呼び出すことはできない。これらの関数はお互いを呼び出せるので、実際のアプリケーションはこれらの関数を組み合わせて構築することができる。例えばプログラムのエントリポイントは多くの場合は通常の関数だ。なぜなら POSIX においてプログラムの実行開始と終了には何ら証明的な制約がないからだ。しかしアプリケーションから使うライブラリの呼び出しにおいては、しばしば証明的な制約をつけたくなる場合がある。このような場合、ライブラリ関数のインターフェイスでは混合の関数を使うことがある。この混合の関数を通常の関数から呼び出すと返り値に命題が返されることがある。このような命題は通常の関数本体で束縛しておくことで、別の混合の関数の引数に渡すこともできる。

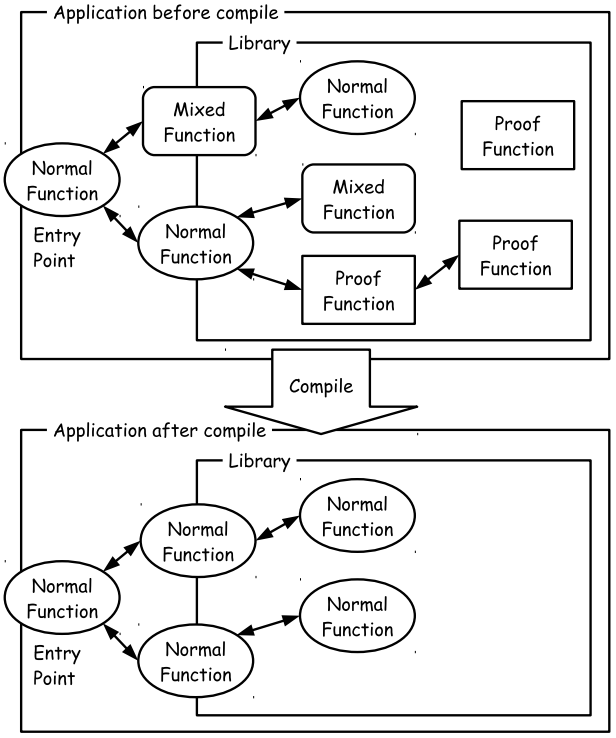


図 1.2: 3 種の関数を組み合わせたアプリケーション

ライブラリ内部においては静的な性質を証明するために、しばしば証明関数を呼び出すことがある。この証明関数は実行時には実体を持たないが、プログラム中の命題に対する関数的な操作を行なうことができる。また、どこからも呼び出されない証明関数も時に有用で、これは証明関数が表わす命題に対して証明を与えたい場合に使われる。

このような 3 種の関数を組み合わせたアプリケーションはコンパイルされると内包する証明コードが型検査された上で削除される (図 1.2)。ATS における証明コードは実行バイナリ中に実体を持

たないのだ。それでも証明コードは関数の呼び出し関係のような静的な性質をコンパイル時に検査するのに役に立つ。

さらにこのプログラムは GC や malloc/free を使わない線形型を使ったコードを対象にすることができる。つまり GC や malloc/free を使わないプログラムに証明を付けることもできるわけだ。

1.3 Coq から ATS/LF へ

ATS 言語におけるキーワード fun を使った関数プログラミングは ML のそれと良く似ていた。キーワード prfun を使った証明関数の使い勝手はどのようなものなのだろうか？

「何をはじめるときも、まずは基礎が大事なんじゃないじゃなイカ？」

そう。僕は「ソフトウェアの基礎 [*5]」を読んで Coq を覚えた。この本には「命題と根拠 [*6]」という章があり、証明のイロハを教えてくれる。この章では曜日にまつわる命題が例として登場する。ATS/LF を習熟するにあたって、この曜日に関する命題から始めるのはどうだろうか。つまり、Coq のコードを真似て ATS/LF のコードを作ってみるのだ。

まず Coq での曜日の定義を見てみよう。

```
(* Coq *)
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.

Inductive good_day : day -> Prop :=
  | gd_sat : good_day saturday
  | gd_sun : good_day sunday.
```

上記は Coq のコードで、最初の Inductive 定義では day という名前で曜日を型定義している。2 つ目の Inductive 定義では good_day という名前で「良い日とは土曜日と日曜日である」という命題を導入している。

これを ATS/LF で書き直すと次のようになる。

```
(* ATS/LF *)
datasort Day =
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
```

[*5] http://proofcafe.org/sf-beta/

[*6] http://proofcafe.org/sf-beta/Prop_J.html

```
  | Sunday

dataprop Good_Day (Day) =
  | Gd_Sat (Saturday) of ()
  | Gd_Sun (Sunday) of ()
```

上記は ATS/LF のコードだ。まず datasort 定義では Day という名前で代数的に曜日を型定義している。この datasort で定義された型は「種 (sort)」と呼ばれて、ATS の静的サイドでのみ使われる。ここでは証明にしか使えない型だと思えば良いだろう。dataprop 定義では Good_Day という名前でやはり「良い日とは土曜日と日曜日である」という命題を導入している。例えば、() に Gd_Sun コンストラクタを適用すると Good_Day (Sunday) という命題をコンストラクトできる。これは「何の前提条件もなしに日曜日が良い日であると言える」ことを意味している。

まったく証明を書いていないが、この ATS/LF コードに問題がないか、型検査してみよう。先の ATS/LF コードを main.dats というファイル名で保存して、ATS コンパイラである patsopt コマンドにかけてみよう。-tc は型検査のみを実行するためのオプションだ。

```
$ patsopt -tc -d main.dats
```

patsopt コマンドは何も表示しない。これは型検査に成功した、つまり証明に問題がないということを意味している。今後しばらく「型検査してみる」と言ったときは、このように main.dats にソースコードを追加して patsopt コマンドを実行してみることにしよう。

さて、型定義と命題を導入したので、なにか簡単な定理を証明してみよう。ソフトウェアの基礎では「日曜日は良い日である」という定理を証明していた。

```
(* Coq *)
Theorem gds : good_day sunday.
Proof. apply gd_sun. Qed.
```

上記の Coq のコードではタクティクを使っている。ATS/LF にはタクティクがないが、コンストラクタを使えばこの定理は証明できる。

```
(* ATS/LF *)
extern prfun gds: Good_Day Sunday
primplement gds = Gd_Sun ()
```

キーワード prfun は証明関数シグニチャの開始だ。つまり命題の型を表わしている。さらにキーワード primplement は証明関数の本体部分の開始になる。証明関数シグニチャは命題、証明関数本体はその証明に対応する。証明関数本体は記述しないこともできるが、それはつまり証明のない命題として導入されていることになる。この ATS/LF コードの場合、Good_Day Sunday 型が命題だ。これを証明するには () に Gd_Sun コンストラクタを適用すれば良いのだった。

ちなみに上記の命題とその証明を見ると命題 Good_Day Monday は成立しないはずだが、命題 Good_Day だけでは成立しないことを立証できない。Good_Day では、Saturday と Sunday についてしか明記していない。Monday の場合に命題 Good_Day がどう扱われるのか何も情報がないのだ。そのため、Good_Day Monday は成立するともしないとも主張できないことになる。

もう少しだけ複雑な証明を書いてみよう。

```
(* Coq *)
Inductive day_before : day -> day -> Prop :=
  | db_tue : day_before tuesday monday
  | db_wed : day_before wednesday tuesday
  | db_thu : day_before thursday wednesday
  | db_fri : day_before friday thursday
  | db_sat : day_before saturday friday
  | db_sun : day_before sunday saturday
  | db_mon : day_before monday sunday.

Inductive ok_day : day -> Prop :=
  | okd_gd : forall d,
      good_day d ->
      ok_day d
  | okd_before : forall d1 d2,
      ok_day d2 ->
      day_before d2 d1 ->
      ok_day d1.

Definition okdw : ok_day wednesday :=
  okd_before wednesday thursday
    (okd_before thursday friday
       (okd_before friday saturday
         (okd_gd saturday gd_sat)
         db_sat)
       db_fri)
    db_thu.
```

上記は Coq のコードだ。最初の `Inductive` 定義では第一引数に渡された曜日の前日が第二引数の曜日であることを主張する命題を `day_before` という名前で導入している。次の `Inductive` 定義では「OK な日とは良い日か良い日の前日である」という命題を `ok_day` という名前で導入している。最後に `okdw` という関数を定義して、`ok_day wednesday` つまり「水曜日は OK な日である」という命題を証明している。

このコードを ATS/LF に翻訳してみたのが次のコードだ。

```
(* ATS/LF *)
dataprop Day_Before (Day, Day) =
  | DB_Tue (Tuesday, Monday) of ()
  | DB_Wed (Wednesday, Tuesday) of ()
  | DB_Thu (Thursday, Wednesday) of ()
  | DB_Fri (Friday, Thursday) of ()
  | DB_Sat (Saturday, Friday) of ()
  | DB_Sun (Sunday, Saturday) of ()
  | DB_Mon (Monday, Sunday) of ()
```

```
dataprop Ok_Day (Day) =
  | {d:Day} Okd_Gd (d) of (Good_Day d)
  | {d1,d2:Day} Okd_Before (d1) of (Ok_Day d2, Day_Before (d2, d1))

prfun okdw: Ok_Day Wednesday = let
  prval okd_sat = Okd_Gd (Gd_Sat ())
  prval okd_fri = Okd_Before (okd_sat, DB_Sat ())
  prval okd_thu = Okd_Before (okd_fri, DB_Fri ())
  prval okd_wed = Okd_Before (okd_thu, DB_Thu ())
in
  okd_wed
end
```

最初の dataprop 宣言では Day_Before という名前でタプルの 1 番目の要素で指定した曜日の前日がタプルの 2 番目で指定した曜日であるという命題を導入している。この命題にはコンストラクタが 7 つあり、例えば () にコンストラクタ DB_Tue を適用すると命題 Day_Before (Tuesday, Monday) がコンストラクトされる。これは別の観点では「火曜日の前日が月曜日である」という命題は無条件に成立する、と解釈することもできる。

次の dataprop 宣言では Ok_Day という名前で第一引数は OK な日であるという命題を導入している。この dataprop 宣言で使っている波括弧は全称量化の宣言で、例えば {d:Day} は「型 Day のどのような d においても」と読む。この命題にはコンストラクタが 2 つあり、例えば Good_Day d にコンストラクタ Okd_Gd を適用すると命題 Ok_Day d がコンストラクトされる。これは別の観点では「d が OK な日である」という命題は「d が良い日である」という条件下でのみ成立する、と解釈することもできる。

最後に prfun キーワードを使って証明関数 okdw を定義している。この証明関数は Ok_Day Wednesday という証明関数シグニチャを持ち、「水曜日は OK な日である」という命題を表わしている。「土曜日は良い日」なので、この定理から「土曜日は OK な日である」という定理が得られる。さらにその定理から「金曜日は OK な日」「木曜日は OK な日」という定理が順に得られ、最後に「水曜日は OK な日」という定理が得られてこれが関数の返り値、すなわち先の証明関数シグニチャと一致できる。キーワード prval は証明変数の束縛宣言だ。例えば、prval okd_sat = Okd_Gd (Gd_Sat ()) はコンストラクタ Okd_Gd が返す命題を束縛して、後の使用にそなえる。ここでは次の行でコンストラクタ Okd_Before が使うことになる。

最後に帰納法の練習として、ソフトウェアの基礎の練習問題を 1 つだけ解いてみよう。

```
(* Coq *)
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS : forall n:nat, ev n -> ev (S (S n)).

Theorem ev_ev_even : forall n m,
  ev (n+m) -> ev n -> ev m.
Proof.
  (* FILL IN HERE *) Admitted.
```

上記は Coq のコードだ。最初の Inductive 定義では偶数を表わす命題を作っている。次の Theorem 定義では「n+m が偶数で、n も偶数ならば、m も偶数である」という命題を関数で表現している。ただしその関数の本体、つまり証明部分は Admitted になっていて空っぽだ。この部分を記述するのが練習問題のようだ。

この練習問題を ATS/LF で解いてみよう。

```
(* ATS/LF *)
dataprop Ev (int) =
  | Ev_0 (0) of ()
  | {n:nat} Ev_SS (n+2) of Ev n

prfun ev_ev_even {n,m:nat} .<n>. (enm: Ev (n+m), en: Ev n): Ev m =
  case+ en of
  | Ev_0 () => enm
  | Ev_SS en' => let
      prval Ev_SS enm' = enm
    in
      ev_ev_even (enm', en')
    end
```

最初に dataprop キーワードを使って偶数であることの命題を定義している。この命題には 2 つのコンストラクタがあって、最初のコンストラクタ Ev_0 は「0 が偶数である」という命題を無条件にコンストラクトできることを表わし、次のコンストラクタ Ev_SS は「n が偶数である」という条件下においてのみ「n+2 が偶数である」という命題をコンストラクトできることを表わしている。

次の関数 ev_ev_even は再帰的な証明関数だ。動的なコードにおいて再帰関数を作る時には不要だったが、証明関数を再帰的に呼び出すときにはその再帰が停止することを保証するために停止性メトリクスが必要になる。上記の例では .<n>. が停止性メトリクスだ。Coq ではこのようなメトリクスは不要だったが、ATS では再帰の度に単調減少するような静的な構造 (自然数、リストなど) をこの停止性メトリクスに指定し、関数が停止することの証左とする。ここでは停止性メトリクスに自然数 n を指定しているので、再帰の度にこの値が小さくならないと型検査エラーになる。この停止性メトリクスの減少は ATS コンパイラが自動的に判定してくれる。case+ キーワードは帰納法の開始だ。ここでは en 変数について場合分けをしている。en 変数が Ev_0 () にマッチした場合、n は 0 なので、enm をそのまま返せば型が合う。というのもこのとき enm の中身は Ev (0+m) だからだ。en 変数が Ev_SS en' にマッチした場合、enm を一段開いて元の関数 ev_ev_even に再帰する。このとき case+ のパターンマッチによって再帰する前と後では n が 1 小さくなっている。このため先の停止性メトリクスは関数 ev_ev_even において成立している。

1.4 実体のない抽象的なリスト

なにか実際の証明をしようにも一般的なリストライブラリを使えた方が楽ができる。ATS には証明を含んだリストライブラリがあり、その証明部分は ilist と呼ばれている。この ilist は命題と証明だけを含むコードで、コンパイル後では実体を持たない。このような実体のないコードを作る理由は、動作するコードの中にある静的な性質を証明するためだ。ATS/LF を使っても動作するコードそのものに直接証明を付けることはできない。そこで、まずは ATS/LF を使って静的な性質を命題として表わした上で証明を付ける。次に具体的な実体を持つリストライブラリで命題を混ぜ書きすることで、先に証明した静的な性質を具体的なリストでも成立させるわけだ。

ATS のソースコード [*7] において ilist に関連するファイルは以下の通りだ。

- libats/SATS/ilist_prf.sats - 抽象的なリストに対する命題
- libats/DATS/ilist_prf.dats - 抽象的なリストに対する証明

まず読んでおくべきなのは ilist_prf.sats で定義されている抽象的なリストの定義だろう。このファイルにおいてリストとは以下のように帰納的に定義されている (図 1.3 も参照)。これは一般的な int を要素に含むリスト構造だ。

```
datasort ilist =
  | ilist_nil of ()
  | ilist_cons of (int, ilist)
```

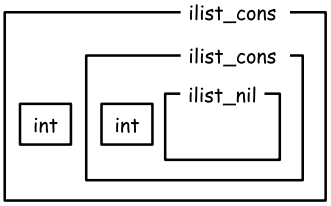


図 1.3: ilist のデータ構造

このリスト ilist を使った命題 LENGTH が定義されている。

```
dataprop LENGTH (ilist, int) =
  | LENGTHnil(ilist_nil, 0) of ()
  | {x:int}{xs:ilist}{n:nat}
    LENGTHcons(ilist_cons (x, xs), n+1) of LENGTH (xs, n)
```

この命題 LENGTH はリスト ilist とその長さの関係をエンコードしている。LENGTH の第一引数がリスト、第二引数がリストの長さを表わす自然数だ。例えば LENGTH (A, 10) と書いた場合、リスト A の長さが 10 であることを意味している。

今度はリストの末尾に要素を追加する命題 SNOC の定義を見てみよう。

```
dataprop SNOC (ilist, int, ilist) =
  | {x:int} SNOCnil (ilist_nil, x, ilist_sing (x)) of ()
  | {x0:int}{xs1:ilist}{x:int}{xs2:ilist}
    SNOCcons (ilist_cons (x0, xs1), x, ilist_cons (x0, xs2))
      of SNOC (xs1, x, xs2)
```

この命題 SNOC はリスト ilist に int の要素を追加する関係をエンコードしている。SNOC の第一引数は要素を追加する前のリスト、第二引数が先のリストに追加する int 型の要素、第三引数が要素を追加した後のリストだ。例えば SNOC (A, 5, B) と書いた場合、リスト A に要素 5 を追加すると、リスト B が得られることを意味している。

これらの命題を使った定理とその証明を見てみよう。次の lemma_snoc_length は LENGTH と SNOC を使った命題だ。

```
prfun lemma_snoc_length
  {xs:ilist}{x:int}{xsx:ilist}{n:int}
  (pf1: SNOC (xs, x, xsx), pf2: LENGTH (xs, n)): LENGTH (xsx, n+1)
```

この命題は SNOC (xs, x, xsx) と LENGTH (xs, n) の存在下では命題 LENGTH (xsx, n+1) を導出できることを示している。日本語で表現すると「長さ n のリスト xs に要素 x を追加したリスト xsx の長さは n+1 である」ということになる。この証明関数に以下のような本体を作ることで、

[*7] https://github.com/githwxi/ATS-Postiats

命題 lemma_snoc_length に証明を与えることができる。

```
primplmnt lemma_snoc_length (pf1, pf2) = let
  prfun lemma
    {xs:ilist}{x:int}{xsx:ilist}{n:int} .<xs>.
    (pf1: SNOC (xs, x, xsx), pf2: LENGTH (xs, n)): LENGTH (xsx, n+1) = let
  in
    case+ pf1 of
    | SNOCnil () => let
        prval LENGTHnil () = pf2 in LENGTHcons (LENGTHnil ())
      end
    | SNOCcons (pf1) => let
        prval LENGTHcons (pf2) = pf2 in LENGTHcons (lemma (pf1, pf2))
      end
  end
in
  lemma (pf1, pf2)
end
```

この証明を読み解くのは比較的簡単だろう。この証明は停止性メトリクスを導入するために、lemma_snoc_length と全く同じシグニチャを持つ証明関数 lemma を内包している。lemma は命題 SNOC (xs, x, xsx) について帰納法を使い、SNOCcons (pf1) の場合には LENGTH (xs, n) も一段分解して再帰する。SNOCnil () に行き着いたら、LENGTH (xs, n) は LENGTHnil () つまり長さ 0 になっているはずで、この場合の返り値は長さ 1 となる。

1.5 リストの具現化

先に見た ilist は抽象的なリストだった。つまりコンパイル時の型検査において使用されるが、コンパイル後では消えてしまう。これではコンパイル時のリスト操作はできても、実行時にはリストを使うことができない。そこでこの ilist を具現化したリストライブラリが 3 種類ある。そのリストライブラリは以下のファイルで実装されている。

- libats/SATS/gfarray.sats - リスト的な証明付き配列のインターフェイス定義
- libats/SATS/gflist.sats - GC を用いた証明付きリストのインターフェイス定義
- libats/SATS/gflist_vt.sats - 線形型を用いた証明付きリストのインターフェイス定義
- libats/DATS/gfarray.dats - リスト的な証明付き配列の実装
- libats/DATS/gflist.dats - GC を用いた証明付きリストの実装
- libats/DATS/gflist_vt.dats - 線形型を用いた証明付きリストの実装

先の説明した通りだが、ここで具現化されたリストは ilist で証明した性質を持っていることが期待されている。というのもこれらのリストは ilist の命題と実行コードを織り交ぜて書かれているためだ。大きな視点では、データ構造と関数シグニチャに ilist の命題が注入されており、これは直感的に元々持っていた関数の型に、証明の命題としての意味を並列して付記する。言語的には、関数の入力と出力を証明と型のタプルを使って表わすことでこの並列された記法を許す。さらに型と命題が混ざった関数本体では、通常の関数や証明関数もしくはその 2 つが並列して付記されたシグニチャを持つ関数を使ってプログラムが構築されている。このようなプログラミングパラダイムを ATS では「定理証明によるプログラミング (Programming with Theorem-Proving (PwTP))」と

呼んでいる。

さて、そんな具現化されたリストライブラリを一つ一つ見ていこう。

1.5.1 gflist

gflist は GC を用いた証明付きリストの実装だ。まずはリスト構造の定義から見てみよう (図 1.4 も参照)。

```
datatype gflist (a:t@ype+, ilist) =
  | gflist_nil (a, ilist_nil) of ()
  | {x:int} {xs:ilist}
    gflist_cons (a, ilist_cons (x, xs)) of (stamped_t (a, x), gflist (a, xs))
```

このリスト定義は datatype で導入されている。つまりその実体はコンストラクタで確保して GC によって解放される。gflist の第一引数は型変数で、リストに内包するデータの型を表わす。第二引数は先に見てきた抽象的なリスト ilist だ。実体となるリスト定義の中に抽象的なリスト定義を持っている、というのは何を意味するのだろう？ このデータ構造を扱う関数の実装を読むとより実感しやすいが、ここでは gflist の静的な性質を ilist で表現しているのだと考えておこう。

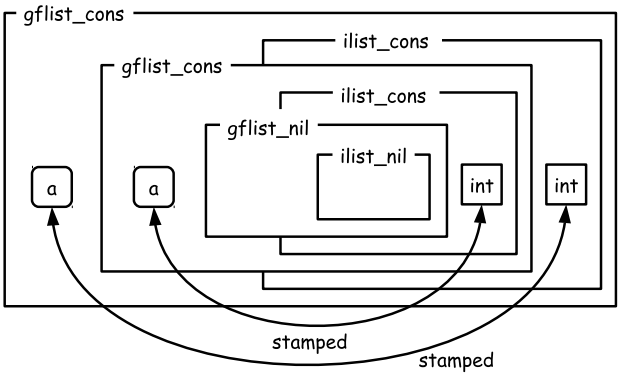


図 1.4: gflist のデータ構造

ちなみに stamped_t (a, x) というのは型変数 a と全量化された int の値を対応づけるための方便だ。このようなテクニックが必要になる理由は ATS/LF では int 型を使った方が証明を行ないやすいからのようだ。この場合、ilist から見て、型変数 a で表わされるリストの内容物とリストそのものの性質とはあまり関係がないと思われるために、その型変数を int 型に写像して扱っている。

次に示すのは gflist を使ったリストの連結関数のシグニチャだ。

```
fun{a:t@ype}
gflist_append
  {xs1,xs2:ilist} (
  xs1: gflist (INV(a), xs1), xs2: gflist (a, xs2)
) :<> [res:ilist] (APPEND (xs1, xs2, res) | gflist (a, res))
```

このリスト連結関数 gflist_append は 2 つの gflist を取り、結合された 1 つの gflist を返す。この時、リストの連結を表わす静的な性質 APPEND (xs1, xs2, res) も一緒にタプルとして返す。つまりこの関数の実装がリストの連結であることが期待できる。このとき、命題と型が混在したタプルには特殊な記号を使うことになっていて、タプルの | 記号の左側が命題、右側が型だ。この関数シグニチャを日本語に直すと「xs1 という性質を持つリストと xs2 という性質を持つリストを結合すると、xs1 と xs2 を結合した性質を持つリストが返る」といったところだろうか。返り値の宣言手前にある角括弧は存在量化の宣言で、例えば [res:ilist] は「型 ilist の値 res が存

在する」と読む。

見方によってはこの gflist_append のインターフェイスは少し弱いとも考えることができるかもしれない。というのも、ilist の命題に以下のようなものがあるからだ。この命題は「xs1 と xs2 を連結したリスト xs の長さは n1+n2 である」と表明している。つまり先の gflist_append 関数のシグニチャではリストの長さの関係が表現されていなかった。

```
prfun
lemma_append_length
  {xs1,xs2:ilist}{xs:ilist}{n1,n2:int} (
  pf: APPEND (xs1, xs2, xs), pf1len: LENGTH (xs1, n1), pf2len: LENGTH (xs2, n2)
) : LENGTH (xs, n1+n2)
```

そこで、gflist_append 関数にリストの長さの命題を付けた、新しい関数 gflist_append_length を次のように作ってみよう。(この関数を含むコード全体はオンライン [*8] から入手できる。)

```
extern fun{a:t@ype}
gflist_append_length
  {xs1,xs2:ilist}{n1,n2:int}
  (pf1: LENGTH (xs1, n1), pf2: LENGTH (xs2, n2) |
    xs1: gflist (a, xs1), xs2: gflist (a, xs2)):
  [res:ilist] (LENGTH (res, n1+n2), APPEND (xs1, xs2, res) | gflist (a, res))

implement{a}
gflist_append_length (pf1, pf2 | xs1, xs2) = let
  val (pf_append | xs) = gflist_append (xs1, xs2)
  prval pf_len = lemma_append_length (pf_append, pf1, pf2)
in
  (pf_len, pf_append | xs)
end
```

関数 gflist_append_length は連結する 2 つのリストに関連する長さの命題を取り、連結したリストの長さの命題を返す。この関数は動的な実装の面では gflist_append 関数と同等だ。その証拠に gflist_append_length の本体では最初に gflist_append を呼び出してリストを連結している。その後、既に証明済みの命題 lemma_append_length を使って、連結したリストの長さを導出している。この長さに関する命題は返り値として使われる。

それでは、作った新しい関数 gflist_append_length を使ってみよう。

```
implement main0 () = {
  val l1 = $list{int}(1, 2, 3)
  val (pf1 | xs1) = list2gflist l1
  val l2 = $list{int}(4, 3, 2, 1)
  val (pf2 | xs2) = list2gflist l2
  val (pf_len, pf_append | xs3) = gflist_append_length (pf1, pf2 | xs1, xs2)
  val (pf3 | l3) = gflist2list xs3
```

[*8] https://github.com/jats-ug/practice-ats/tree/master/test_gflist

```
  val () = print_list<int> l3
}
```

$list{int}(1, 2, 3) は list 型のリテラル定義だ。gflist をコンストラクトするには list 型から list2gflist 関数を使って変換するのがてっとり早い。この list2gflist は以下のような型シグニチャを持ち、変換された gflist 型の値とともに、その長さの命題も返す。

```
castfn list2gflist {a:t@ype}{n:int}
  (xs: list(INV(a), n)) :<> [xs:ilist] (LENGTH (xs, n) | gflist (a, xs))
```

この list2gflist が返した長さの命題を渡すことで、gflist_append_length を呼び出すことができる。glist にはその内容物を印字する関数がないので、関数 gflist2list を使って、gflist_append_length が返した連結済み glist を list 型に変換してから、print_list 関数を使ってコンソールに印字している。

新しい関数 gflist_append_length を作ることで、gflist ライブラリの使い勝手を知ることができた。しかし、証明の面では gflist_append の型シグニチャで十分だ。というのも既に ilist ライブラリ側で同等の関係を証明済みだからだ。gflist に ilist を埋め込んだ後は、gflist を使った関数実装の中で、gflist の加工が ilist の意味と同じになるように気をつければ証明の記述は十分だろう。

また、今回作った gflist 型を使用した main0 関数の中では、明示的なデータの解放は不要だった。これは gflist が datatype を使って定義されていて、GC で管理されるためだ。もし通常のデータ型ではなく、線形型を使う場合には明示的な解放 (線形型の消費) が必要になる。これは次の章でわかるだろう。

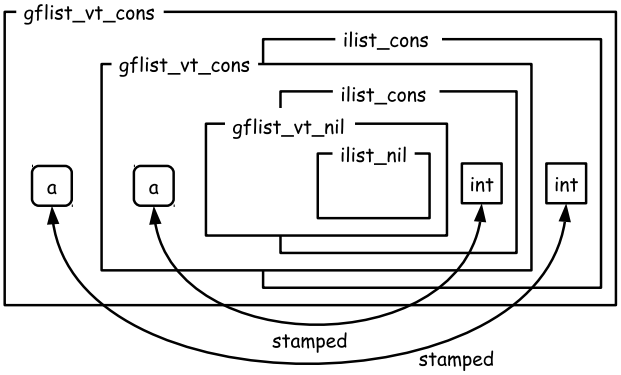


図 1.5: gflist_vt のデータ構造

1.5.2 gflist_vt

gflist_vt は線形型を用いた証明付きリストの実装だ。またリスト構造の定義から見てみよう (図 1.5 も参照)。

```
datavtype gflist_vt (a:vt@ype+, ilist) =
  | gflist_vt_nil (a, ilist_nil) of ()
  | {x:int} {xs:ilist}
    gflist_vt_cons
      (a, ilist_cons (x, xs)) of (stamped_vt (a, x), gflist_vt (a, xs))
```

このリスト定義は dataviewtype で導入されている (datavtype は dataviewtype の短縮名)。つまりこのデータ型は線形型で、その実体の生成と消費をコンストラクタと GC ではなく手動で行なう必要がある。このような型はデータ観型と呼ばれている。線形型はリソースを扱う型で、このような生成/消費の対応を強制する。gflist_vt への引数は gflist と同じだ。これは gflist_vt が gflist の特性はそのままに線形型を使った表現になっていることになる。

gflist と同様に、gflist_vt に対するリストの連結関数 gflist_vt_append を見てみよう。

```
fun{a:vt@ype}
gflist_vt_append
  {xs1,xs2:ilist} (
  xs1: gflist_vt (INV(a), xs1), xs2: gflist_vt (a, xs2)
) :<!wrt> [res:ilist] (APPEND (xs1, xs2, res) | gflist_vt (a, res))
```

この gflist_vt_append の関数シグニチャは gflist_append とほとんど同じに見えるが、gflist_vt が線形型であるために、その意味は少し異なる。第一引数と第二引数に ! 記号がついていないため、この gflist_vt_append 関数は xs1 と xs2 のデータ構造を解放 (消費) して、返り値として新しいデータ構造 gflist_vt (a, res) を生成することを表わしている。これは gflist_vt は ilist のような性質を持つが、データ構造の取り扱いは異なるということになる。このように ATS/LF と PwTP を使うことで静的な性質と動的な振舞いを緩く結びつけることができる。また、動的なコードと静的な証明が 1 つのソースコード中に混ざって書き下されるために、両者が乖離してしまうことも防げる。

ここでは gflist_vt_append の使用感をつかむために gflist_vt_append に長さに関する性質を加えてみよう。(この関数を含むコード全体はオンライン [*9] から入手できる。)

```
extern fun{a:t@ype}
gflist_vt_append_length
  {xs1,xs2:ilist}{n1,n2:int}
  (pf1: LENGTH (xs1, n1), pf2: LENGTH (xs2, n2) |
    xs1: gflist_vt (a, xs1), xs2: gflist_vt (a, xs2)):
  [res:ilist] (LENGTH (res, n1+n2), APPEND (xs1, xs2, res) | gflist_vt (a, res))

implement{a}
gflist_vt_append_length (pf1, pf2 | xs1, xs2) = let
  val (pf_append | xs) = gflist_vt_append (xs1, xs2)
  prval pf_len = lemma_append_length (pf_append, pf1, pf2)
in
  (pf_len, pf_append | xs)
end

implement main0 () = {
  val l1 = $list_vt{int}(1, 2, 3)
  val (pf1 | xs1) = list2gflist_vt l1
  val l2 = $list_vt{int}(4, 3, 2, 1)
  val (pf2 | xs2) = list2gflist_vt l2
  val (pf_len, pf_append | xs3) = gflist_vt_append_length (pf1, pf2 | xs1, xs2)
  val (pf3 | l3) = gflist2list_vt xs3
  val () = print_list_vt<int> l3
  val () = free l3
```

[*9] https://github.com/jats-ug/practice-ats/tree/master/test_gflist_vt

```
}
```

gflist_vt_append_length の型シグニチャと本体は gflist_append_length とそっくりだ。唯一目立った違いは main0 関数の最後で、リスト l3 を明示的に解放しているところだ。これは list_vt 型と gflist_vt 型が両方とも線形型であることに由来している。gflist_vt_append_length など線形型の引数を取る関数はその引数を消費する (解放する) が、リスト l3 だけは gflist2list_vt 関数で生成された後、誰も解放しないために、明示的な解放が必要になる。この明示的なリソースの解放は少しわずらわしく感じられるが、解放し忘れや二重解放はコンパイル時エラーになる。GC が使えない状況下では線形型が唯一のデータ管理の方法となるため、この特性は有用だ。また線形型を使わないデータ型を使用した場合と、線形型を使ったデータ観型を使用した場合とで、コードの構造はあまり変わらない。しかもその GC を使わない線形型を使ったコードに対して ATS/LF は証明を付けることができるのだ。

1.5.3 gfarray

gfarray はリスト的な証明付き配列の実装だ。また配列構造の定義から見てみよう。

```
dataview gfarray_v (a:vt@ype+, addr, ilist) =
  | {l:addr}
    gfarray_v_nil (a, l, ilist_nil) of ()
  | {x:int}{xs:ilist}{l:addr}
    gfarray_v_cons (a, l, ilist_cons (x, xs)) of
      (stamped_vt (a, x) @ l, gfarray_v (a, l+sizeof(a), xs))
```

この配列定義は dataview で導入されている。つまりその実体は生のメモリに格納されていて、ポインタの先のデータを表わす駐観 (at-view) によってそのデータにアクセスする。この定義には引数が 3 つある。その内 2 つ、つまり第一引数と第三引数は gflist や gflist_vt と同様、型変数と抽象的なリスト ilist だ。残る第二引数 addr はポインタを表わすアドレスで、これはこの配列の先頭アドレスを指している。このアドレスの用途は少しわかりにくいが、gfarray_v_cons コンストラクタを見ればわかりやすい (図 1.6 も参照)。このコンストラクタは (stamped_vt (a, x) @ l, gfarray_v (a, l+sizeof(a), xs)) という条件下において、gfarray_v (a, l, ilist_cons (x, xs)) を成すことができると言っている。stamped_vt (a, x) @ l はアドレス l に型変数 a の実体があることを意味し、gfarray_v (a, l+sizeof(a), xs) はアドレス l から 1 要素分老いたアドレスが gfarray_v 型であることを意味している。つまり gfarray_v_nil から若いアドレス方向に向かってこの配列はリスト的に成長していることがわかる。

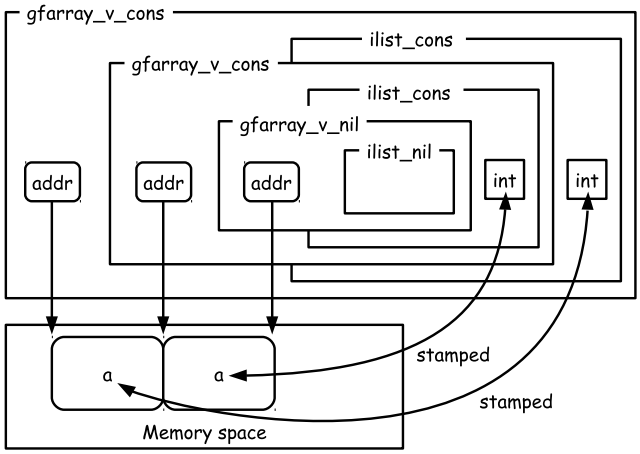


図 1.6: gfarray のデータ構造

gfarray はベタなメモリ空間上のデータ構造を扱うので、リスト的なユーティリティ関数を作ることができないが、それでもリストの分割関数などを作ることはでき、それが以下の gfarray_v_split だ。

```
prfun
gfarray_v_split
  {a:vt0p}{l:addr}{xs:ilist}{n:int}{i:nat | i <= n}
(
  pflen: LENGTH (xs, n)
, pfarr: gfarray_v (a, l, xs)
) : [xs1,xs2:ilist]
(
  LENGTH (xs1, i)
, LENGTH (xs2, n-i)
, APPEND (xs1, xs2, xs)
, gfarray_v (a, l, xs1)
, gfarray_v (a, l+i*sizeof(a), xs2)
)
```

この関数は長さ n のリスト xs を i 番目の位置で 2 つのリスト xs1 と xs2 に分割している。また、この関数は純粋な証明関数として実装されている。gfarray_v は純粋な線形型であったために、データ構造が実体を持たないのだ。実体を持たないのであれば、リストの要素はどこに格納されているのだろうか？ それは先に見た通りベタなメモリアドレスに格納されている。gfarray_v が内包している駐観は、このベタなメモリアレイに格納された要素に触ることができるか (デリファレンスできるか) を表わす切符のようなものだと考えるといいかもしれない。先の gfarray_v_split はデータ構造を弄るのではなく、この駐観という静的な切符をやりとりするだけなので、純粋な証明関数で済んでいた。当然だが、この gfarray_v_split 関数はコンパイル後には実体を持たない。当然実行時のオーバーヘッドもゼロだ。

ベタなメモリにリスト的なデータ構造を持ち込む gfarray の特性を生かして、ATS のソースツリーには証明付きバイナリサーチのソースコード [*10] もあった。今の自分にはこの証明コードを読み解くのは骨だったが、同様の証明を書く際には役に立ちそうだった。

1.6 回文の性質

「大変なことがわかったでゲソ! 回文でない会話はこの島では失礼らしいじゃなイカ!」
なんと。とすると、この島で暮すには回文を安全に扱えた方がいい。ちょうどさっきまで証明付きリストライブラリである ilist とその実体である gflist を調べていたので、この 2 つを作って回文を扱うライブラリを証明付きで設計してみよう。(この回文ライブラリのソースコードはオンライン [*11] から入手できる。) 回文の命題とその定理を証明した後、その命題を混ぜ書きした回文ライブラリを作るのだ。

まずは回文の命題を導入しよう。

```
dataprop PAL (ilist) =
  | PALnil (ilist_nil) of ()
```

[*10] https://github.com/githwxi/ATS-Postiats/blob/master/doc/EXAMPLE/PCPV/bsearch_arr.dats
[*11] https://github.com/jats-ug/practice-ats/tree/master/atslf_palindrome

```
  | {x:int} PALone (ilist_sing (x)) of ()
  | {x:int}{l,ll:ilist}
    PALcons (ilist_cons (x, ll)) of (PAL (l), SNOC (l, x, ll))
```

この命題は少しわかりずらい。コンストラクタ PALnil は無条件に空の回文の命題を生成する。もう一つのコンストラクタ PALone は要素が 1 つだけの回文を生成する。1 つしか要素がないのであれば、回文であることは自明だからだ。最後の PALcons は回文の帰納的な表現で、回文なリスト l の末尾に 要素 x を追加したリスト ll の先頭に要素 x を追加したリストは回文であると読める。

さらに回文に対する操作の命題を作ろう。次の命題 PAPPEND は回文とリストを連結する操作の関係をエンコードしている。命題 PAPPEND の第一引数が連結する前の回文、第二引数が連結するリスト、第三引数が連結した結果の回文だ。

```
dataprop PAPPEND (ilist, ilist, ilist) =
  | {pxs:ilist} PAPPENDnil (pxs, ilist_nil, pxs) of PAL (pxs)
  | {pxs,pxsx,ys,pzs:ilist}{x:int} PAPPENDcons (pxs, ilist_cons (x, ys), pzs)
      of (SNOC (pxs, x, pxsx), PAPPEND (ilist_cons (x, pxsx), ys, pzs))
```

次に何か証明すべき定理の命題を 1 つだけ宣言してみよう。次の命題は「リスト l とその逆順である lr を連結したリスト m は回文」であることを表明している。

```
prfun pal_app {l,lr,m:ilist}
  (pf1: REVERSE (l, lr), pf2: APPEND (l, lr, m)): PAL (m)
```

この命題にはまだ本体がないので、証明されていない。証明関数本体を書いて証明をしたいが、少し楽をするために補題、つまり証明関数本体のない証明関数シグニチャを導入しよう。

```
extern prfun
lemma2_reverse_scons {x:int}{xs:ilist}{ys1:ilist}
  (REVERSE(ilist_cons(x,xs), ys1)):
  [ys:ilist] (REVERSE(xs, ys), SNOC(ys, x, ys1))

extern prfun
lemma2_append_scons {x:int}{xs,ys:ilist}{ys1,zs1:ilist}
  (APPEND(xs, ys1, zs1), SNOC(ys, x, ys1)):
  [zs:ilist] (APPEND(xs, ys, zs), SNOC(zs, x, zs1))
```

これらの補題があれば、命題 pal_app を簡単に証明できる。

```
primplement
pal_app {l,lr,m} (pf1, pf2) = let
  prfun lemma  {l,lr,m:ilist} .<l>.
      (pf1: REVERSE (l, lr), pf2: APPEND (l, lr, m)): PAL (m) =
    case+ pf2 of
    | APPENDnil () => let prval REVAPPnil () = pf1 in PALnil () end
    | APPENDcons(pf2) => let
        prval (pfrev, pfsnoc) = lemma2_reverse_scons (pf1)
        prval (pfapp, pfsnoc2) = lemma2_append_scons (pf2, pfsnoc)
```

```
        prval pfpal = lemma (pfrev, pfapp)
      in
        PALcons (pfpal, pfsnoc2)
      end
in
  lemma (pf1, pf2)
end
```

他にも証明すべき定理が回文にはあると思われたが、とりあえず証明側はこれだけで済まそう。

1.7 回文を扱う関数

先に回文の性質を ATS/LF を使って証明した。しかし、証明コードだけでは実際に回文を取り扱うことはできない。あくまで回文の性質を証明しただけだからだ。先に ATS/LF で設計した性質を組み込みながら回文ライブラリを作ってみよう。最初にこれから作る関数を想定して関数シグニチャを決めよう。

```
fun{a:t@ype} pal_empty
  (): (PAL (ilist_nil) | gflist (a, ilist_nil ()))
fun{a:t@ype} pal_sing
  {x:int} (x: stamped_t (a, x)):
  (PAL (ilist_sing(x)) | gflist (a, ilist_sing(x)))
fun{a:t@ype} pal_pappend
  {pxs,xs:ilist}
  (pf: PAL (pxs) | pxs: gflist (INV(a), pxs), xs: gflist (a, xs)):
  [pxsx:ilist] (PAL (pxsx), PAPPEND (pxs, xs, pxsx) | gflist (a, pxsx))
fun{a:t@ype} print_pal
  {xs:ilist} (pf: PAL (xs) | xs: gflist (INV(a), xs)): void
```

今回回文ライブラリを作るにあたり、具現化されたリストとして gflist を使うことにした。やはり GC を前提としたデータ構造を使った方が楽ができるからだ。先にも見た通りだが、GC を前提にした datatype を線形型を使う dataviewtype に変換するのは比較的簡単だった。今回のライブラリも線形型を使う必要性が生じた時点で gflist_vt を使うように変更すればいいだろう。

個々の関数シグニチャを見てみよう。pal_empty は中身が空の回文を作り、回文の性質である PAL と回文の実体である gflist 返す。pal_sing は要素が 1 つだけの回文を作り、やはり PAL と gflist を返す。この 2 つの関数で得られるリストは回文であることは自明なので、引数に命題を渡す必要はない。pal_pappend はなんらかの回文の性質を持つリストに通常のリストを連結して、回文を返す。この時、pal_pappend は結果が回文であるという性質 PAL と一緒に、行なわれた操作が回文の連結であることを示す命題 PAPPEND を返す。PAPPEND をこのライブラリのユーザが使うことはないが、pal_pappend の動作が回文的な連結の性質を持っていることを期待することができる。最後の print_pal は回文をコンソールに印字する。この関数は回文のリストである gflist だけでなく、その性質が回文であることを表わす命題 PAL も同時に要求している。そのため、print_pal を使ってコンソールに印字するかぎり、回文でない文章が画面に表示されることはないはずだ。

関数シグニチャを決めたので、関数本体を実装しよう。最初の 2 つの関数、pal_empty と pal_sing の実装は簡単だろう。両方とも単に命題と型をコンストラクタによって作るだけだ。

```
implement{a}
pal_empty () = (PALnil () | gflist_nil ())

implement{a}
pal_sing (x) = (PALone () | gflist_cons (x, gflist_nil))
```

ライブラリの中心的役割を持つ pal_pappend を実装しよう。この実装は PAPPEND の性質を満たすように証明を混ぜ書きする必要がある。

```
implement{a}
pal_pappend (pfpal | pxs, xs) = let
  fun loop {pxs,xs:ilist} .<xs>.
      (pfpal: PAL (pxs) | pxs: gflist (a, pxs), xs: gflist (a, xs)):
      [pxsx:ilist] (PAL (pxsx), PAPPEND (pxs, xs, pxsx) | gflist (a, pxsx)) =
    case+ xs of
    | gflist_nil () => (pfpal, PAPPENDnil pfpal | pxs)
    | gflist_cons(x, xs1) => let
      val (pfsnoc | pxs) = gflist_snoc (pxs, x)
      val pxs = gflist_vt2t pxs
      val (pfpalx, pfappend | pzs) =
        loop (PALcons (pfpal, pfsnoc) | gflist_cons (x, pxs), xs1)
      prval pfappendx = PAPPENDcons (pfsnoc, pfappend)
    in
      (pfpalx, pfappendx | pzs)
    end
in
  loop (pfpal | pxs, xs)
end
```

最後に画面に印字する関数である print_pal を作ろう。この関数は入力で渡された gflist を通常の list に変換して画面に印字する。入力されたリストが回文であるかどうかは関数シグニチャで制約されているので、関数本体でケアする必要はないからだ。

```
implement{a}
print_pal (_ | xs) = let
  val (_ | xs) = gflist2list xs
in
  print_list<a> xs
end
```

これでやっと今回作った回文ライブラリを使ったアプリケーションを組むことができる。

```
implement main0 () = {
  // Pullup
  val (pfpal | lpal) = pal_empty ()
  val (_ | l) = list2gflist $list{char}('L', 'U', 'P')
```

```
  val (pfpal, _ | lpal) = pal_pappend (pfpal | lpal, l)
  val () = print_pal<char> (pfpal | lpal)
  val () = print "\n"
  // Devil never even lived.
  val (pfpal | lpal) = pal_sing<char> (stamp_t 'R')
  val (_ | l) = list2gflist
                  $list{char}('E', 'V', 'E', 'N', 'L', 'I', 'V', 'E', 'D')
  val (pfpal, _ | lpal) = pal_pappend (pfpal | lpal, l)
  val () = print_pal<char> (pfpal | lpal)
  val () = print "\n"
}
```

この main0 関数では 2 つの回文を構築して印字している。まず “PULLUP” という回文を作るために、空の回文を pal_empty で作る。この空の回文に “LUP” を回文的に連結して、“PULLUP” を得る。次に “DEVILNEVEREVENLIVED” という回文を作るために、回文の中央の文字 “R” のみを含む回文を pal_sing で作る。この要素が 1 つだけの回文に “EVENLIVED” を回文的に連結して、“DEVILNEVEREVENLIVED” を得る。

このアプリケーションをコンパイルして実行すると回文をコンソールに印字する。

```
$ patscc main.dats -DATS_MEMALLOC_LIBC
$ ./a.out
P, U, L, L, U, P
D, E, V, I, L, N, E, V, E, R, E, V, E, N, L, I, V, E, D
```

これでこの島の住人とも失礼なく会話できるだろう。

1.8 住めば都

「いい風じゃなイカー」

そうだ。暮してみればこの ATS/LF しか証明器のないこの島での生活も悪くなかった。ATS/LF が証明からコードを自動的に吐き出さないのは一見デメリットに思える。なぜなら、手書きで書いた実行コードに証明を混ぜても、その実行コードが完全に安全であるとは言えないからだ。しかしこれは逆にメリットになる場面もある。その 1 つは実行コードを簡単には証明できない場合だ。そのような場合、ATS/LF は本来証明すべき証明より弱い証明を作り、その証明を手書きの実行コードに関与させることで実行コードの安全性と設計効率をバランスさせることができる。もう 1 つは効率的な実行コードを得たい場合だ。この場合は証明サイドでは本質的なアルゴリズムを実装し、その証明と効率的な実行コードを混ぜて書くことで実行コードの安全性をある程度担保できる。

さらにこの島では SMT ソルバである Z3 [*12] を ATS コンパイラの外部ソルバ [*13] として使えるようになりつつあるようだ。上手くいけば ATS/LF における証明が半自動化できるかもしれない。

1.9 ライセンス

[*12] https://github.com/Z3Prover/z3

[*13] http://www.illtyped.com/projects/patsolve/

第2章

二重否定と継続

— @nushio

2.1 悪魔の契約

『契約の獣キュゥべえ (以下甲) と少女 (以下乙) は以下のように契約する。

(1) 甲は乙に 1 億円支払う、または、(2) 乙が甲に 1 億円支払えば、甲は乙の願いを何でも叶えてあげる、ただし、(1)、(2) の選択権は甲が有する。甲乙の間に、魂の引き渡しや加工等といった、上記以外の権利義務お節介は発生しない』

などという契約をさせる詐欺が流行っているそうなので、皆さん気をつけるように。さて、先に進む前に、まずは考えてみたい。あなたが少女の立場だとして、この契約に価格をつけるとしたらいくらの価値があるだろうか？

常識的な解釈では、次のようになるのではないか。(1)、(2) の選択権はキュゥべえにあるわけだが、もしキュゥべえが (1) を選択してきたなら、あなたはタダで 1 億円を手に入れる。(2) を選択してきたなら、なんとかして 1 億円を調達しないといけないが、その見返りは途方もなく大きい。少なくとも、借金をして 1 億円を調達し、2 億円と利子を願うことができる。したがってこちらのオプションの価値も 1 億円以上ということになる。借金が無理ならもっとやましい手段で資金を調達したうえで、その証拠も記憶も、完全に消してしまうこともできるし、望むなら全能の神にだってなれるわけだ。

ところが、実はこの契約には一銭の価値もないのだ。キュゥべえは何の自己負担もなく、この契約を履行してしまえる。後には『願いを何でも叶えてあげる』という甘言に釣られた、少女の後悔だけが残るだろう。そして第二次性徴前の少女が 1 億円もの大金を調達するとなると、その手段は大きな後悔を伴うものに限られるに違いない。

エロ同人みたいに！

エロ同人みたいに！

キュゥべえはいったいどのような手を使ってこの契約を履行するのか？ このような甘言を振りまいて、少女を絶望に陥れるキュゥべえから身を守るにはどうすればいいのか？

それを、プログラミング言語と型の理論から解き明かしていくのが、今回のテーマだ。

2.2 悪魔のプログラム

この『契約』を利用する戦略は、プログラムで書けば以下のようになるだろう。

```
useTheContract :: Either Money (Money -> IO' Void) -> IO' ()
useTheContract c = case c of
  Left money -> do
    story $ "さやかちゃん：いきなりお金を手に入れた。ラッキー！ " ++ show money
  Right qbey -> do
```

```
  story "さやかちゃん：お金を工面してキュゥべえに渡すよ。"
  (g :: Void) <- qbey Money
  story   "さやかちゃん：神になったよ！ 自分で自分を褒めてあげたい感じ？ "
  story $ "さやかちゃん：とりあえずお金を倍に増やして借金返して: "
       ++ show (absurd g :: Money)
       ++ show (absurd g :: Money)
  story $ "さやかちゃん：それから満漢全席食べるよ！ : "
       ++ show (absurd g :: MankanZenseki)
```

このプログラムは、まえにのべた契約に価格をつける戦略を Haskell のプログラム化したものだ。ここで、`Either a b` は a 型または b 型の値のどちらかが入っている型だ。また、`Void` とは、hackage にあるライブラリ `void` に属するモジュール `Data.Void` の提供する型である。`Void` を特徴づけるのは、次のような関数 `absurd` の存在だ。

```
absurd :: Void -> a
```

信じがたいことに、`Void` 型の値がひとつでもあれば、それを `absurd` に適用することで、どんな型の値でも作り出すことができるのだ。[*1]

以上を踏まえれば、上記のプログラム `useTheContract` の引数の型 `Either Money (Money -> IO' Void)` が、くだんのキュゥべえの契約に対応しているという気分がわかってもらえるだろうか。`IO'` というのは IO モナドの機能を拡張して継続が使えるようにしたものだ。継続とは何か... それが今回の話のネタなので、解説をお待ちいただきたい。

では、このプログラムを実行してみよう。じつは、この契約を提供するキュゥべえ側は、以下のプログラムにより、この契約を履行することができる。

```
type Contract = Either Money (Money -> IO' Void)

offerContract :: IO' Contract
offerContract = callCC f
  where
    f :: (Contract -> IO' Void) -> IO' Contract
    f k = do
      story "キュゥべえ：ボクは (2) を選ぶよ。"
      return (Right (\x ->
        story "キュゥべえ：ボクは (1) を選ぶよ。"
          >> k (Left x)))
```

キュゥべえとさやかちゃんを対話させるには、以下のコードが必要だ。

```
{-# LANGUAGE ScopedTypeVariables #-}
import Control.Monad
import Control.Monad.IO.Class
import Control.Monad.Cont
import Data.Void
```

[*1] このような `Void` だが、何の問題もなく Haskell のプログラムとして実装することができる。気になる方はライブラリ `void` のソース、もしくはこの章の最後にある独立版を参照いただきたい。

```
-- Narrate the story
story :: MonadIO m => String -> m ()
story = liftIO . putStrLn

-- A lot of money.
data Money = Money
instance Show Money where show _ = "[Money]"
-- Sayaka-chan's favorite food.
data MankanZenseki = MankanZenseki deriving (Show)

-- The IO monad equipped with continuation
type IO' = ContT () IO

main :: IO ()
main = runContT (offerContract >>= useTheContract) return
```

以上 3 つのコード片を結合すると実行可能な Haskell のプログラムができる。fairly-tale-qbey.hs というファイル名で保存したとして、実行した結果はこうなる。

```
$ runhaskell fairly-tale-qbey.hs
キュゥべえ：ボクは (2) を選ぶよ。
さやかちゃん：お金を工面してキュゥべえに渡すよ。
キュゥべえ：ボクは (1) を選ぶよ。
さやかちゃん：いきなりお金を手に入れた。ラッキー！ [Money]
$
```

何が起こっているのか、すぐには理解しがたい。出力結果を順に追っていこう。

プログラムの出力を信じるなら、キュゥべえはまず (2) を選択している。従ってさやかちゃん側のプログラムの冒頭のパターンマッチでは Right の分岐が実行されるはずだ。実際、Right の分岐の冒頭のストーリーが表示されてはいるのだが...

```
useTheContract c = case c of
  Left money -> do
    ...
  Right qbey -> do
    story "さやかちゃん：お金を工面してキュゥべえに渡すよ。"
    (g :: Void) <- qbey Money
    story   "さやかちゃん：神になったよ！ 自分で自分を褒めてあげたい感じ？ "
```

どうやら、さやかちゃんが Money (一億円) を工面して qbey に渡した後、g :: Void を受け取る前までの間に、キュゥべえ側の「ボクは (1) を選ぶよ。」というメッセージが表示され、それに呼応して、さやかちゃんの Left 側の分岐が実行されて、プログラムはそのまま終了しているのだ。

実は、ここでキュゥべえは、まず (2) を選んでおいて、さやかちゃんから一億円を受け取った瞬間に、「自分が (1) を選んだ世界線」に飛び、さやかちゃんから受け取った一億円をただそのまま渡したのだ。Left 側のさやかちゃんは、無邪気に喜んでいる。その受け取った一億円が (2) の世界線

論理	型
$A \wedge B$ 論理積、A かつ B	`(a, b)` タプル、`a` と `b` の組
$A \vee B$ 論理和、A または B	`Either a b` `Either` 型、`a` または `b` のいずれか
$A \Rightarrow B$ 導出、A ならば B	`a -> b` 関数型、`a` から `b` への関数
$\neg A$ 否定、A ではない	`a -> Void` 「`a` ならば矛盾」
$\top$ 恒真	`()` 唯一の値が属するような型
$\bot$ 矛盾、	`Void` それに属する値がないような型

表 2.1: カリー・ハワード対応で対応する論理式と型の例。

の自分自身が大きな代償を払って稼いだものであることも、(2) の世界線全体がその一億円を生み出すためだけに実行され、キュゥべえが脱出したあとは実行されず放置されていることも知るよしもないのだ。

main プログラムを見るかぎり、

```
main :: IO ()
main = runContT (offerContract >>= useTheContract) return
```

キュゥべえ側のプログラム `offerContract` は、さやかちゃん側のプログラム `useTheContract` に、ただ `Contract` 型の値を一度渡しているだけに思える。

キュゥべえに対し、さやかちゃんのプログラムの、`Left` と `Right` の両方の分岐を実行し、あまつさえ片方の分岐の途中で自分に制御を戻す、などという人生をもてあそぶ力を与えているのが**継続**だ。契約を履行、いやプログラムを実行している最中に、「やっぱり選択肢を選び直します」などという、通常の感覚では許されない行為を可能にしてしまうのが継続だ。しかも Haskell において継続を実装するのに、IO モナドや処理系に手を加える必要はない。継続は、IO モナドに限らず、任意のモナドと合成して使える、モナドトランスフォーマとして、ごく通常の外部ライブラリとして実装されている。

なぜそのようなことができるのか？

そのからくりを理解するために、まずはカリー・ハワード対応について復習しておこう。

2.3　カリー・ハワード対応

カリー・ハワード対応とは、命題の世界と型の世界をたがいに関連付ける、プログラミング言語理論の重要な発見だ (表 2.1)。カリー・ハワードで対応する命題 A と型 `a` の間には、「命題 A が正しい」と「型 `a` を持つ、エラーにならず完了するプログラムを書くことができる」とが同値である、という関係がある。このことから、プログラムの性質を命題論理をつかって推論することができるし、逆に十分強力な型推論器をもつプログラミング言語ならば、その言語の型で証明が書けてしまう。この性質は、この本でも紹介する ATS/LF や Coq, Agda といった証明系プログラミング言語においてもっともよく活用されている。

カリー・ハワード対応の例をいくつか例を挙げてみよう。

`(a->b, b->c) -> (a->c)` は $(A \Rightarrow B) \wedge (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$ に。

`(a,b) -> a` は $A \wedge B \Rightarrow A$ に。

`Either a b -> Either b a` は $A \vee B \Rightarrow B \vee A$ に対応している。

この 3 つの組のいづれについても、それぞれの型をもったプログラムを容易に書くことができる。そして、対応する命題は明らかに正しい。

具体的に、カリー・ハワード対応を使ってなにか証明してみよう。次のプログラムは、型名がほとんど日本語で、型注釈が多めに付けられてはいるが、れっきとした Haskell のコンパイル可能で停止するプログラムである。

```
{-# LANGUAGE ScopedTypeVariables #-}

推理 :: forall 新幹線 在来線 米原を通る.
  (Either 新幹線 在来線, 新幹線 -> 米原を通る, 在来線 -> 米原を通る) -> 米原を通る
推理 (x :: Either 新幹線 在来線,
      p ::新幹線 -> 米原を通る,
      q :: 在来線 -> 米原を通る) =
  case x of
    Left  (y :: 新幹線) -> (p y :: 米原を通る)
    Right (z :: 在来線) -> (q z :: 米原を通る)
```

これは、探偵小説に出てきそうな推理を表現している。

定理. 犯人は新幹線もしくは在来線に乗って逃亡した、かつ、新幹線は米原を通る、かつ、在来線は米原を通る、ということから、犯人は米原を通る、ということが言える。

推理. x :: 犯人は新幹線もしくは在来線に乗って逃亡した、p :: 新幹線は米原を通る、q :: 在来線は米原を通る

を仮定して結論を導く。

まず、x より、

- y :: 犯人は新幹線に乗っている
- z :: 犯人は在来線に乗っている

のいずれかである。

もし犯人が新幹線に乗っているならば、y に p を適用することにより、犯人は米原を通る。

もし犯人が在来線に乗っているならば、z に q を適用することにより、犯人は米原を通る。

従って、いずれの場合にも犯人は米原を通るので、米原で網を張ればよい。 □

カリー・ハワード対応の紹介が済んだので、「悪魔の契約」をカリー・ハワード対応の観点から考察していこう。

悪魔の契約のカリー・ハワード対応物は、`(Either a (a->Void))` 型のプログラムを常に書くことができる、ということだ。これは不思議な事実だと思わないか？ 任意の型 a に対して `(Either a (a->Void))` 型の値が作れるとは。そんな値を作るには、a 型の値か `a->Void` 型の値のどちらかを用意しないといけないが、任意の型 a に対して a 型の値を用意しておく、というのはちょっとできそうもないし、`a->Void` 型の関数を書く、というのはもっと難しそうだ。

ところが、型 `(Either a (a->Void))` にカリー・ハワード対応する命題は $A \vee \neg A$ である。これ

は「A または not A」と言っている。真偽がはっきりしている命題の世界で「A または not A」を認めるのは自然な感覚だろう。「すべての自然数は偶数または非偶数 (奇数)」であるし、「なのはちゃんの年齢は 9 歳以上または 9 歳未満」である。これは排中律 (二重否定の除去) と呼ばれる、古典論理の公理のひとつだ。

ところが、「A または not A」を公理として認めると不都合が生じる。それは、古典論理の世界では証明できてしまう、『悪魔の契約』のようなものを記述できてしまうことだ。ここで、排中律を公理として認めない「直観主義論理」をの創始者とされるのがブラウワー (Luitzen Egbertus Jan Brouwer, 1881-1966) だ。排中律を認めなければ、背理法も使えない。これまで慣れ親しんできた数学の証明のうち、背理法を使うものは無効になってしまうのだ。

$A \vee \neg A$ という命題は、排中律に親しんだ私たちにとっては自然に思えるのに、この命題とカリー・ハワード対応する型 `Either a (a->Void)` をもつプログラムは、継続という詐術めいたトリックを使わなければとうてい書けそうには思えない。古典論理の公理である「排中律」と、『悪魔の契約』をも記述可能にしてしまう「継続」。ふたつの概念のギャップは大きいように見える。

しかし、継続は、排中律を公理として認めると当然ついてくる結果なのだ。このことの証拠として、Haskell プログラム上で、「排中律を公理として導入し、それを用いて継続を実装する」デモをお見せしよう。じつは、Haskell の継続モナドライブラリは、まさにこの方法で実装されているのだ。

2.4　継続モナドの実装

この節では Haskell における継続モナドの実装を紹介する。ただし、実際の継続モナド `Control.Monad.Cont` の実装は、応用性を高めるため、より複雑な実装を採用している。ここでは「すごい H 本」風に、より原始的だが理解しやすい、独自の実装を解説する。

さて、排中律を認めるということは、ある命題 A とその二重否定 $\neg\neg A$ が同値になる、ということだ。この二重否定 $\neg\neg A$ はプログラムの世界では `(a -> Void) -> Void` に対応する。

私は二重否定の中で計算をしている、という文脈を型コンストラクタ `M` で表そう。もうネタバレしているが、`M` はモナドになる。

```
data M a = M { runM :: (a -> Void) -> Void}

instance Monad M where
  return a0 = M ($ a0)
  m >>= k   = M $ \ c -> runM m (\ x -> runM (k x) c)
```

上記の型宣言文は、Haskell のふつうの型宣言文であって特別なことはしていない。これで Haskell 型上の明示的な二重否定 `(a -> Void) -> Void` と、二重否定をくるんだ文脈 `M a` との相互変換が使えるようになる。

```
  M    :: ((a -> Void) -> Void) -> M a
  runM :: M a -> ((a -> Void) -> Void)
```

この `M` モナドの中では、二重否定の除去はいとも簡単に書ける。

```
removeDoubleNeg :: ((a -> Void) -> Void) -> M a
removeDoubleNeg = M
```

このモナドのコンストラクタでくるむだけで OK だ。

継続機能を持つモナドトランスフォーマも、まったく同様にして実装できる。以下の `T` は、引数

のモナド m に継続を扱う機能を付加するモナドトランスフォーマだ。

```
data T m a = T { runT :: (a -> m Void) -> m Void}
instance Monad (T m) where
  return a0 = T ($ a0)
  m >>= k   = T $ \ c -> runT m (\ x -> runT (k x) c)
```

モナドの bind (>>=) の実装を見ると、m >>= k と書いたときに、m には「x を引数にとって k を実行する関数」が渡されることがよくわかる。主導権は m 側にあり、m は k の入った継続をまったく呼ばないのも、複数回呼び出すのも意のままだ。これが、少女の人生を弄ぶ悪魔の力のみなもとなのだ。

ところで、これらのモナドで記述した計算を実際に実行するには、「二重否定」の文脈から抜け出して (モナドトランスフォーマを外して) 中身のモナドを取り出す方法がなければならない。これは、どうすればいいのだろうか。

二重否定の文脈 (a -> Void) -> Void を通常の値 a に変換するには、(a -> Void) 型の関数を引数に渡してやればよい。モナドトランスフォーマ版の場合には (a -> m Void) 型だ。これらの関数は、最終的な継続を受け取って生の値を返すから、終端関数とでも呼ぶことにしようか。

実は、Void を直接利用した実装には欠点がある。それは、終端関数そのものは Void 型の値を利用しないと実装できない、ということだ。今回の実装では、次のようにして a -> m Void 型の関数 f を undefined から供給してやることで、計算を成り立たせることにした。なあに、心配することはない。将来的にこの実装を Haskell モジュールにまとめるにあたって、値コンストラクタ T とレコード関数 runT をエクスポートしないことによって、トランスフォーマ T をカプセル化すれば、この undefined にアクセスする方法はないはずだ。

```
getEffect :: forall a m. (Monad m) => T m a -> m ()
getEffect k = runT k f >> return ()
  where
    f :: a -> m Void
    f _ = return undefined
```

Haskell の継続モナドライブラリ Control.Monad.Cont では、Void の代わりに全称量子化 (forall r.) された型変数 r を使う方法を採用している。これは [1] らの方法に基づいている。こちらは、undefined を使わずに実装できる、という利点がある反面、ユーザーが r を勝手にインスタンス化できてしまうという欠点がある。このために、Control.Monad.Cont の実装 (forall r. (a -> r) -> r) は二重否定に対応するとはいえず、むしろカリーハワード対応においてもとの型 a に対応する命題よりも強力な命題になってしまっている。このため、本章ではカリーハワード対応との整合性を取るため (a -> Void) -> Void の表現を採用した。

さて、この T があれば、次のように call/cc を実装できる。call/cc とは何か？ の解説は他にゆずることにして、これで継続が実装できたことになる。

```
callCC :: ((a -> T m b) -> T m a) -> T m a
callCC f = T $ \ c -> runT (f (\ x -> T $ \ _ -> c x)) c
```

そして、『悪魔の契約』こと、排中律は、call/cc を用いて次のように実装できてしまうのだ。

```
exmid :: T m (Either a (a -> T m b))
exmid = callCC f
```

```
  where
    f k = return (Right (\x -> k (Left x)))
```

2.5　タネもシカケも、ないんだよ！

最後に、これまでの話をまとめて、外部ライブラリに依存せず、Prelude の機能のみで「悪魔の契約」デモプログラムを実装しておく (次ページ参照)。[*2]

このプログラムで、M と名のついたモナドは Writer モナドと Cont モナドを合成したものをハードコードしたものに相当する。main 関数の本文はただ一行の putStrLn で、そこに式 getStory $ offerContract >>= useContract が計算したストーリーの文字列が渡されている。つまり、悪魔との契約の物語—継続を利用する古典論理的な計算が、Haskell のピュアな計算、直観主義論理のみを許す計算の上でエミュレートできているのだ。このエミュレーションは、モナド M に含まれる継続渡しスタイル—二重否定により実現されている。

```
data M a = M { runM :: (a -> (Void, String)) -> (Void, String)}
```

2.6　まとめ

この記事では、通常の感覚では絶対に許されないような、少女の魂を弄ぶ『悪魔の契約』プログラムすら書けるようにしてしまう**継続**と、通常の感覚では認めない人はいないであろう**二重否定の除去**が、じつは等価であることを示した。だのになぜ、二つの概念にはこんなにも差があるように感じられるのか？ この違和感を解決することが、著者の動機であった。

そういえば、結局、二重否定の除去を公理として認めるということは、((a -> Void) -> Void) -> a 型のプログラムの存在を認める、ということだったのだ。((a -> Void) -> Void) なんていう得体の知れないものから、ただの a を作れるとは、それ自体じゅうぶん不思議な力じゃないか。これさえあれば、call/cc が作れるというのも不自然ではない（実際作れる。）

この感触は、直感主義者たちが古典論理に抱いたであろう違和感を反映しているのだろう。私たちは、二重否定を受け入れるならば、『悪魔の契約』の妥当性も受け入れることになるのだ。

参考文献

[1] WADLER, P. Call-by-value is dual to call-by-name. *ACM sigplan notices 38*, 9 (2003), 189–201.

[*2] 本章のプログラムは、ghc 7.10 以降では「M が Monad なのに Applicative にはなってない」旨のエラーが出て実行できない。本来、すべての Monad は Applicative であるべきなので、この仕様変更に従いたいところだが、本文中では冗長になるので Applicative や Functor のインスタンス宣言は割愛した。

```
{-# LANGUAGE ScopedTypeVariables #-}
data Void = Void {absurd :: a -> Void}

data M a = M { runM :: (a -> (Void, String)) -> (Void, String)}

instance Monad M where
  return a0 = M ($ a0)
  m >>= k   = M $ \ c -> runM m (\ x -> runM (k x) c)

tell :: String -> M ()
tell w0 = M $ \k -> let (b1,w1) = k () in (b1, w0++w1)

callCC :: ((a -> M b) -> M a) -> M a
callCC f = M $ \ c -> runM (f (\ x -> M $ \ _ -> c x)) c

offerContract :: M (Either a (a -> M b))
offerContract = callCC f
  where
    f k = return (Right (\x -> k (Left x)))

useContract :: M ()
useContract = do
  tell "Madoka : Watashi Kami Ni Naru\n"
  tell "Homura : Yamete\n"
  tell "Kyubey : QPPY!\n"
  r <- exmid
  case (r :: Either Int (Int -> M Void)) of
   Left x -> do
     tell "Madoka : I am rich: I got USD_"
     tell (show x)
   Right k -> do
     tell "Madoka : I will give kyubey money and will become god.\n"
     v <- k 100
     tell "Madoka : Now I am god! I can do anything!"
     tell (show (absurd v :: Double))

getStory :: forall a. M a -> String
getStory k = snd $ runM k f
  where
    f :: (a -> (Void, String))
    f = (\ _ -> let (v,_) = runM k f in (v,""))

main :: IO ()
main = do
  putStrLn $ getStory $ offerContract >>= useContract
```

第3章

真の名を〜データ設計とAlloy〜

— @osiire

ことばは沈黙に 光は闇に 動は静のなかにこそ あるものなれ
飛翔せるタカの 虚空にこそ 輝ける如くに
– エアの創造より

3.1 プロローグ

3.1.1 困難な課題

ローク島の魔法学院では豊穣の祭りに向けた準備が慌しく進められていた。今年は魔法使いとしても名高いオー島の領主とその美しい妃が招待される予定とあって、院生達はいつになく張り切っている。夕食会のディナーのために特別なワインが揃えられ、広間のセッティングには様式の長までが自ら進んでたいまつの配置から天井の飾りにまで事細かに指示を出していた。一方、学院から離れローク島最北端の岬に建つ隠者の塔では、名付けの長が書いては消す大量の真の名を今日も7人の院生が黙々と書き写し覚えている。その隠者の塔の一室で、ハイタカは大量の書き込みがある縒れたノートを前に一人焦っていた。名付けの長に出された課題に対する作業が遅々として進まなくなっていたからである。

「塔の中にある物の真の名を全て列挙し、関連性を示せ」

そう長から言い渡されたのは、塔からうっすらと見えるまぼろしの森の木々の濃い緑が映えていた頃であった。

「隠者の塔はそう広くない。真の名などすぐに列挙できるだろう。」

未熟な4年生はタカをくくっていた。しかし、実際に課題に取りかかると、その困難さはすぐに分かった。まず、物の真の名を探り当てる事が意外と大変だった。既に名を知っている物は簡単だが、真の名を知らない物については教科書や古代文献を探さねばならない。その上で、調べた真の名を呼んでみてまやかしの術から開放されるかどうかなど、実験して確かめる必要がある。あばかれていない真の名の場合、それを探り当てるのに魔法使いが一生を費やすこともあるくらいである。塔の中にそこまで難しい物ははないとはいえ、探り当てに躓くとそれだけで一週間浪費することもあった。

さらに、名を列挙していく内に、名同士には一対多/多対多などといった関係がある事が分かった。例えば、テーブル (真の名をトーニと言う) には必ず一枚の天板と一本以上の足があるといった風に。これが長が指示していた関連性である。そうなると、いくら狭い塔の中とはいえ示さなければならない関連性の組み合わせは 100 や 200 ではなくなる。関連性の発見と記述はどんどん膨大な物になっていき、提出予定のノートは関連性の線が幾重にも交差して黒くなる一方だった。

この気の遠くなるような課題、提出期限は冬至までである。もう残された時間は少ない。こんな

課題ができないようではヒスイにも勝てない！ カラスノエンドウは今頃どうしているであろうか。どうにかこの状況を打開できないか。ハイタカの頭の中では、何度も同じ言葉が浮かんでは消えていた…。

3.1.2 師の教え

海から吹く風がずいぶん寒くなってきたある日、食堂で遅い昼食を食べ終えたハイタカは塔への戻り道の途中の道端に、タラクサカムの一種が生えているのを偶然見付けた。

「珍しい。オジオンに教えてもらったことがある、薬草としても使えるカントウタラクサカムだ。こんなところでも自生しているのか。」

その黄色い小さな花と幾重にも重なったギザギザの葉を膝をついて眺めながら、彼はゴンド山で師と過ごした日々を思い出していた。オジオンは山中の草木の効能や名前 (真の名も) を丁寧に教えてくれたものだった。

師は偉大な魔法使いだ。しかし、さすがのオジオンでも記憶力に限界があるはず。どうやってあんな大量の名を覚えていたんだろう。老いた魔法使いの後ろ姿を想像しながら、そんな疑問がふと浮かんだ瞬間、思い出した。

「そうだ、Alloy だ！ オジオンは沢山の草木の名を Alloy Analzyer で整理していた。あれを使えば塔の名もきっとうまく整理できるに違いない！」

思わず立ち上がって叫んでいた。ハイタカは早速、今しがた出てきたばかりの食堂の横を抜け、隠者の棟に比べると一際明るい電算棟に駆け込んだ。

3.2 Alloy の基礎

3.2.1 Alloy とは

電算棟の自習室には 10 台近くの平机が所狭しと並べられ、平机一つにディスプレイと端末一つが配置されていた。それぞれの机の下ではゴミ箱のような大きな端末が唸りをあげ、部屋の天井に這っている黄色いケーブルと何かを送受信していた。中には”Macintosh”と書かれたディスプレイ一体型の小型端末もあったが、彼は使い方がよく分からなかったので近寄らないことにしていた。ハイタカは、大きく”X”と描かれた画面の前に座り、真の名を入力してログインした。

「Ged」

ネットの情報によると、Alloy Analyzer [3](`http://alloy.mit.edu/alloy/`) とは、Daniel Jackson らによって発明された有界モデル発見器というものの一種らしかった。

- Alloy Analyzer は Z に似た記法の言語を持ち、物 (エンティティ) 同士の関係 (Relation) をベースに仕様 (モデル) を表現できる。
- 記述されたモデルから、モデルを満たす例 (インスタンス) を SAT ソルバーを用いてスモールスコープ内で全探索し、図まで示してくれる。(スモールスコープとは、記述された複数のエンティティの数をそれぞれ 0 から一定の数 N まで変化させた時に得られるインスタンスの集合の事。一定の数 N が実質的に 10 程度と大きくないので「スモール」スコープと呼ばれる。)
- Java 製のオープンソースソフトウェア (`http://alloy.mit.edu/alloy/download.html`) であり、Java の実行環境があれば誰でもすぐに利用できる。

これらの性質を持つ Alloy Analyzer は [3] でも述べられている通り、所謂「データ設計」に応用できるようだ。データ設計では物 (エンティティ) やその属性に名前を与えてその構造を決める。関係の記述が得意である Alloy Analyzer は、これらの構造を簡潔かつ正確に記述できるという訳だ。しかも、Alloy Analyzer が示してくれるインスタンスや無矛盾性チェックは、設計をセルフチェックするために利用できる。

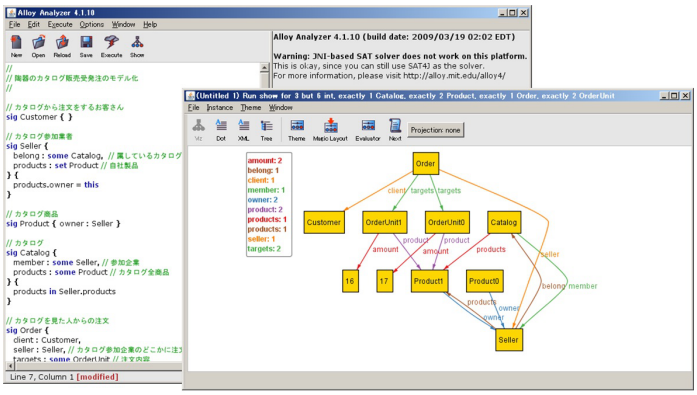


図 3.1: Alloy Analyzer と表示されるインスタンス図の例

「これは長の課題にぴったりだ。」
ハイタカは twitter に呟こうとしたが、まだ我慢した。

3.2.2 基礎構文と意味

まずは Alloy を使いこなせるようにならなければならない。書籍 [4](日本語版もある `https://sites.google.com/site/softwareabstractionsja/`) に詳しい解説があった。文法のチートシート `https://docs.google.com/file/d/0BzmwV-bnsTo4cEFIM1Fodkc4dVU/edit?pli=1` もネット上で発見した。それらによると、まず物 (エンティティ) を定義するには、シグネチャー (`signature`) を使うらしい。

```
sig テーブル {}  //シグネチャーの定義
run {}
```

(ここでは便宜上日本語でシグネチャーを定義するが、本当は真の名で記述されている) これでテーブルという名前の定義になる。もっとも簡単なモデルである。このコードを Alloy Analyzer の画面上部にある `Execute` ボタンで実行すると `run {}` が走り、スモールスコープ内でのインスタンス検索が行われる。`instance found` と画面右側に出れば成功だ。(図 3.2) 画面上部にある `Show` ボタンを押すと、インスタンス例が図示される。

図 3.2: Execute ボタンで実行. Instance found で OK.

モデルを拡張してみよう。テーブルには天板と足がある。これを属性 (Alloy では**フィールド**と呼ぶ) として加えると次のように書く。(正確な文法については、[2] を参照の事)

```
sig 天板，足 {}
sig テーブル {
  上部: one 天板，
  下部: some 足
}
```

one と some は多重度の明示で、それぞれ 1 個、1 個以上を意味する (多重度について詳しくは後述する)。上記のモデルは、テーブルには 1 個の天板と 1 個以上の足があるという意味になる。多重度は省略すると one を記述したのと同じになる。

上部, 下部はフィールドの名前であるが、それ自体が二項関係になっている。上部はテーブルと天板の二項関係、下部はテーブルと足の二項関係である。(ここで二項関係とは、集合 A,B の直積の部分集合である。例えば「上部」という二項関係はテーブル集合と天板集合の直積の部分集合である。) 上部と下部の例を示しておく。

- 上部のインスタンス例: { (テーブル 1, 天板 1) (テーブル 2, 天板 2) }
- 下部のインスタンス例: { (テーブル 1, 足 1) (テーブル 1, 足 2) (テーブル 2, 足 2) }

シグネチャー以外にも enum によって列挙子が作れるようだ。

```
enum 材料 { 木，プラスティック，金属 }
```

材料の種類には、木、プラスティック、金属があるという意味のモデルとなる。これは次のコードと等価である。

```
abstract sig 材料 {}
one sig 木，プラスティック，金属 extends 材料 {}
```

abstract sig はインスタンスの存在しないシグネチャーの宣言。カテゴライズのために使われる。one sig の one は該当シグネチャーのインスタンスは常に 1 個という意味である。extends は継承を示すキーワードで、木、プラスティック、金属が材料の要素である事を意味する。

とりあえず、これだけ分かれば簡単なデータ設計は進められそうだ。

3.3 Alloy による概念データ設計

ツールとして Alloy を使うとしても、概念データ設計の考え方自体は通常のものと何ら変わりない。基本的には、エンティティと属性を洗い出して定義し、その関係と制約を記述していく作業だ。

- データを構成していると思わる概念を整理し、属性を洗い出す。
- データ間に存在する共通部分に着目し、新しい名前をつけて抽象化する。単に共通部分を括り出すのではなく、共通部分が抽象化した概念の具体例となる事が前提。
- データの整合性を保ちやすくするために、構造を工夫して不変条件を少なくする。正規化や代数的データ型化を行う。
- リソース系/イベント系という観点でデータを区分けしておくと後工程がわかりやすくなる。

まず、データ設計でよく用いられる ER 図 (IDEF1X) で決まっている作法を Alloy で模倣してみよう。

3.3.1 多重度の表現

ER 図ではエンティティ同士の多重度を定義する。これは概念データ設計でも重要な作業だ。1 個、0 個または 1 個、0 個以上、1 個以上、という 4 種類の多重度をよく使う。Alloy でこれらを表現するにはそれぞれ、one, lone, set, some という多重度指定を用いる。

- one: 1 個
- lone: 0 個または 1 個
- set: 0 個以上
- some: 1 個以上

例えば、テーブルには 1 本以上の足があり 0 個以上の飾りが付いている事を表現するためには、次のようにする。

```
sig 天板，足，飾り {}
sig テーブル {
  上部: one 天板,
  下部: some 足,  // 1 個以上の足
  装飾: set 飾り  // 0 個以上の飾り
}
```

ここで留意しなければならないことが一点ある。このモデルでは、テーブルと足、テーブルと飾りはそれぞれ多対多の関係になっている点である。なぜなら、例えば「装飾」はテーブルと飾りの直積集合の部分集合であり、それ以外の制約はない。したがって、同一の飾りが同時に他のテーブルの装飾にもなり得るのである。

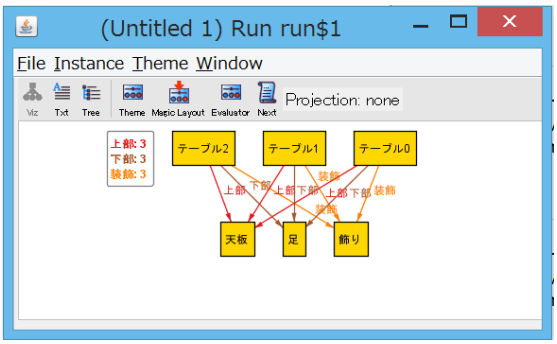


図 3.3: 同一の飾りが複数のテーブルの装飾になっているインスタンス例

もし、特定の飾りは同時に複数のテーブルに飾られないという制約を与えたい場合、モデルに次のような制約を加える。

```
sig 天板，足，飾り {}
sig テーブル {
  上部: one 天板,
  下部: some 足,  // 1 個以上の足
  装飾: set 飾り  // 0 個以上の飾り
}
```

```
fact {
  all x:飾り | lone 装飾.x // 同時に複数のテーブルには飾られない。
}
```

ここで fact という無条件で成り立つ制約を記述する機能を使っている。all x:**飾り**の部分は「全ての飾り x について」という意味であり、x を変数として束縛している。右側の**装飾**.x の部分にあるドットは、結合演算子と呼ばれるものである。左側の関係の最右項目と右側の関係の最左項目の共通部分を重ね、該当共通部分を消し去ったあとの関係を返す演算である。言葉では分かりにくいので、結合演算の例を示す。

- 例 1: { (テーブル 1, 飾り 1) (テーブル 1, 飾り 2) } . { 飾り 1 } ⇒ { テーブル 1 }
- 例 2: { (テーブル 1, 飾り 1) (テーブル 1, 飾り 2) (テーブル 2, 飾り 3) } . { 飾り 1 飾り 3 } ⇒ { テーブル 1 テーブル 2 }

結合演算の結果、**装飾**.x はテーブルの集合になる。その集合が lone であると制約しているので、特定の飾りが対応するテーブルは 0 個または 1 個だけという意味になる。0 個であれば飾られていない飾り、1 個であれば特定のテーブルに飾られている飾りになる。このように制約すると、ER 図で言うところの一対多の関係になる。

なお、fact による制約を使わずとも飾りシグネチャーへの制約として下記のようにも書ける。

```
sig 天板，足 {}
sig テーブル {
  上部: one 天板,
  下部: some 足， // 1 個以上の足
  装飾: set 飾り  // 0 個以上の飾り
}
sig 飾り {} { lone 装飾.this } // 飾りは同時に複数のテーブルには飾られない。
```

3.3.2　依存性の表現

一対多の関係においては、一方のエンティティは他方のエンティティが存在しないと存在できない場合が多い。例えば、テーブルが存在しないと対応する足は存在せず、どのテーブルにも属さない宙ぶらりんの「テーブルの足」は存在しない。このような関係は、一対多かつ依存の関係になる。一対多かつ依存の関係の場合には、前述の 1 対多の制約において lone とされていた部分を one に変更すればよい。

```
sig 天板，足，飾り {}
sig テーブル {
  上部: one 天板,
  下部: some 足， // 1 個以上の足
  装飾: set 飾り  // 0 個以上の飾り
}
fact {
  all x:飾り | lone 装飾.x // 同時に複数のテーブルには飾られない。
  all x:足 | one 下部.x // 足はテーブルに一対多で依存.
}
```

下部.x はテーブルの集合である。その集合が one であると制約しているので、特定の足が対応するテーブルは 1 個だけかつ 1 個は必ず存在するという意味になる。

3.3.3 対称的な多対多の関係

ユーザーが複数の部署に所属し、部署には複数のメンバーがいるというよくある多対多のモデルを記述してみる。多対多の関係を作るには set か some という多重度を用いればいいことは前述した。

```
sig ユーザー {
  所属: set 部署,
}
sig 部署 {
  メンバー: set ユーザー,
}
```

ところが、このモデルのインスタンス例を Alloy で追っていくと、下記のような例 (図 3.4) が得られる。

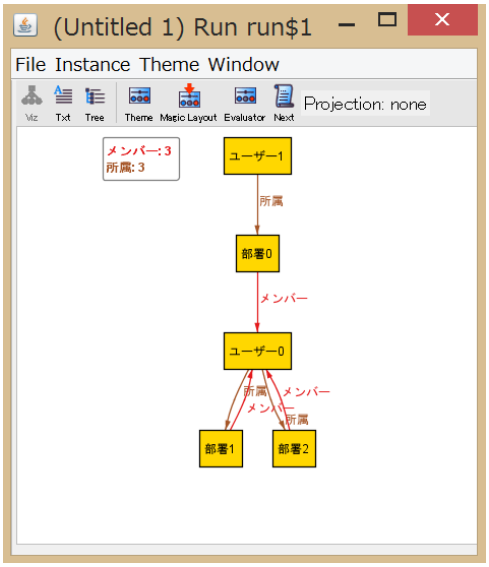


図 3.4: 非対象的な多対多関係のインスタンス例

特定のユーザーがある部署に所属しているのに、部署からはメンバーと認識されていない場合が見て取れる。これでは所属とメンバーという二項関係が対称的な関係になっていない。そこで二つの関係を対称的とするために、メンバーの要素を反転させると所属に等しくなるという制約を加える。二項関係の転置演算子~を使うと、次にように書ける。

```
sig ユーザー {
  所属: set 部署,
}
sig 部署 {
```

```
  メンバー: set ユーザー,
}
fact { 所属 = ~メンバー } // メンバーは反転させると所属と同一であるという制約
```

これで意図した対称的な多対多の関係となる。

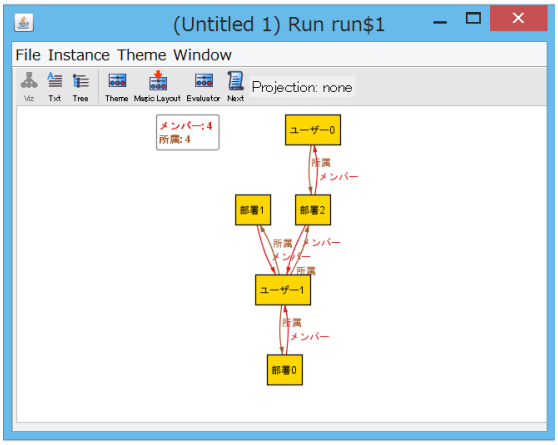


図 3.5: 対象的な多対多関係のインスタンス例

なお、物理データ設計でよく行われるように多対多の関係そのものをエンティティ化する方法もあるが、Alloy ではインスタンス例が見づらくなるので (概念データ設計の段階では) 避けた方が良い。

3.3.4　一意性の表現

ER 図ではどのカラムを組み合わせるとテーブル内で行が特定できるか (一意性) を候補キーとして列挙して表現する。これは、正規化を意識する上でも必要な作業である。Alloy でシグネチャーインスタンスの一意性を示したい場合は、候補キーが同じなら同一のインスタンスであるという制約を加えればいい。例えば、病院では患者を名前と生年月日と性別で区別している場合があるが、これを表現してみよう。

```
sig 文字列, 日付 {}
enum 男女 { 男, 女 }
sig 患者 {
  名前: 文字列,
  生年月日: 日付,
  性別: 男女,
  住所: 文字列 // 住所だけ候補キーではない.
}
fact {
  // 名前、生年月日、性別が同じなら同一人物
  all x,y:患者 |
    x.名前 = y.名前 && x.生年月日 = y.生年月日 && x.性別 = y.性別 => x = y
}
```

なお、記号=> は「ならば」を意味している。

3.3.5 直和/継承の表現

一般的に ER 図で表現できる関係は、多重度、依存性、一意性の 3 つである。ここまでで、これらは全て Alloy でも模倣できた。一方で、概念データ設計をするにはこれだけでは足りない。概念とデータを整理する過程で、直和 (代数的データ型) や継承も使いたくなるのは自然な流れである。Alloy ではこれらも表現することができる。

例えば、トランプのカードをモデル化してみよう。ここでトランプのカードは 4 種類のマークと数字、ジョーカーを含むものとする。

```
enum スート { スペード，ハート，クラブ，ダイヤ } // 4 種類のマーク
abstract sig カード {} // カードという物
sig ジョーカー extends カード {} // ジョーカーはカードの一種.
sig 数字カード extends カード { // カードはマークと数字を持つ
  マーク: スート,
  数字: Int
}
```

カードはカードという概念をカテゴライズするシグネチャーである。abstract なのでインスタンスはない。**ジョーカー**と**数字カード**は**カード**を継承しており、**カード**の一種であることを意味している。同一のシグネチャーを extends するとそれらは互いに素に分かれるようになっており、この意味で**カード**は**ジョーカー**と**数字カード**の直和になっている。このような構造は頻繁に現れるので、パターンとして覚えておくと便利である。

もう一つ、多重継承の例を見てみよう。例えば、牛乳とコーヒーがあり混ぜればコーヒー牛乳になる関係を表現してみる。実は Alloy は多重継承を実現する次のような直感的な書き方は出来ない。

```
sig 牛乳，コーヒー {}
sig コーヒー牛乳 extends 牛乳，コーヒー {} // このような書き方はできない
```

このような事ができてしまうと、extends したシグネチャーが互いに素になるという原則が崩れるためである。その代わり、Alloy では in キーワードを使った部分集合シグネチャーを定義でき、その機能で多重継承を表現する。[1]

```
abstract sig 飲み物 {} // まず土台を作る
sig 牛乳 extends 飲み物 {} // 牛乳は飲み物の一種である
sig コーヒー in 飲み物 {} // 飲み物のカテゴライズ (部分集合シグネチャー)
sig コーヒー牛乳 extends 牛乳 {} // コーヒー牛乳は牛乳の派生物
fact {コーヒー牛乳 = コーヒー } // コーヒー牛乳はコーヒーの一種でもある.
```

かなり面倒だが上記コードではコーヒー牛乳はコーヒーのフィールドと牛乳のフィールドの両方を持ったシグネチャーになる。コーヒーは飲み物をカテゴライズする部分集合シグネチャーと呼ばれるもので、コーヒーというインスタンスは作らないが、飲み物にコーヒーのフィールドを与えることができるシグネチャーとなる。まず、コーヒー牛乳は牛乳を extends して牛乳の一種としておき、その後 fact でコーヒー集合と一致する制約を入れている。コーヒー牛乳はコーヒー枠に等しく、その中にあるの牛乳の一部がコーヒー牛乳になるイメージである。(図 3.6)

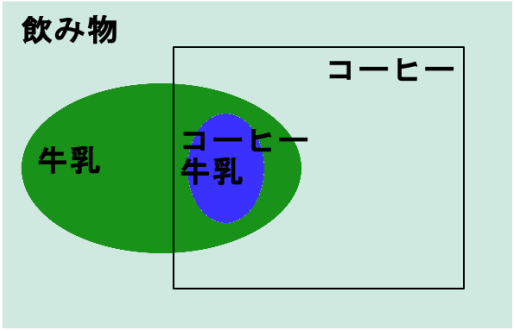


図 3.6: 多重継承は部分集合シグネチャーで表現する

3.3.6　制約の記述

データ構造には暗黙の条件が多々あり、従来多くはそれらをコメントに残すだけであった。自然言語で書かれたコメントは非形式的で曖昧さが残り、時には矛盾した条件さえ記述されている。しかし、Alloy でデータ設計及び不変条件を記述すれば、設計段階でそれら制約の記述可能性や (スモールスコープ内での) 無矛盾性をセルフチェックでき、制約意図を正確に他人に伝えられるという利点がある。ここが Alloy でデータ設計する真骨頂とも言える。

例えば、某システムで実際に記載した制約 (少しわかりやすくしたもの) を挙げてみる。

```
sig 取引成立 {
  買い注文, 売り注文: some 注文
} {
  // 対向に同じ発注元はない.
  disj[売り注文.発注元, 買り注文.発注元]

  // 売買方向の制約
  売り注文.売買 in 売り
  買い注文.売買 in 買い

  // 全ての取引条件は合致.(合致という述語は別途定義している)
  all x:売り注文.取引条件, y:買い注文.取引条件 | 合致[x,y]

  // 注文数量がバランスしている
  sum(売り注文.注文数) =  sum(買い注文.注文数)
}
```

このデータはある種のオークションシステムにおいて注文同士の条件が合致し、取引が成立したことを示すイベント系データである。取引が成立するためには種々の条件があり、自己売買の禁止から各種取引条件の合致が求められる。そこでそれらを具体的に記載したところ、不足していた項目が明らかになったり、構造を変えた方がよい点が見つかるなどした。単なるコメントの記載では不十分であることが実感できた例である。

3.3.7 その他の留意点

Alloy Analyzer は、概念データ設計において表現したい構造をシンプルかつ強力に表現できるツールであることが分かった。また、ソルバーとインスタンスの図示によって設計作業を手助けしてくれる。最後に Alloy を概念データ設計実務に応用する際の留意点を挙げておく。

- インスタンス検索範囲が小さいとモデルに矛盾がなくても解が見つからないことがある。デフォルトの `for 4` では小さいので、`for 7` 程度にすればシグネチャー数が多くてもまず大丈夫である。
- 関係の名前は衝突すると曖昧だという理由で Alloy に拒否される。どの関係かを指定するには、定義域指定か値域指定をする必要があるので留意。そもそも衝突しないように接頭語を付けるなどした方が便利な時も多い。
- モデルが巨大になってくるとインスタント例図が複雑になりすぎるが、射影によって表示シグネチャーを絞る事で解決できる。しかし、表示シグネチャーを選ぶプルダウンがスクロールしないので、シグネチャー数が一定以上になると画面からはみ出して選択できない。そのうち修正版を作りたい。
- インスタンスを 1 個に絞る、`lone` も強制的に `one` にする、不要なインスタンスを射影すると、疑似 ER 図ぽっいものを図示できる。

3.4 概念モデルから RDB テーブルへ

概念モデルの作成では、処理スピードやメモリ量といったリソースの制約については考慮せず、純粋にデータ構造を検討した。一方、プログラミングにおけるデータ構造や RDB テーブルを具体的に決める場合には、リソースの節約、処理スピードのための構造の導入、RDB ならではの特殊事情などを追加で加味して概念モデルをベースに再検討する必要がある。Alloy で作成した概念モデルをプログラミング用データ構造や RDB に発展させる際に、ポイントとなる点を挙げておく。結論としては、RDB のテーブル設計をするなら Alloy より ER 図を作成する専用のエディタを利用した方がよい。

3.4.1 直和の展開

直和 (代数的データ型) は、データ構造から不変条件を明示的に減らす事ができるので概念データモデルでは積極的に利用したい。しかし、残念ながらこれを直接表現できないプログラミング言語があり、また RDB では全く表現できない。そこで、直和構造は別の不変条件の多い構造に変換しなければならない。先のカードの直和の例を RDB テーブルに直すなら、カードの種類を表現するカラムと数字のカラムを併記し、数字のカラムは null が入ることを想定する。

```
create table Card {
  -- 1=スペード，2=ハート，3=クラブ，4=ダイヤ，5=ジョーカー
  cardType int not null,
  cardNumber int -- 数字．cardType がジョーカーの場合は null になる．
}
```

3.4.2 レコードの参照方法

テーブル内の特定の行を指し示すために、サロゲートキーを用いるか候補キーを列挙して対応するか、アプリケーション内のでそのテーブルへのアクセス方法とそのスピードも考慮して決定しなければならない。(概念モデルの時には単に特定のエンティティを指し示すだけでよかった。)

3.4.3　1 対 N の表現方法

1 対多になる関係は、多側のテーブルに 1 側のテーブルを一意に参照するキーを追加する。逆に 1 側のテーブルからは該当項目が消えるので、字面だけでは構造が追いにくくなる。Alloy でこの構造を書いてもキー（数字）との関係が張られるだけで、1 対多の構造はやはり見えない。ER 図を書けば関係は一目瞭然である。

3.4.4　経年変化の考慮

イベント系テーブルはそのデータの経年変化を防ぐために、リソース系テーブルへの参照を切らなければならない。このために、正規形を崩しリソース系テーブルのデータをコピーする構造へ変更する。

3.4.5　結合テーブルの削減

SQL 文で結合するテーブルが多くなりすぎたり特定のテーブルへのアクセスが集中したりすると、検索パフォーマンスが劣化してしまう。実際、継承を含むオブジェクトモデルをそのまま RDB に落とした某大型システムのパフォーマンスがこのせいでひどいことになった事例を知っている。そこで、抽象化の過程で共通部分として括り出していたエンティティは、正規化に影響しないのであれば参照元に埋め込んだ方がよい場合がある。

3.5　エピローグ

冬至の日、ハイタカは作成したモデルを手に、塔の一番上にある名付けの長の部屋を訪ねた。長の部屋は優雅なクラシック音楽が常に流れ、フランス製の高そうなワインが並び、大量の本と論文が雑然と積み上げられている中、なぜか四隅に熱帯植物が飾られている。学生の間では「熱帯のフランス」と呼ばれている場所である。ハイタカはその部屋の真ん中に座り、長からの評価を待っている。ハイタカから提出されたコードを一通り眺めた名付けの長は、くるりと椅子を回し彼の方を向いて少々驚いたような顔をした。

「真の名の列挙もできその関連性も整理されている。Alloy Anazlyer をよく知っていたな。」

「オジオンが使っていたので思い出しました。」

「そうか、君は彼の大魔法使いの弟子だったな…」

ハイタカは得意気に説明を加えだした。

「データ設計でよく利用される ER 図作成においてポイントとなるのは、多重度、依存、一意性の表現です。これは Alloy 言語でもシンプルに表現できました。加えて、直和や継承といった構造も表現できます。インスタンスの図示はデータ構造の試行錯誤にも有用で、Alloy Analzyer は十分実用になるツールだと感じました。」

「さらに、不変条件も記述して形式的にチェックできた事は、データ構造の妥当性の確認に有用でした。これはクラス図作成では得られない重要な利点だと思われます。」

いつになく饒舌な彼は Alloy Analzyer を使っていた師を自慢しているようでもあった。

「うむ、確かに。ただ、そこまでは良いが、Alloy 言語は普及しているとは言い難い。Alloy 言語によるモデリングコードがそのまま実装工程にも引き継がれるなら理想だが、実際には Alloy 言語への理解を実装工程の全員に強制することは難しい。そういう場合はどうすれば良いか？」

名付けの長は現実主義者である。ハイタカは回答に困ったがこう言うしかなかった。

「作成したモデルのドキュメントへの変換など、別途作業が必要になってしまう点は今後の課題です。」

名付けの長は微笑みながら言った。

「なかなかよいすべり出しじゃな。」

その夜、「Alloy Analyzer 最強」とハイタカが twitter に書き込んだことで一騒動起こるのだが、そ

れはまた別の話である。

参考文献

[1] DENNIS, G., AND SEATER, R. Alloy Language and Analysis. `http://www.sts.tu-harburg.de/teaching/ws-08.09/VSS/04b-Alloy_language.pdf`. [Online; accessed 11-July-2015].

[2] JACKSON, D. Core Alloy4 Syntax. `http://alloy.mit.edu/alloy/documentation/alloy4-grammar.txt`. [Online; accessed 11-July-2015].

[3] JACKSON, D. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol. 11*, 2 (2002), 256–290.

[4] JACKSON, D. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

ネタ元

この記事の執筆にあたり、「ゲド戦記」から人物や設定を拝借いたしました。著者のアーシュラ・K・ル＝グウィンに敬意を表します。

第4章

Lagrangeの未定乗数法

— @dif_engine

物語の概要と目的

現実世界における「滑らかな問題」の最適解の候補を見出すための手法として応用上重要な Lagrange の未定乗数法について学びます。この記事では線形代数や微積分の復習をしながら、ゆったりとしたペースでこの手法について学びます。

4.1 プロローグ

魔理沙が遊びに来ている。一時期はしきりに Haskell についての質問をしてきたが、最近は学習理論に興味があるらしい。

「—— でさ、Lagrange の未定乗数法ってのがでてきてさ」

「……ええ、学習理論の本を読んでいると、分布の最尤パラメータを求める場合に Lagrange の未定乗数法を使ってる事が多いわね」

「そうそう、でもあれって一体なんでうまくいくのかよくわからないんだよな」

「変数が少ない場合の話はわかるかしら？」

「等高線と拘束条件の曲線が接してるから……みたいな絵で説明してるのは見たことあるぜ。二変数のときはあれで一応わかったことにしてるんだけどさ」

「次元がもっと高い場合の話がわからない……？」

「そうそう！　高次元でも同じだよ、とか言われても高次元はうまく思い浮かべられないぜ」

「高次元への直観が働かないから、高次元への直観を前提とした説明も理解できず、ついには Lagrange の未定乗数法自体が疑わしくなってしまった……という感じかしら」

「それな。パチュリーはやっぱすごいな」

「おだてたって何もでないわよ。さて、では Lagrange の未定乗数法の確認をしておきましょう。でもその前に最低限の記号の整理をしておきましょう ——。」

本記事での約束 (1)

- 自然数 M, N は $1 \leq M < N$ という関係を満たすものとします。N 次元ユークリッド空間 $\mathbb{R}^N$ をしばしば直和 $\mathbb{R}^M \oplus \mathbb{R}^{N-M}$ と同一視します。この同一視に従って、$x \in \mathbb{R}^M$ と $y \in \mathbb{R}^{N-M}$ の対 $\langle x, y \rangle$ を $\mathbb{R}^N$ の元とみなします。
- $\mathbb{R}^N$ の部分集合 Ω 上で定義された関数 F を、直積 $\mathbb{R}^M \times \mathbb{R}^{N-M}$ 上の関数に置き換えたものを $\overline{F}$ で表すことにします：

$$\overline{F}(x, y) := \begin{cases} F(\langle x, y \rangle) & \langle x, y \rangle \in \Omega, \\ \varnothing & \text{otherwise.} \end{cases}$$

- 自然数 $K(\geq 1)$ に対して、
$$\mathscr{O}_K := \left\{ \mathbb{R}^K \text{ の開集合 } \right\}.$$
- 点 $x \in \mathbb{R}^K$ に対して、
$$\mathscr{O}_K(x) := \left\{ P \in \mathscr{O}_K \,\middle|\, x \in P \right\}.$$
つまり、$\mathscr{O}_K(x)$ は点 x の開近傍全体からなる集合です。
- i 行目で j 列目の要素が $f(i,j)$ として与えられた $M \times N$ 行列を
$$\left[\begin{array}{c|l} f(i,j) & \begin{array}{l} i : 1 \downarrow M \\ j : 1 \rightarrow N \end{array} \end{array} \right]$$
とも書くことにします。
- 一列しかない行列は
$$[\ f(i)\ \mid\ i : 1 \downarrow M\ ]$$
のように書くことにします。
- 一行しかない行列は
$$[\ f(i)\ \mid\ i : 1 \rightarrow M\ ]$$
のように書くことにします。

「この行列の表記はあまり見たことないな」
「成分の添字のどっちが行でどっちが列だか混乱しそうな場合にはこの記法は便利よ」
「なるほどな……！」

4.2 Lagrange の未定乗数法

「——では Lagrange の未定乗数法のステートメントを書き出してみるわ」

Lagrange の未定乗数法

前提：

- $\Omega :\in \mathscr{O}_N$
- $f, g_1, \ldots, g_M :\ \in C^1(\Omega;\mathbb{R})$
- $V := \left\{ x \in \Omega \,\middle|\, {}^{\forall} i \in \{1,\ldots,M\}\ \ \left(g_i(x) = 0\right) \right\}$
- $z_0 :\in V$

主張：**(Lag1)** かつ **(Lag2)** のとき **(Lag3)** かつ **(Lag4)**.
ただし、

(Lag1) $\quad {}^{\forall} z \in V \quad \operatorname{rank}\left(\left[\begin{array}{c|l} \dfrac{\partial g_i}{\partial z_j}(z) & \begin{array}{l} i : 1 \downarrow M \\ j : 1 \rightarrow N \end{array} \end{array} \right] \right) = M.$

(Lag2) $\quad z_0$ が $f\big|_V$ の極値点である.

(Lag3) $\quad {}^{\exists}\lambda_1, \ldots, \lambda_M \in \mathbb{R} \quad f'(z_0) = \displaystyle\sum_{k=1}^{M} \lambda_k\ g_k{}'(z_0),$

ただし
$$f'(z_0) = \left[\begin{array}{c|l} \dfrac{\partial f}{\partial z_i} & i : 1 \rightarrow N \end{array} \right], \text{ etc.}$$

(Lag4) $\forall i \in \{1,\ldots,M\} \quad g_i(z_0) = 0.$

「Lagrange の未定乗数法ってこんなステートメントだったっけ？　なんか目的関数に変数を増やして微分して……みたいな感じの話だったような……？」

「ええ、“ラグランジュ関数” を一度導入してから、その微分に対する条件として記述するスタイルもあるわね。条件 **(Lag3)** をそのスタイルで記述するためには “ラグランジュ関数” F を

$$F(z_1,\ldots,z_N,\lambda_1,\ldots,\lambda_M) := f(z_1,\ldots,z_N) - \sum_{k=1}^{M} \lambda_k \ g_k(z_1,\ldots,z_N)$$

として定義してやってから

(Lag3') $\dfrac{\partial F}{\partial z_i}\big((z_0)_1,\ldots,(z_0)_N,\lambda_1,\ldots,\lambda_M\big) = 0 \qquad (i = 1,\ldots,N).$

(Lag4') $\dfrac{\partial F}{\partial \lambda_j}\big((z_0)_1,\ldots,(z_0)_N,\lambda_1,\ldots,\lambda_M\big) = 0 \qquad (j = 1,\ldots,M).$

とすればいいわ」

「ええと、**(Lag3')** は条件 **(Lag3)** と同じことになるな。条件 **(Lag4')** は条件 **(Lag4)** と同じく制約条件を表してるのか。ラグランジュ関数を使っても使わなくても言ってることは一緒だな」

「その通り。ラグランジュ関数を使うと制約条件も一つの関数にエンコードできるので、その意味では便利だけどね」

「理解できなくて困ってるのは条件 **(Lag3')** の形がどっから出てくるかだったけど、それと等価な条件 **(Lag3)** でも『わからなさ』の意味では変わらないな」

「さて、ところで条件 **(Lag1)** に rank が出てきてるわね」

「えーと……行列の階数……だっけか」

「定義は言えるかしら？」

「んー……なんかの個数……だっけか？」

「教科書によって定義は違うけど、一つの定義例は『行列を線形写像とみなした場合の像の次元』ね。つまり、空間全体を行列 A で写したとき、写した先の空間 $\mathrm{Im}A$ の次元が $\mathrm{rank}A$ になるわ：」

$$\mathrm{rank}A = \dim(\mathrm{Im}A).$$

「なんか行列から要素を抜き出して並べて、みたいなやり方もなかったっけ？」

「そうね、小行列式を使って行列の階数を定義することもあるわね。たとえばこんな感じね：

$$\mathrm{rank}A = \max\left\{ n \,\middle|\, {}^{\exists}i_1,\ldots,i_n \ {}^{\exists}j_1,\ldots,j_n \quad \det\left(\left[\begin{array}{c|c} a_{i_p,j_q} & \begin{array}{l} p:1 \downarrow n \\ q:1 \to n \end{array} \end{array}\right]\right) \neq 0 \right\}.$$

それはともかく —— こうして話していて気づいたんだけど、線形代数は苦手みたいね？」

「確かに苦手だぜ！」

なるほど、やはりこのまま Lagrange の未定乗数法の説明を続けても消化できないかもしれない。

「少し回り道になるけど、連立一次方程式のあたりの復習と整理をしておいたほうが良さそうね」

4.3 連立一次方程式

「—— 連立一次方程式は、たくさんの一次方程式を並べたものだわ：」

$$\left.\begin{array}{l} a_{11}\cdot x_1 + a_{12}\cdot x_2 + \cdots + a_{1N}\cdot x_N = b_1, \\ a_{21}\cdot x_1 + a_{22}\cdot x_2 + \cdots + a_{2N}\cdot x_N = b_2, \\ \quad \cdots\cdots\cdots \\ a_{M1}\cdot x_1 + a_{M2}\cdot x_2 + \cdots + a_{MN}\cdot x_N = b_M. \end{array}\right\} \tag{4.1}$$

「今更なんだけど、なんで連立一次方程式が大事なんだ？」

「それは本当に今更な話ね……。連立一次方程式は、数学の色々な問題で出現するわ。だから連立一次方程式について知識を深めておけば数学の色々な問題を攻略しやすくなるのよ」

「それは今の Lagrange の未定乗数法についても当てはまるのか？」

「あとで陰関数定理をつかって Lagrange の未定乗数法を説明するつもりだけど、陰関数定理のステートメントや証明を理解するために連立一次方程式をある程度深く理解しておくと役に立つわ」

「なるほど」

「さて、方程式 (4.1) は行列の積を使えば

$$A \cdot x = b \tag{4.2}$$

のように短く書けるわ。ただし ——

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} \\ a_{21} & a_{22} & \cdots & a_{2N} \\ \vdots & \vdots & & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MN} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_M \end{bmatrix}.$$

とします」

「連立一次方程式をわざわざ行列の積の形に書きなおしたのはどうして？」

「簡潔な書式で記述したかったからよ。簡潔な書式は、複雑な操作の readable な記述を可能にするわ」

「なるほどな」

4.3.1 行列についての記法

「行列を扱う上で便利な記法の復習をしておきましょう。まず、$N_1 \times N_2$ 行列と $N_2 \times N_3$ 行列の積は次のように定義されるんだったわね：」

$$\left[a(i,j) \;\middle|\; \begin{array}{l} i:1 \downarrow N_1 \\ j:1 \rightarrow N_2 \end{array} \right] \cdot \left[b(i,j) \;\middle|\; \begin{array}{l} i:1 \downarrow N_2 \\ j:1 \rightarrow N_3 \end{array} \right] := \left[\sum_{k=1}^{N_2} a(i,k) \cdot b(k,j) \;\middle|\; \begin{array}{l} i:1 \downarrow N_1 \\ j:1 \rightarrow N_3 \end{array} \right]$$

「そうそう、$N_1 \times N_2$ 行列と $N_2 \times N_3$ 行列の積を取ると真ん中の N_2 がキャンセルされて $N_1 \times N_3$ 行列になるんだよな」

「こうして積の定義を眺めてみると、行列の成分は実数でなくても『行列の積』を考えられそうだと思いつくわ」

「複素数や多項式でもいいということ？」

「ええ、実数の代わりに複素数や多項式を入れてもいいわね。でも、行列を入れてもいいのよ」

「行列を成分とする行列……？」

「例えばこんな行列ね：」

$$\left[\begin{array}{cc} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} & \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \\ \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix} & \begin{bmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{bmatrix} \end{array} \right]$$

「こういう『行列の行列』同士で積を考える……ということか？」

「ええ、例えば形式的にこんな計算を行ってみることにすると ——」

$$\left[\begin{array}{cc} \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array}\right] & \left[\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{array}\right] \\ \left[\begin{array}{cc} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{array}\right] & \left[\begin{array}{ccc} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{array}\right] \end{array}\right] \cdot \left[\begin{array}{c} \left[\begin{array}{cc} f_{11} & f_{12} \\ f_{21} & f_{22} \end{array}\right] \\ \left[\begin{array}{cc} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{array}\right] \end{array}\right]$$

$$= \left[\begin{array}{c} \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array}\right] \cdot \left[\begin{array}{cc} f_{11} & f_{12} \\ f_{21} & f_{22} \end{array}\right] + \left[\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{array}\right] \cdot \left[\begin{array}{cc} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{array}\right] \\ \left[\begin{array}{cc} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{array}\right] \cdot \left[\begin{array}{cc} f_{11} & f_{12} \\ f_{21} & f_{22} \end{array}\right] + \left[\begin{array}{ccc} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \end{array}\right] \cdot \left[\begin{array}{cc} g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{array}\right] \end{array}\right]$$

となるわね」

「うん。確かにそうなるな」

「――そして、このこの計算を続行して得られる行列は結局

$$\left[\begin{array}{ccccc} a_{11} & a_{12} & b_{11} & b_{12} & b_{13} \\ a_{21} & a_{22} & b_{21} & b_{22} & b_{23} \\ c_{11} & c_{12} & d_{11} & d_{12} & d_{13} \\ c_{21} & c_{22} & d_{21} & d_{22} & d_{23} \\ c_{31} & c_{32} & d_{31} & d_{32} & d_{33} \end{array}\right] \cdot \left[\begin{array}{cc} f_{11} & f_{12} \\ f_{21} & f_{22} \\ g_{11} & g_{12} \\ g_{21} & g_{22} \\ g_{31} & g_{32} \end{array}\right]$$

を計算したものをブロック分けしたものと同じになるのよ」

「えーと……なるほど、いくつかの成分を実際に計算してみると確かにそうなるな」

「ある程度大きな行列どうしの積を考えるとき、適切なブロック分けを導入すると計算の見通しが良くなる場合があるの。普通の行列であっても『行列を成分とする行列』として考えると便利なことがあるわ。例えば $M \times N$ 行列を『一列の行列』を一行に並べたものだと考える――

$$\left[\begin{array}{cccc} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \vdots & \vdots & & \vdots \\ a_{M1} & a_{M2} & \dots & a_{MN} \end{array}\right] = \left[\left[\begin{array}{c} a_{11} \\ a_{21} \\ \vdots \\ a_{M1} \end{array}\right] \left[\begin{array}{c} a_{12} \\ a_{22} \\ \vdots \\ a_{M2} \end{array}\right] \dots \left[\begin{array}{c} a_{1N} \\ a_{2N} \\ \vdots \\ a_{MN} \end{array}\right]\right]$$

こんな風に『列ブロックへの分解』を考えると『一列だけの行列』への作用が記述しやすくなるわ。いま、

$$\mathbf{a}_j := [\ a_{ij}\ \mid i : 1 \downarrow M\ ],\ \mathbf{x} := [\ x_i\ \mid i : 1 \downarrow N\ ]$$

とすると

$$\left[\begin{array}{c|c} a_{ij} & \begin{array}{c} i : 1 \downarrow M \\ j : 1 \rightarrow N \end{array} \end{array}\right] \cdot \mathbf{x} = \sum_{k=1}^{N} x_k\, \mathbf{a}_k$$

となるわ」

「こう書いてもいいんだよな：」

$$[\mathbf{a}_1\ \ \mathbf{a}_2 \dots \mathbf{a}_N] \cdot \mathbf{x} = \sum_{k=1}^{N} x_k\, \mathbf{a}_k$$

「その通りね。単位行列 E_N なんかもこんな風に『列ブロックへ分解』できるわ：」

$$E_N = \left[\mathbf{e}_1^{(N)}\ \ \mathbf{e}_2^{(N)}\ \ \dots\ \ \mathbf{e}_N^{(N)}\right].$$

念のために書けば

$$\mathbf{e}_k^{(N)} = \left[\ \delta_{k,i}\ \middle|\ i : 1 \downarrow N\right]$$

だったわね。ただし、ここでの δ はいわゆるクロネッカーのデルタで、

$$\delta_{i,j} := \begin{cases} 1 & (i = j), \\ 0 & \text{otherwise} \end{cases}$$

だね」

「表記法の話が続いて退屈だな……」

「うん、そろそろ表記法の話は締めくくるわ。最後に『必要に応じて $K \times 1$ 行列を $\mathbb{R}^K$ と同一視する』という規約に触れておくわ：」

$$\mathrm{Mat}(K \times 1) \simeq \mathbb{R}^K.$$

「―― この規約によって、たとえば『$\mathbb{R}^N$ の標準基底 $\left\langle \mathbf{e}_1^{(N)}, \mathbf{e}_2^{(N)}, \ldots, \mathbf{e}_N^{(N)} \right\rangle$』と言ってよくなるわね」

「確かに、こうすれば多少の記号の節約にはなるかもな……」

「同じくこの規約によって、$M \times N$ 行列 A と $x \in \mathbb{R}^N$ の積 $A \cdot x$ が定義できるようになるわ」

4.3.2 連立一次方程式の解（1）

「―― さて、さっきも言ったとおり連立一次方程式は

$$A \cdot x = b$$

と書けるわね。まずは A が正方行列の場合を考えましょう。これはどう解けるか覚えてるかしら？」

「えーと、逆行列 A^{-1} を使えば

$$x = A^{-1} \cdot b$$

だよな」

「大体合ってる……けど、逆行列はいつでも存在するわけじゃないことには注意が必要ね」

「あー、$\det(A) \neq 0$ じゃなきゃいけないんだった」

「$\det(A) \neq 0$ となる正方行列 A を『正則行列』と呼ぶんだったね」

「そんな言葉もあったな」

「A が正則行列の場合に大事なことは『解はただ一つのみ存在する』ことね」

「それって証明できることなのか？」

「そうね、線形代数のよい復習になるとおもうから証明を自分で考えてみるといいわ」

4.3.3 連立一次方程式の解（2）

「さて、では次に A が $M \times N$ 行列の場合を考えるわ。ただし、あとの議論に必要な範囲だけ考えたいので、つぎの制限を設けることにするわ：」

- $b = [0, \ldots, 0]^\intercal$.
- $M < N$.
- $\mathrm{rank}(A) = M$.

「―― この場合の大きな特徴は、答えが複数個あるかもしれないことね」

「正直、一般の連立方程式あたりから面倒になって投げてたぜｗｗｗ」

「笑い事じゃないんだけど……でもそうね、この辺は線形代数の撃墜ポイントの一つね。さて、$\mathrm{rank}(A) = M$ だから、一般性を失うことなく」

$$\det\left(\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MM} \end{bmatrix}\right) \neq 0 \tag{4.3}$$

だと仮定できるわね」

「えっと……なんでそんな仮定をしていいんだ？」

「だから、必要に応じて変数名を入れ替えることにすれば結局同じことでしょ」

「えーと……、何が必要で、何と何が同じ？」

なるほど、そういうことか。こういう「数学的言い回し」を理解するためには国語能力だけでなく「数学的な運動感覚」とでも言うべきものも必要になる。線形代数を学ぶ事を通して、そういう感覚を養うことが期待されているわけだが、感覚というのは教えるのも教わるのも少し難しい。

「そうね、少し回り道になるけど丁寧に説明してみることにするわ。まず、

$$A = [\mathbf{a}_1 \ \mathbf{a}_2 \ \cdots \ \mathbf{a}_N]$$

と列ブロックへ分解して考えると、$\mathrm{rank}(A) = M$ という条件から、次が言えるわよね：」

$$^{\exists} i_1, \ldots, i_M \in \{1, \ldots, N\} \qquad \det([\mathbf{a}_{i_1} \ \mathbf{a}_{i_2} \ \cdots \ \mathbf{a}_{i_M}]) \neq 0.$$

「小行列式を使った rank の定義からだと、これはすぐわかるな」

「どの定義でも同じことよ。さて、そこで

$$\{j_1, \ldots, j_{N-M}\} := \{1, \ldots, N\} - \{i_1, \ldots, i_M\}$$

としてやって、$i_1, \ldots, i_M$ と $j_1, \ldots, j_{N-M}$ を使って行列 T を

$$T := \left[\mathbf{e}^{(N)}_{i_1} \ \ldots \ \mathbf{e}^{(N)}_{i_M} \ \mathbf{e}^{(N)}_{j_1} \ \ldots \ \mathbf{e}^{(N)}_{j_{N-M}}\right]$$

として定義した上で

$$B := A \cdot T$$

としてやると、行列 B の最初の M 列を取り出して横に並べた行列は正則行列になるわ」

「ええと……そうだな、そもそもそうなるように $i_1, \ldots, i_M$ を選んだんだからな」

「そして、未知変数 x についての一次方程式 $A \cdot x = [0, \ldots, 0]^{\intercal}$ の解は、未知変数 y についての一次方程式 $B \cdot y = [0, \ldots, 0]^{\intercal}$ の解から、$x = T \cdot y$ としてやれば得られるわ」

「ちょっと待って —— y が $B \cdot y = [0, \ldots, 0]^{\intercal}$ を満たすとき、

$$A \cdot x = A \cdot (T \cdot y) = A \cdot (T \cdot T^{-1}) \cdot T \cdot y = (A \cdot T) \cdot y = B \cdot y = [0, \ldots, 0]^{\intercal}$$

だから、確かにそうだね」

「T は正則行列だから、$A \cdot x = [0, \ldots, 0]^{\intercal}$ の解から $B \cdot y = [0, \ldots, 0]^{\intercal}$ の解を作ることもできるわね。こんなふうにして、行列 T を介することで『最初の M 列を横に並べて作った行列が正則の場合』に議論を還元できるのよ。つまり、この場合だけに制限して論じても議論の本質は損なわれないことになるわ」

「なるほど、それがさっき言っていた『一般性を失うことなく』という謎フレーズが指してることなんだな」

「英語の “without loss of generality” というフレーズを直訳調に日本語に移したものなんだけどね」

「でもさ、実際にプログラムを作るときには『行列 T は適当にそっちで考えてください』ってわ

けには行かないよな？」

「ええ、そうね。連立一次方程式を解くプログラムを作る場合には『最初の M 列を横に並べて作った行列が正則だと仮定して良い』では済まないわね。結局、数学は人間の理解のために書かれていて、プログラムは機械のために書かれている、そういう違いがあるのだと思う」

「それはどうだろう、論理プログラミングや定理証明支援システムなんかが普及しつつあるから『人間のため』と『機械のため』の線引きは曖昧じゃないかな？」

「確かに、その線引きは曖昧かもね。そして、数学とプログラミングの文化がもっと互いに近くなったら、将来は『必要ならば変数名の付け替えを適宜行うことによって』だとか『一般性を失うことなく〜と仮定してよい』といったフレーズを目にする機会も減るかもしれない」

「なるほどな」

「ただ、いま議論したように『最初の M 列を横に並べて作った行列が正則の場合』に絞って考えれば、複雑な添字操作を避けて議論の本質に集中できるという利点があるわね」

「つまるところ、『何が本質か』ってあたりで問題意識の違いがあるみたいだな」

「あまり深刻な思想的対立と考えなくてもいいかもしれないけどね —— 複雑な添字を使ってると印刷ミスが起こりやすいからなるべく簡単に書きたかった、ぐらいの意味合いだってあったろうし」

「確かに、それもありそうだな」

「さて、『一般性を失うことなく』でずいぶん脱線してしまったけれど本筋に戻ることにしましょう —— 式 (4.3) が成り立つので、

$$A_1 := \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & & \vdots \\ a_{M1} & a_{M2} & \cdots & a_{MM} \end{bmatrix}, \quad A_2 := \begin{bmatrix} a_{1,M+1} & a_{1,M+2} & \cdots & a_{1,N} \\ a_{2,M+1} & a_{2,M+2} & \cdots & a_{2,N} \\ \vdots & \vdots & & \vdots \\ a_{M,M+1} & a_{M,M+2} & \cdots & a_{M,N} \end{bmatrix}$$

とすると A_1 は正則行列になるわね。元の行列 A は、ブロック行列として

$$A = [A_1 \ A_2]$$

と表されるわ。まず、次のように z_j を作ると ——

$$\begin{aligned} y_j &:= \mathbf{e}_j^{(N-M)} \qquad (j = 1, \ldots, N-M), \\ x_j &:= -{A_1}^{-1} \cdot A_2 \cdot y_j \qquad (j = 1, \ldots, N-M), \\ z_j &:= \langle x_j, y_j \rangle \qquad (j = 1, \ldots, N-M). \end{aligned}$$

この z_j たちは方程式 $A \cdot x = [0, \ldots, 0]^\intercal$ の $N-M$ 個の非自明解となるわ」

「考えてみるぜ：

$$A \cdot z_j = A \cdot \langle x_j, y_j \rangle = A_1 \cdot x_j + A_2 \cdot y_j = [0, \ldots, 0]^\intercal$$

だから確かに z_j は解になってるな」

「さて、これで $A \cdot x = [0, \ldots, 0]^\intercal$ が $N-M$ 個の解を持つ事が示されたね。これらが一次独立だということも明らかだわ」

「そうだね、一次独立性の定義から簡単にわかるね」

「さて、今は斉次の場合を考えてるから結局

$$\mathrm{Ker}(A) = \left\{ A \cdot x = [0, \ldots, 0]^\intercal \text{の解} \right\} \supseteq \mathrm{span} \langle z_1, \ldots, z_{N-M} \rangle$$

となることがわかったわね。だから

$$\dim \left(\mathrm{Ker}(A) \right) \geq N - M$$

となるわね」

「うん、確かにそうだな」

「この不等号を左右逆にひっくり返したものも成り立つわ」

「……どうやって示すんだろう？」

「話の流れとしては、$A\cdot x=[0,\ldots,0]^{\intercal}$ に K 個の一次独立な解が存在すると仮定すると $K\leq N-M$ となることを示せばいいわね。では、まず K 個の一次独立な解（列ベクトル）を横に並べて作った行列を

$$Z:=\begin{bmatrix} x_{1,1} & \cdots & x_{1,K} \\ \vdots & & \vdots \\ x_{M,1} & \cdots & x_{M,K} \\ y_{1,1} & \cdots & y_{1,K} \\ \vdots & & \vdots \\ x_{N-M,1} & \cdots & x_{N-M,K} \end{bmatrix}$$

としましょう。行列 Z の各列を $\mathbb{R}^N$ に属するベクトルだとみなすとき、高々 N 個までしか一次独立ではあり得ないわね。だから $K\leq N$ で、

$$\mathrm{rank}(Z)=K \tag{4.4}$$

となるわね。今、

$$X:=\begin{bmatrix} x_{1,1} & \cdots & x_{1,K} \\ \vdots & & \vdots \\ x_{M,1} & \cdots & x_{M,K} \end{bmatrix},\ Y:=\begin{bmatrix} y_{1,1} & \cdots & y_{1,K} \\ \vdots & & \vdots \\ x_{N-M,1} & \cdots & x_{N-M,K} \end{bmatrix}$$

とすると行列 Z が斉次方程式の解を集めて作られていることから

$$A\cdot Z=\mathbf{O}$$

となるけど、この条件を行列 X,Y を使って書き直せば

$$[A_1\ A_2]\cdot\begin{bmatrix} X \\ Y \end{bmatrix}=\mathbf{O}$$

となるわね。これを更に変形していけば

$$\begin{aligned} &\Leftrightarrow A_1\cdot X+A_2\cdot Y=\mathbf{O}\ \Leftrightarrow\ X+A_1^{-1}\cdot A_2\cdot Y=\mathbf{O} \\ &\Leftrightarrow\ \begin{bmatrix} E_M & A_1^{-1}\cdot A_2 \\ \mathbf{O} & E_{N-M} \end{bmatrix}\cdot\begin{bmatrix} X \\ Y \end{bmatrix}=\begin{bmatrix} \mathbf{O} \\ Y \end{bmatrix}. \end{aligned} \tag{4.5}$$

上式左辺第一因子は上三角行列なのでその行列式は対角成分の積すなわち 1 になるわね。よってこれは正則行列だね。一般に、正則行列はその作用によって rank を変えないわ。よって次の関係式が成り立つわね：

$$\mathrm{rank}\left(\begin{bmatrix} E_M & A_1^{-1}\cdot A_2 \\ \mathbf{O} & E_{N-M} \end{bmatrix}\cdot\begin{bmatrix} X \\ Y \end{bmatrix}\right)=\mathrm{rank}\left(\begin{bmatrix} X \\ Y \end{bmatrix}\right). \tag{4.6}$$

それから、式 (4.5) の両辺の rank を取れば次の関係式が得られるわ：

$$\mathrm{rank}\left(\begin{bmatrix} E_M & A_1^{-1}\cdot A_2 \\ \mathbf{O} & E_{N-M} \end{bmatrix}\cdot\begin{bmatrix} X \\ Y \end{bmatrix}\right)=\mathrm{rank}\left(\begin{bmatrix} \mathbf{O} \\ Y \end{bmatrix}\right). \tag{4.7}$$

(4.6)(4.7) から：

$$\mathrm{rank}\left(\begin{bmatrix} X \\ Y \end{bmatrix}\right)=\mathrm{rank}\left(\begin{bmatrix} \mathbf{O} \\ Y \end{bmatrix}\right). \tag{4.8}$$

これと (4.4) を合わせれば

$$\mathrm{rank}\left(\begin{bmatrix} \mathbf{O} \\ Y \end{bmatrix}\right) = K \tag{4.9}$$

となるけど、rank の性質から

$$\mathrm{rank}\left(\begin{bmatrix} \mathbf{O} \\ Y \end{bmatrix}\right) = \mathrm{rank}(Y) \leq \min\{K, N-M\}. \tag{4.10}$$

(4.8)(4.9)(4.10) より結局

$$K \leq N-M \tag{4.11}$$

となるわ」

「長くなったけど結局これまで考えたことと合わせれば

$$\dim\left(\mathrm{Ker}(A)\right) = N-M$$

だとわかったわけだな」

「そういうことね。今までの結果を『N 次元空間に M 個の一次独立な制約を加えると $N-M$ 次元空間になる』と標語的にまとめることもできるわね」

「なんとなく知っていることではあったけど、きちんと理詰めで理解したことはなかったな」

「二次元や三次元ぐらいまでなら頭に思い浮かべて理解することができるけど、線形図形の場合に限ったとしても、次元がもっと高い場合にどうなるかはわかりづらいわね」

「わかりづらいというのもあるし、たとえわかったところでそれをどうやって言葉で他人に伝えるかという問題もあるな」

「ええ、だから数ベクトル空間を使って問題を形式化したうえで議論したのよ」

「数ベクトル空間を使って問題を形式化するとなんで他人に伝わりやすくなるんだろう？」

「それはね、高次元の線形図形の問題を形式化してしまえば幾何的な視点の変更や操作 —— これらは言葉で伝えるのが難しい —— を、計算や組み合わせ的議論 —— これらは工夫すれば他人に伝えられる —— によって述べられるからよ」

「難しいことを簡単なことで置き換えてる……？」

「願わくばね。高次元線形図形の問題を数ベクトル空間の操作に置き換えた代価として、数ベクトル空間の操作から高次元の図形を見通すことが求められるようになったわ。果たしてそれが簡単なことだと言い切れるのかどうか……。」

「よくわからないけど、とりあえず『高次元が見えちゃう特殊能力者』しか議論に参加できないよりはいいと思うぜ……！」

「さて、わざわざここまで詳しく斉次連立一次方程式の話をしたのは、これが『陰関数定理』で一番単純な場合に関係するからよ。いま

$$F(x,y) := A_1 \cdot x + A_2 \cdot y$$

としてベクトル値関数 F を定義すると、関係式

$$F(x,y) = [0,\ldots,0]^{\intercal}$$

は x について解けて、$g(y) := -A_1^{-1} \cdot A_2 \cdot y$ とすると

$$F(g(y),y) = [0,\ldots,0]^{\intercal}$$

となるわね」

「確かに、そうなるね」

4.3.4 陰関数

「一般に、関係式を満たすように決まる関数を『陰関数』と言うのよ」

「ちょっとよくわからないなぁ」

「もう少し丁寧に説明するわ。まず、関数 f が

$$f(x) = x^3 + 2x^2 + 5$$

のように具体的に、与えられている場合が一般的な関数ね。これに対して、関係式

$$x^7 + x^3t^3 + t^7 = 1$$

のような関係式を通して間接的に関数 $x(t)$ が与えられていると考えるとき、この $x(t)$ を『陰関数』と呼ぶのよ」

「なるほどな……？　根号と加減乗除の組み合わせで具体的に表示することはできなさそうだけど、一応関数ではあるのか。でも待って、具体的に表示できないのに存在することだけ言えるってのも不思議……だよね？」

「今の場合 $x(t)$ を『根号と加減乗除の組み合わせで表示すること』はできないけれど、x についての方程式 $x^7 + x^3t^3 + t^7 - 1 = 0$ の解は（代数学の基本定理から）存在することを保証できるわ」

「なるほど。『存在する』と『限られた道具の組み合わせでそれを表示することができる』は違う話だったね、そういえば」

「そういうことね」

「陰関数を考えるとき、(i) それは存在するのか (ii) 存在したとして、それは一意か ということが問題になるわ。比較的簡単な関係式

$$x^2 + y^2 = 1$$

を x について形式的に解くと

$$x(y) = \pm\sqrt{1 - y^2}$$

となって、値が一意に定まらないという意味で『関数』とは言い難いわね。うまくどこかに制限を設けないと陰関数は決まらないわ。そういう意味では、陰関数というのはなかなか繊細なものだわ。これについては、代数的な関係式とは限らない一般的な関係式についての定理が知られているわ：」

陰関数の定理

前提：

- $\Omega :\in \mathscr{O}_N$
- $F \in C^1(\Omega;\mathbb{R}^M)$
- $z_0 = \langle x_0, y_0 \rangle :\in \Omega$
- $\partial_1 \overline{F}(x_0, y_0)$ が正則行列

主張：

$$^\exists\Omega_1 \in \mathscr{O}_{N-M}(y_0)\ \ ^\exists\Omega_2 \in \mathscr{O}_M(x_0)\ \ ^\exists g\colon \Omega_1 \to \Omega_2 \quad \textbf{(Imp1) (Imp2) (Imp3) (Imp4)};$$

(Imp1) $^\forall y \in \Omega_1 \qquad \overline{F}\,(g(y), y) = \overline{F}(x_0, y_0).$

(Imp2) $^\forall \langle x, y\rangle \in \Omega_2 \times \Omega_1 \qquad \overline{F}(x, y) = \overline{F}(x_0, y_0) \Rightarrow x = g(y).$

(Imp3) $g \in C^1\,(\Omega_1; \Omega_2)\,.$

(Imp4) $g'(y) = -\left(\partial_1 \overline{F}\big(g(y), y\big)\right)^{-1} \cdot\ \partial_2 \overline{F}\big(g(y), y\big).$

ただし、

$$\partial_1\overline{F}(x,y) := \left[\begin{array}{c|c} \dfrac{\partial\overline{F}_i}{\partial x_j}(x,y) & \begin{array}{l} i:1\downarrow M \\ j:1\to M \end{array} \end{array}\right],$$

$$\partial_2\overline{F}(x,y) := \left[\begin{array}{c|c} \dfrac{\partial\overline{F}_i}{\partial y_j}(x,y) & \begin{array}{l} i:1\downarrow M \\ j:1\to N-M \end{array} \end{array}\right].$$

「あー……話が面倒になってきた……」

「期待するような目で見てもダメよ。Lagrange の未定乗数法は元々多変数の関数の話なんだから多変数の微積分の話になるのは仕方ないわ」

「いま **(Imp4)** を見て気づいたんだけど、この $-\Big(\partial_1\overline{F}\big(g(y),y\big)\Big)^{-1}\cdot\ \partial_2\overline{F}\big(g(y),y\big)$ ってのは連立一次方程式のところで出てきた $-A_1^{-1}\cdot A_2$ と同じ形だな」

「気づいてくれて嬉しいわ。この形こそ、陰関数の定理がさっきの連立一次方程式の話の一般化に対応している証拠なのよ」

4.4 多変数の微分の基礎事項

本記事での約束 (2)

- $^\forall x\in\mathbb{R}^K$ に対して、そのノルム $|x|$ を

$$|x| := \left(\sum_{k=1}^{K}|x_k|^2\right)^{1/2}$$

と定義します。

- $^\forall A\in \mathrm{Mat}(K_1\times K_2)$ に対して、そのノルム $\|A\|$ を

$$\|A\| := \sup_{x\in\mathbb{R}^{K_2},|x|\le 1} |A\cdot x|$$

と定義します。

- $^\forall x\in\mathbb{R}^K, {}^\forall a>0$ に対して、開球 $\mathrm{B}^{(K)}(x;a)$ を次のように定義します：

$$\mathrm{B}^{(K)}(x;a) := \Big\{y\in\mathbb{R}^K\ \Big|\ |x-y|<a\Big\}.$$

「ノルムってのはあれだよな、長さみたいな？」

「線形空間のノルムはベクトルの大きさに該当する概念で、ノルムを通して線形空間に位相が入る——つまり連続性だとか極限を論じられるようになる——というのが大事なことね」

「行列全体ってのも『線形空間』なんだね……？」

「線形空間の公理さえ満たしていれば線形空間として扱えるのよ」

「なるほど……でも行列にノルム入れると何がうれしいんだろう」

「ノルムが入れば距離が、そして距離が入れば位相が定まるわ」

「——そして位相が定まれば極限や連続性を語れるわけか」

「そうね。例えば行列式なんかも『行列の連続関数』だと言えるわ。つまり

$$^\forall A\in\mathrm{Mat}(N\times N)\ {}^\forall\varepsilon>0\ {}^\exists\delta>0\ {}^\forall B\in\mathrm{Mat}(N\times N)\quad \|B-A\|<\delta\ \Rightarrow\ |\det(B)-\det(A)|<\varepsilon$$

ということね」

「どうやって示すんだろう……？」

「概略だけ示すわね：$A=\left[\begin{array}{c|c} a_{ij} & \begin{array}{c} i:1\downarrow N \\ j:1\to N \end{array} \end{array}\right]$, $B=\left[\begin{array}{c|c} b_{ij} & \begin{array}{c} i:1\downarrow N \\ j:1\to N \end{array} \end{array}\right]$ とするとき、

$$|\det(B)-\det(A)| \leq \sum_{\sigma\in \mathrm{Sym}(N)} \left| \prod_{i=1}^{N} b_{i,\sigma(i)} - \prod_{i=1}^{N} a_{i,\sigma(i)} \right|. \tag{4.12}$$

一方、$\|B-A\|<\delta$ ならば、$^{\forall}j$ に対して $\left|(B-A)\cdot \mathbf{e}_j\right|<\delta$ より

$$\sum_{i=1}^{N} |b_{ij}-a_{ij}|^2 < \delta^2.$$

このことから結局

$$\|B-A\|<\delta \Rightarrow {}^{\forall}i,j\in\{1,\ldots,N\} \quad |b_{ij}-a_{ij}|<\delta.$$

(4.12) とこれを組み合わせれば連続性が出るわ」

「なるほど……行列の差のノルムが小さければ成分同士の差も小さい事を使うんだな！」

「さて、Lagrange の未定乗数法は関数の極値点についての命題だったから、念のためこれらについても復習しておくわ：」

定義：関数の極値点

前提：

- D : 自然数
- $\Omega :\in \mathscr{O}_D$
- $f\colon \Omega\to\mathbb{R}$
- $x_0 :\in \Omega$

次のように定義する：

■点 x_0 が関数 f の極大点 $\overset{\text{def}}{\iff}$ $^{\exists}G\subseteq\Omega$ $\Big(G\in\mathscr{O}_D(x_0) \ \wedge\ {}^{\forall}y\in G\ \ f(x_0)\geq f(y)\Big)$.

■点 x_0 が関数 f の極小点 $\overset{\text{def}}{\iff}$ $^{\exists}G\subseteq\Omega$ $\Big(G\in\mathscr{O}_D(x_0) \ \wedge\ {}^{\forall}y\in G\ \ f(x_0)\leq f(y)\Big)$.

■点 x_0 が関数 f の極値点 $\overset{\text{def}}{\iff}$ $\big(x_0$が関数 f の極大点$\big) \ \vee\ \big(x_0$が関数 f の極小点$\big)$.

「ここでは極大ってのは local maxima で、極小ってのは local minima だね」

「『極大』とか『極小』と訳される言葉は数学の中だけでも幾つかあるけど、今使ってるのはそういう意味だね」

4.4.1 微分

「―― さて、一変数のときの微分は標語的に言えば『一次式による近似』だと言えるわ。そして、この観点に基づいて、微分は多変数の場合に次のように一般化されるわ：」

定義：多変数関数の微分

前提：

- K_1, K_2 : 自然数
- $\Omega :\in \mathscr{O}_{K_1}$
- $f\colon \Omega\to\mathbb{R}^{K_2}$
- $x_0 :\in \Omega$

次のように定義する：

■関数 f が x_0 で微分可能

$$\overset{\text{def}}{\iff} \quad {}^{\exists}A \in \mathrm{Mat}(K_2 \times K_1) \quad \left|f(x_0+h)-f(x_0)-A\cdot h\right| = o(|h|) \quad (h \to [0,\dots,0]^{\intercal}).$$

このとき $f'(x_0) := A$ とする。

■関数 f が Ω 上で微分可能 $\overset{\text{def}}{\iff}$ ${}^{\forall}x \in \Omega$ $\big(f$ は x で微分可能$\big)$.

「ふむふむ……つまり多変数の場合は『微分係数』が行列になるのか。一変数の場合は 1×1 行列だとみなせるから、確かにこれは一変数の場合の一般化になってるぽいな」

「さて、『微分係数』を一般化した行列の成分については次の命題が基本的よ：」

微分の性質 (1)

前提：

- K_1, K_2 : 自然数
- $\Omega :\in \mathscr{O}_{K_1}$
- $f\colon \Omega \to \mathbb{R}^{K_2}$
- $x_0 :\in \Omega$
- f は x_0 で微分可能

次が成立する：

$$f'(x_0) = \left[\ \frac{\partial f_i}{\partial x_j}(x_0)\ \middle|\ \begin{array}{l} i : 1 \downarrow K_2 \\ j : 1 \to K_1 \end{array}\ \right].$$

略証. $f'(x_0) = [\mathbf{a}_1\ \cdots\ \mathbf{a}_{K_1}]$ とするとき、${}^{\forall}j \in \{1,...,K_1\}, {}^{\forall}\varepsilon \in \mathbb{R}$ に対して

$$f(x_0+\varepsilon\mathbf{e}_j)-f(x_0)-[\mathbf{a}_1\ \cdots\ \mathbf{a}_{K_1}]\cdot\varepsilon\,\mathbf{e}_j = o(|h|) \quad (h \to [0,\dots,0]^{\intercal})$$

となる（煩雑なので $\mathbf{e}_j^{(K_1)}$ を $\mathbf{e}_j$ と略記した）。$[\mathbf{a}_1\ \cdots\ \mathbf{a}_{K_1}]\cdot\mathbf{e}_j = \mathbf{a}_j$ なので

$$\mathbf{a}_j = \lim_{\varepsilon\to 0}\frac{f(x_0+\varepsilon\,\mathbf{e}_j)-f(x_0)}{\varepsilon} = \left[\ \frac{\partial f_i}{\partial x_j}(x_0)\ \middle|\ i : 1 \downarrow K_2\right].$$

□

「なるほど、『微分係数』の行列の成分はそれぞれの成分をいろんな方向に偏微分したやつになってるんだね」

「微分可能な関数の極値点の様子も、一変数の場合を次のように一般化できるわ：」

微分の性質 (2)

前提：

- K : 自然数
- $\Omega :\in \mathscr{O}_K$
- $f\colon \Omega \to \mathbb{R}$
- $x_0 :\in \Omega$
- f は x_0 で微分可能

x_0 が f の極値点のとき、次が成立する：

$$f'(x_0) = [0,...,0].$$

略証. まず、点 x_0 が f の極大点である場合を示す。極大点の定義から、ある $\delta > 0$ を取れば

$$^{\forall}x \in \Omega \quad \Big(|x - x_0| < \delta \Rightarrow f(x_0) \geq f(x)\Big).$$

一方、微分の定義から

$$f(x_0 + h) - f(x_0) - f'(x_0) \cdot h = o(|h|) \qquad (h \to [0, \ldots, 0]^{\intercal})$$

であり、$h = \varepsilon \mathbf{e}_j$ とすると次が得られる：

$$f(x_0 + \varepsilon \mathbf{e}_j) - f(x_0) = \varepsilon \frac{\partial f}{\partial x_j}(x_0) + o(\varepsilon) \qquad (\varepsilon \to 0).$$

もし $\dfrac{\partial f}{\partial x_j} \lesseqgtr 0$ ならば、狭い範囲の ε の変動で $f(x_0 + \varepsilon \mathbf{e}_j) > f(x_0)$ とできることがわかる。これは極大性に反するので $\dfrac{\partial f}{\partial x_j} = 0$ でなければならない。これがすべての $j \in \{1, ..., K\}$ について成立する。よって

$$f'(x_0) = [0, ..., 0].$$

極小点である場合も同様の議論で示せるので略する。 □

「『できることがわかる』の辺りの細かい議論を省略したんだね」

「難しくないから省略したのよ」

「簡単なのになぜ省略したのかな……」

「第一に、解析学における連続性の扱いについてきちんと理解していれば省略された部分を補うのは容易であること、第二に、きちんと書くのは面倒であること、からかな。もっと短く言えば、簡単だけど面倒」

「なるほどなぁ（迫真）」

「さて、合成関数の微分に関して、一変数の場合を一般化した次のような定理が成り立つわ：」

4.4.2 連鎖律

連鎖律

前提：

- $\Omega_1 :\in \mathscr{O}_{N_1}$
- $\Omega_2 :\in \mathscr{O}_{N_2}$
- $\Omega_3 :\in \mathscr{O}_{N_3}$
- $f \colon \Omega_1 \to \Omega_2$
- $g \colon \Omega_2 \to \Omega_3$

主張：**(Ch1)** かつ **(Ch2)** のとき **(Ch3)** かつ **(Ch4)**.
ただし、

(Ch1) f はΩ_1上微分可能

(Ch2) g はΩ_2上微分可能

(Ch3) $g \circ f$ はΩ_1上微分可能

(Ch4) $\forall x \in \Omega_1 \qquad (g \circ f)'(x) = \big(g' \circ f\big)(x) \cdot f'(x).$

略証.

$$\begin{aligned}
&\left|g\circ f(x+h)-\left(g'\circ f\right)(x)\cdot f'(x)\cdot h\right| \\
&=\left|g\left(f(x+h)\right)-g\left(f(x)\right)-g'\left(f(x)\right)\cdot f'(x)\cdot h\right| \\
&\leq\left|g\left(f(x+h)\right)-g\left(f(x)\right)-g'\left(f(x)\right)\cdot\left(f(x+h)-f(x)\right)\right| \\
&\qquad\qquad+\left|g'\left(f(x)\right)\right|\left|f(x+h)-f(x)-f'(x)\cdot h\right|
\end{aligned}$$

ここで $y:=f(x),\Delta_h:=f(x+h)-f(x)$ とすると

$$\begin{aligned}
&=\left|g(y+\Delta_h)-g(y)-g'(y)\cdot\Delta_h\right| \\
&\qquad\qquad+\left|g'(f(x))\right|\left|f(x+h)-f(x)-f'(x)\cdot h\right|.
\end{aligned}$$

g の微分可能性と f の連続性より、最後の式の第一項は $o(h)$ となる。第二項は f の微分可能性により $o(h)$ になる。よって

$$\lim_{h\to[0,\ldots,0]^\intercal}\frac{\left|g\circ f(x+h)-g\circ f(x)-(g'\circ f)(x)\cdot f'(x)\cdot h\right|}{|h|}=0.$$

$$\therefore\ \ (g\circ f)'(x)=(g'\circ f)(x)\cdot f'(x).$$

□

「この式は一変数のときの式と同じ形だと思うようにすれば覚えやすいね」

4.5 未定乗数法の証明

「さて——多変数の微積分についてこれくらい復習しておけば、陰関数の定理 (p.55) を使って Lagrange の未定乗数法を証明できるわ：」

未定乗数法の定理の証明. 前提 **(Lag1)**(p.46) により、一般性を失うことなく

$$\det\left(\left[\begin{array}{c|c} \dfrac{\partial g_i}{\partial x_j}(z_0) & \begin{array}{l} i:1\downarrow M \\ j:1\to M \end{array} \end{array}\right]\right)\neq 0$$

と仮定して議論できるわね。いま、$G\colon\Omega\to\mathbb{R}^M$ を

$$G(z):=[g_1(z),\ldots,g_M(z)]^\intercal$$

と定義する。（定理の前提より $G(z_0)=[0,\ldots,0]^\intercal$ である。）陰関数の定理より次が成立するわ：

$$^\exists\Omega_1(y_0)\in\mathscr{O}_{N-M}(x_0)\ \ ^\exists\Omega_2\in\mathscr{O}_M\ \ ^\exists\varphi\colon\Omega_1\to\Omega_2\quad\textbf{(1) (2) (3) (4) :}$$

ただし

(1) $^\forall y\in\Omega_1\quad\overline{G}(\varphi(y),y)=[0,\ldots,0]^\intercal.$

(2) $^\forall\langle x,y\rangle\in\Omega_2\times\Omega_1\quad\left(\overline{G}(x,y)=[0,\ldots,0]^\intercal\Rightarrow x=\varphi(y)\right).$

(3) $\varphi\in C^1(\Omega_1;\Omega_2).$

(4) $\varphi'(y)=-\left(\partial_1\overline{G}(\varphi(y),y)\right)^{-1}\cdot\ \partial_2\overline{G}(\varphi(y),y).$

関数 $f\big|_V$ は、z_0 の近傍で $y\in\Omega_1$ の関数として

$$y\mapsto\overline{f}(\varphi(y),y)$$

と表せる。f の微分を V に沿ったところで計算すると

$$\Big(\overline{f}(\varphi(y),y)\Big)' = \partial_1\overline{f}\big(\varphi(y),y\big)\cdot\varphi'(y)+\partial_2\overline{f}\big(\varphi(y),y\big)$$
$$= -\partial_1\overline{f}\big(\varphi(y),y\big)\cdot\Big(\partial_1\overline{G}\big(\varphi(y),y\big)\Big)^{-1}\cdot\ \partial_2\overline{G}\big(\varphi(y),y\big)+\partial_2\overline{f}\big(\varphi(y),y\big).$$

$z_0=\langle x_0,y_0\rangle$ とすると $x_0=\varphi(y_0)$ なので

$$f'(z_0)=0\ \Rightarrow\ \Big(\overline{f}(\varphi(y),y)\Big)'\Big|_{y=y_0}=0$$
$$\Leftrightarrow \partial_2\overline{f}(x_0,y_0)=\partial_1\overline{f}(x_0,y_0)\cdot\Big(\partial_1\overline{G}(x_0,y_0)\Big)^{-1}\cdot\ \partial_2\overline{G}(x_0,y_0).$$

よって、$\Lambda:=\partial_1\overline{f}(x_0,y_0)\cdot\Big(\partial_1\overline{G}(x_0,y_0)\Big)^{-1}$ とすると

$$\partial_2\overline{f}(x_0,y_0)=\Lambda\cdot\partial_2\overline{G}(x_0,y_0). \tag{4.13}$$

一方、Λ の定義より

$$\partial_1\overline{f}(x_0,y_0)=\Lambda\cdot\partial_2\overline{G}(x_0,y_0). \tag{4.14}$$

(4.13)(4.14) をブロック行列記法でまとめると

$$f'(z_0)=\Lambda\cdot\begin{bmatrix}\partial_1\overline{f}(x_0,y_0) & \partial_2\overline{f}(x_0,y_0)\end{bmatrix}=\Lambda\cdot G'(z_0). \tag{4.15}$$

よって、$\Lambda=[\lambda_1,\ldots,\lambda_M]$ とすると

$$\frac{\partial f}{\partial z_k}(z_0)=\sum_{i=1}^{M}\lambda_i\frac{\partial g_i}{\partial z_k}(z_0)\qquad(k=1,\ldots,N).$$

□

「……なるほどなぁっ！」

「無理して『よくわかった、感動した』って感じに盛り上げようとしなくていいわよ」

「微分係数の行列が二つに分割されたあと合流するところがおもしろかったです（小学生並みの感想）」

4.6 陰関数の定理の証明

「さて、陰関数の定理の証明を理解するために必要な事項を整理しておきましょう」

4.6.1 有限増分の定理

有限増分の定理

前提：

- $\Omega:\in\mathscr{O}_N$
- $f\colon\Omega\to\mathbb{R}^M$

主張：**(FinInc1)** かつ **(FinInc2)** のとき **(FinInc3)**.

ただし、

(FinInc1) Ωは凸集合

(FinInc2) f はΩ上微分可能

(FinInc3) $\forall a,b\in\Omega\qquad |f(b)-f(a)|\le\left(\sup_{x\in\Omega}\|f'(x)\|\right)|b-a|.$

証明. 関数 $q\colon [0,1] \to \Omega$ を

$$t \mapsto t\,b + (1-t)\,a$$

として定義すると、合成関数の微分の公式より

$$\big(f\circ q\big)'(t) = f'\circ q(t)\cdot q'(t) = f'\big(q(t)\big)\cdot(b-a).$$

したがって

$$f\circ q(1) - f\circ q(0) = \int_0^1 \big(f\circ q(t)\big)'(t)dt = \int_0^1 f'\big(q(t)\big)\cdot(b-a)dt.$$
$$\Leftrightarrow f(b)-f(a) = \int_0^1 f'\big(q(t)\big)\cdot(b-a)dt.$$

これを利用すると：

$$|f(b)-f(a)| \le \int_0^1 \big|f'\big(q(t)\big)\cdot(b-a)\big|\,dt \le \int_0^1 \|f'\big(q(t)\big)\|\cdot|b-a|dt$$
$$\le \left(\sup_{x\in\Omega}\big\|f'(x)\big\| \int_0^1 dt\right)|b-a| = \left(\sup_{x\in\Omega}\big\|f'(x)\big\|\right)\ |b-a|.$$

□

「ベクトル値関数が積分されてるけど……」
「成分ごとの Remann 積分を考えればいいわ」
「なるほど」
「陰関数の定理の証明に便利な系を示しておくわ：」

4.6.2 有限増分の定理の系

有限増分の定理の系

前提：

- $\Omega :\in \mathscr{O}_N$
- $f\colon \Omega \to \mathbb{R}^M$

主張：**(Lem1)** かつ **(Lem2)** のとき **(Lem3)**.
ただし、

(Lem1) Ωは凸集合

(Lem2) f はΩ上微分可能

(Lem3) $\forall a,b,c \in \Omega \qquad \big|f(b)-f(a)-f'(c)\cdot(b-a)\big| \le \left(\sup_{x\in\Omega}\ \|f'(x)-f'(c)\|\right)|b-a|.$

（この命題は Jost の本の補題 8.8 に相当します。）

証明. $F\colon \Omega \to \mathbb{R}^M$ を次のように定義する：

$$F(x) := f(x) - f'(c)\cdot x.$$

すると $F'(x) = f'(x) - f'(c)$ となる。有限増分の定理により次が得られる：

$$|F(b)-F(a)| \le \left(\sup_{x\in\Omega}\big\|f'(x)-f'(c)\big\|\right)|b-a|.$$
$$\big|f(b)-f(a)-f'(c)\cdot(b-a)\big| \le \left(\sup_{x\in\Omega}\big\|f'(x)-f'(c)\big\|\right)|b-a|.$$

□

4.6.3 陰関数の定理の証明

「いよいよ準備ができたので陰関数の定理の証明に移りましょう」

「長くなりそうなので、あらすじだけでも……」

「あらすじねぇ……。そうね、あらすじとしては『陰関数を、帰納的に定義された関数列の極限として構成します』という感じかな」

「なるほど」

陰関数の定理の証明. 一般性を失うことなく、$\overline{F}(x_0,y_0)=[0,\dots,0]^{\intercal}$ として良い。次の関係に注意する：

$$\begin{aligned}
&\overline{F'}(x,y)=\begin{bmatrix}\partial_1\overline{F}(x,y) & \partial_2\overline{F}(x,y)\end{bmatrix},\\
&\partial_1\overline{F}(x,y)=\overline{F'}(x,y)\cdot[\mathbf{e}_1,\mathbf{e}_2,\dots,\mathbf{e}_M],\\
&\partial_2\overline{F}(x,y)=\overline{F'}(x,y)\cdot[\mathbf{e}_{N-M+1},\mathbf{e}_{N-M+2},\dots,\mathbf{e}_N].
\end{aligned}$$

$\overline{F'}(x,y)$ は仮定より Ω 上連続なので、$\partial_1\overline{F}(x,y)$ と $\partial_2\overline{F}(x,y)$ はともに Ω 上連続である。更に、det は行列の連続関数なので、連続関数の合成で得られる

$$\langle x,y\rangle\mapsto\det\left(\partial_1\overline{F}(x,y)\right)$$

もまた Ω 上の連続関数である。仮定より

$$\det\left(\partial_1\overline{F}(x_0,y_0)\right)\neq 0$$

であるから、連続関数の性質より次が言える：

$$^{\exists}a,b>0\quad{}^{\forall}\langle x,y\rangle\in\mathrm{B}^{(M)}(x_0;a)\times\mathrm{B}^{(N-M)}(y_0;b)\quad\det\left(\partial_1\overline{F}(x,y)\right)\neq 0. \tag{4.16}$$

以下において

$$L_0:=\partial_1\overline{F}(x_0,y_0)$$

とする。

$\partial_1\overline{F}$ の $\langle x_0,y_0\rangle$ における連続性より次が言える：

$$^{\exists}\eta\in(0,a)\ {}^{\exists}\delta_1\in(0,b)\ {}^{\forall}\langle x,y\rangle\in\mathrm{B}^{(M)}(x_0;\eta)\times\mathrm{B}^{(N-M)}(y_0;\delta_1)\quad\|\partial_1\overline{F}(x,y)-L_0\|<\frac{1}{8\|L_0^{-1}\|}. \tag{4.17}$$

$y\mapsto\overline{F}(x_0,y)$ の y_0 における連続性より次が言える：

$$^{\exists}\delta_2>0\quad{}^{\forall}y\in\mathrm{B}^{(N-M)}(y_0;\delta_2)\qquad\left|\overline{F}(x_0,y)\right|<\frac{\eta}{2\|L_0^{-1}\|}. \tag{4.18}$$

$\delta:=\min\{\delta_1,\delta_2\}$ として更に

$$\Omega_1:=\mathrm{B}^{(N-M)}(y_0;\delta),\ \Omega_2:=\mathrm{B}^{(M)}(x_0;\eta),\ A:=\overline{\Omega_2}$$

とする。

次のような関数を定義しておく：

$$G(x,y):=x-L_0^{-1}\cdot\overline{F}(x,y), \tag{4.19}$$

$$T^{(y)}(x):=G(x,y). \tag{4.20}$$

このとき、以下の **(1)(2)** が成立する：

(1) $\forall y \in \Omega_1 \quad \forall x_1, x_2 \in \Omega_2 \qquad \left| T^{(y)}(x_1) - T^{(y)}(x_2) \right| \leq \frac{3}{8} |x_1 - x_2|.$

(2) $\forall y \in \Omega_1 \quad \forall x \in \Omega_2 \qquad T^{(y)}(x) \in \Omega_2.$

(1) の証明

以下のように $T^{(y)}(x_1) - T^{(y)}(x_2)$ の評価をする：

$$\begin{aligned}
&\left| T^{(y)}(x_1) - T^{(y)}(x_2) \right| = \left| x_1 - L_0^{-1} \cdot \overline{F}(x_1, y) - x_2 + L_0^{-1} \cdot \overline{F}(x_2, y) \right| \\
&= \left| L_0^{-1} \cdot \left(\overline{F}(x_2, y) - \overline{F}(x_1, y) - L_0 \cdot (x_2 - x_1) \right) \right| \\
&\leq \left\| L_0^{-1} \right\| \left| \overline{F}(x_2, y) - \overline{F}(x_1, y) - L_0 \cdot (x_2 - x_1) \right|. \qquad (4.21)
\end{aligned}$$

上式二番目の因子を評価する：

$$\begin{aligned}
&\left| \overline{F}(x_2, y) - \overline{F}(x_1, y) - L_0 \cdot (x_2 - x_1) \right| \\
&\leq \left| \overline{F}(x_2, y) - \overline{F}(x_1, y) - \partial_1 \overline{F}(x_0, y) \cdot (x_2 - x_1) \right| + \left| \left(\partial_1 \overline{F}(x_0, y) - \partial_1 \overline{F}(x_0, y_0) \right) \cdot (x_2 - x_1) \right|
\end{aligned}$$

最後の式の第一項に有限増分の定理の系を適用すると

$$\begin{aligned}
&\leq \left(\sup_{\xi \in \Omega_2} \left\| \partial_1 \overline{F}(\xi, y) - \partial_1 \overline{F}(x_0, y) \right\| \right) |x_2 - x_1| + \left\| \partial_1 \overline{F}(x_0, y) - L_0 \right\| |x_2 - x_1| \\
&\leq \left(\sup_{\xi \in \Omega_2} \left\| \partial_1 \overline{F}(\xi, y) - L_0 \right\| + \left\| \partial_1 \overline{F}(x_0, y) - L_0 \right\| \right) |x_2 - x_1| + \left\| \partial_1 \overline{F}(x_0, y) - L_0 \right\| |x_2 - x_1|
\end{aligned}$$

(4.17) より

$$\leq \left(\left(\frac{1}{8 \left\| L_0^{-1} \right\|} + \frac{1}{8 \left\| L_0^{-1} \right\|} \right) + \frac{1}{8 \left\| L_0^{-1} \right\|} \right) |x_2 - x_1| \qquad (4.22)$$

(4.21)(4.22) より次が得られる：

$$\left| T^{(y)}(x_1) - T^{(y)}(x_2) \right| \leq \frac{3}{8} |x_1 - x_2|.$$

よって **(1)** が示された。

(2) の証明

次のように評価する：

$$\begin{aligned}
&\left| T^{(y)}(x) - x_0 \right| \leq \left| T^{(y)}(x) - T^{(y)}(x_0) \right| + \left| T^{(y)}(x_0) - x_0 \right| \\
&= \left| T^{(y)}(x) - x_0 \right| \leq \left| T^{(y)}(x) - T^{(y)}(x_0) \right| + \left| L_0^{-1} \cdot \overline{F}(x_0, y) \right|
\end{aligned}$$

(1) の評価を用いれば

$$\leq \frac{3}{8} |x - x_0| + \left\| L_0^{-1} \right\| \left| \overline{F}(x_0, y) \right|$$

(4.17) より

$$\leq \frac{3}{8} |x - x_0| + \frac{\eta}{2} < \eta.$$

よって **(2)** が示された。以上により次が成り立つ：

$$\begin{aligned}
&\forall y \in \Omega_1 \quad \forall x \in \Omega_2 \qquad T^{(y)}(x) \in \Omega_2, \\
&\forall y \in \Omega_1 \quad \forall x \in A \qquad T^{(y)}(x) \in A.
\end{aligned}$$

さらに

$$\left(T^{(y)}\right)^n := \underbrace{T^{(y)} \circ \cdots \circ T^{(y)}}_{n\text{ 個}}$$

とする。このとき $m \geq n$ に対して、

$$\begin{aligned}
&\left|\left(T^{(y)}\right)^m(x) - \left(T^{(y)}\right)^n(x)\right| = \left|\sum_{k=n}^{m-1}\left(T^{(y)}\right)^{k+1}(x) - \left(T^{(y)}\right)^k(x)\right| \\
&\leq \sum_{k=n}^{m-1}\left(\frac{3}{8}\right)^k \left|T^{(y)}(x) - x\right| \leq \sum_{k=n}^{m-1}\left(\frac{3}{8}\right)^k \left(\left|T^{(y)}(x)\right| + |x|\right) \\
&\leq 2\eta \sum_{k=n}^{m-1}\left(\frac{3}{8}\right)^k \longrightarrow 0 \quad (m,n \to \infty).
\end{aligned} \tag{4.23}$$

よって、

$$\varphi_k(y) := \left(T^{(y)}\right)^k(x_0)$$

とすると $\{\varphi_k(y)\}_{k=0}^{\infty}$ は Cauchy 列となる。各 $y \in \Omega_1$ に対して

$$g(y) := \lim_{k\to\infty} \varphi_k(y) \tag{4.24}$$

とする。関係式

$$\varphi_{k+1}(y) = T^{(y)}\big(\varphi_k(y)\big) = \varphi_k(y) - L_0^{-1} \cdot \overline{F}\big(\varphi_k(y), y\big)$$

の $k \to \infty$ の極限を取ると F の連続性により次が得られる：

$$g(y) = g(y) - L_0^{-1} \cdot \overline{F}\big(g(y), y\big).$$

$$\therefore \ {}^{\forall} y \in \Omega_1 \quad \overline{F}\big(g(y), y\big) = 0. \tag{4.25}$$

以上により **(Imp1)** が示された。

次に **(Imp2)** を示そう。まず、次に注意する：

$$\overline{F}(x,y) = [0,\ldots,0]^{\intercal} \ \Leftrightarrow \ T^{(y)}(x) = x.$$

言葉で書けば、$\overline{F}(x,y) = [0,\ldots,0]^{\intercal}$ であることは x が $T^{(y)}$ の不動点であることと同値だということである。y に対して、このような不動点がただひとつ存在することが以下のようにして示される。いま、$x_1 = T^{(y)}(x_1)$ かつ $x_2 = T^{(y)}(x_2)$ としよう。すると、次が成り立つ：

$$|x_2 - x_1| = \left|T^{(y)}(x_2) - T^{(y)}(x_1)\right| \leq \frac{3}{8}|x_2 - x_1|.$$

したがって $|x_2 - x_1| = 0$ つまり $x_1 = x_2$ がわかる。これにより **(Imp2)** が示された。

つぎに、関数 g の連続性について考える。関数 g は各点 y についての点列の極限として構成されたので、y についての連続性はいまのところ明らかではないことに注意する。以下で議論するように、実は関数列 $\{\varphi_k\}_{k=0}^{\infty}$ が $C^0(\Omega_1;\Omega_2)$ に含まれる事が言える。あらためて、関数列 $\{\varphi_k\}_{k=0}^{\infty}$ を次のように帰納的に定義する：

$$\begin{aligned}
&\varphi_0(y) := x_0, \\
&\varphi_{k+1}(y) := T^{(y)}\big(\varphi_k(y)\big) \qquad (k \geq 0).
\end{aligned}$$

明らかに $\varphi_0 \in C^0(\Omega_1;\Omega_2)$ である。帰納法を適用するために、$\varphi_k \in C^0(\Omega_1;\Omega_2)$ を仮定したときに $\varphi_{k+1} \in C^0(\Omega_1;\Omega_2)$ となることを示そう。そのために次のような評価を行う：

$$\begin{aligned}
&|\varphi_{k+1}(y_2)-\varphi_{k+1}(y_1)| = \left|T^{(y_2)}(\varphi_k(y_2)) - T^{(y_1)}(\varphi_k(y_1))\right| \\
&= \left|\varphi_k(y_2) - L_0^{-1}\cdot\overline{F}\big(\varphi_k(y_2),y_2\big) - \varphi_k(y_1) - L_0^{-1}\cdot\overline{F}\big(\varphi_k(y_1),y_1\big)\right| \\
&\le |\varphi_k(y_2)-\varphi_k(y_1)| + \left\|L_0^{-1}\right\| \ \left|\overline{F}\big(\varphi_k(y_2),y_2\big) - \overline{F}\big(\varphi_k(y_1),y_1\big)\right|. \qquad (4.26)
\end{aligned}$$

$^\forall\varepsilon > 0$ に対して、F の連続性から次が言える：

$$^\exists\rho_1,\rho_2>0 \quad |x_1-x_2|<\rho_1 \wedge |y_1-y_2|<\rho_2 \Rightarrow \left|\overline{F}(x_2,y_2)-\overline{F}(x_1,y_1)\right| < \frac{\varepsilon}{2\left\|L_0^{-1}\right\|}$$

φ_k の連続性（帰納法の仮定）より

$$^\exists\rho_3>0 \quad |y_1-y_2|<\rho_3 \Rightarrow |\varphi_k(y_2)-\varphi_k(y_1)| < \min\left\{\frac{\varepsilon}{2},\rho_1\right\}.$$

よって、$\rho_0 := \min\{\rho_2,\rho_3\}$ とすると次が成り立つ：

$$|y_1-y_2|<\rho_0 \Rightarrow \left|\overline{F}\big(\varphi_k(y_2),y_2\big) - \overline{F}\big(\varphi_k(y_1),y_1\big)\right| < \frac{\varepsilon}{2\left\|L_0^{-1}\right\|}, \qquad (4.27)$$

$$|y_1-y_2|<\rho_0 \Rightarrow |\varphi_k(y_2)-\varphi_k(y_1)| < \frac{\varepsilon}{2}. \qquad (4.28)$$

式 (4.26)(4.27)(4.28) を合わせれば次が得られる：

$$|y_1-y_2|<\rho_0 \Rightarrow |\varphi_{k+1}(y_2)-\varphi_{k+1}(y_1)| < \varepsilon.$$

以上により、φ_k が連続ならば φ_{k+1} も連続になることが示された。関数列 $\{\varphi_k\}_{k=0}^\infty$ の一様収束性はすでに (4.23) で示したので、『一様収束する連続関数列は連続関数に収束する』により g の連続性が従う。

最後に **(Imp3)(Imp4)** をまとめて示そう。

$y_1 \in \Omega_1, x_1 := g(y_1), y \in \Omega_1, y := g(y)$ とすると、$\overline{F}(x_1,y_1) = \overline{F}(x,y) = [0,\dots,0]^\intercal$ となる。$^\forall\langle x,y\rangle \in \Omega_2\times\Omega_1$ に対し、$\nu(x,y)$ を関係式

$$\overline{F}(x,y)-\overline{F}(x_1,y_1) = \partial_1\overline{F}(x_1,y_1)\cdot(x-x_1) + \partial_2\overline{F}(x_1,y_1)\cdot(y-y_1) + \nu(x,y) \qquad (4.29)$$

で定義する。点 y_1 における微分の定義と g の連続性により

$$\nu(x,y) = \nu(g(y),y) = o(|y-y_1|) \ \ (y\to y_1). \qquad (4.30)$$

$\overline{F}(x_1,y_1) = \overline{F}(x,y) = [0,\dots,0]^\intercal$ だから式 (4.29) より次が得られる：

$$\partial_1\overline{F}(x_1,y_1)\cdot(x-x_1) = -\partial_2\overline{F}(x_1,y_1)\cdot(y-y_1) + \nu(x,y). \qquad (4.31)$$

$L := \partial_1\overline{F}(x_1,y_1), K := \partial_2\overline{F}(x_1,y_1)$ とすると

$$L\cdot(x-x_1) = -K\cdot(y-y_1) + \nu(x,y). \qquad (4.32)$$

これにより

$$g(y)-g(y_1) = -L^{-1}\cdot K\cdot(y-y_1) + L^{-1}\cdot\nu(g(y),y).$$

評価 (4.30) と合わせれば

$$\left|g(y)-g(y_1)+L^{-1}\cdot K\cdot(y-y_1)\right| = o(|y-y_1|) \ \ (y\to y_1).$$

以上により

$$g'(y_1) = -L^{-1}\cdot K = -\left(\partial_1\overline{F}\big(g(y_1),y_1\big)\right)^{-1}\cdot\ \partial_2\overline{F}\big(g(y_1),y_1\big).$$

y_1 についてこれが連続であることはこの表示より明らか。

□

4.7 エピローグ

「パチュリーには悪いけど、陰関数の定理の証明は追い切れなかったよ。」

「そうね、関数の一様収束の話や、連続関数についての議論にある程度慣れていないと厳しいかもしれないわね」

「まあでも陰関数の定理を使えば Lagrange の未定乗数法ってのは簡単なんだな！」

線形代数の階数の理解も怪しかった人が「簡単だ」と言うか……！　と思ってしまったが楽しそうに金髪を揺らす魔理沙を見ていたら言い返す気もなくなってしまった。

「念のために強調しておけば、Lagrange の未定乗数法の条件 **(Lag3)** は『極値点であるための必要条件』に過ぎないわね。極値点の候補を絞り込むことには使えるけど、本当に極値点であることを確認するためには局所的に——つまり適当な近傍で——最大値や最小値になってることを確認しないといけないわ」

「どうやれば確認できるんだろう？」

「適用する問題の対称性などの特性をうまく利用して停留点が極値点であることを示すこともあるけど、一般には二階微分を使う方法があるわね」

「一変数でも、滑らかな関数の凸性は二階微分の符号で判断できたね」

「この記事では多変数関数 $f\colon \mathbb{R}^N \to \mathbb{R}^M$ の『一回微分』を $M \times N$ 行列として定義したわね。本質は行列であることじゃなくて、この行列が線形写像 $\mathbb{R}^N \to \mathbb{R}^M$ に対応してることなの。そして f の『n 階微分』は

$$\overbrace{\mathbb{R}^N \times \cdots \times \mathbb{R}^N}^{n\text{ 個}} \to \mathbb{R}^M$$

という多重線形写像になるわ」

「多重線形写像……？　ちょっと話が難しいなぁ」

「——したがって、『二階微分』は双線型写像になるわね。$M = 1$ の場合は二次形式論でやったように、双線型写像を正方行列で表す事ができるわね。この正方行列を**ヘッセ行列**といいます。滑らかな多変数の実数値関数の凸性はこのヘッセ行列を調べることでわかります。今回は連立一次方程式の復習で結構ページを取ったので、二次形式の復習が必要なヘッセ行列周辺の話題は全てカットしました」

「今みたいに制約が等式の場合だけじゃなくて不等式で制約が付いた場合はどうすればいいんだろう？」

「不等式制約問題については『カルーシュ・キューン・タッカー条件』というものがあるわね。でも話も結構長くなったし、そろそろお茶にしましょうか？」

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。

4.7.1 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] 熊ノ郷 準「偏微分方程式」 共立出版 (1978)
行列の表記方法はこの本で使われているものを参考にしました。
[3] 古屋茂「行列と行列式」 培風館 (1957)
[4] 佐竹一郎「線形代数学」 裳華房 (1957, 増補版 1973)
線形代数についての記述は上の二書を参考にしました。
[5] C. M. ビショップ「パターン認識と機械学習（上・下)」 丸善出版 (2012)
C. M. Bishop, “Pattern Recognition and Machine Learning” Springer(2006) の日本語訳です。この本の色々な箇所で Lagrange の未定乗数法が用いられています。
[6] ユルゲン・ヨスト「ポストモダン解析学」 シュプリンガー (2000)
Jürgen Jost “Postmodern Analysis” Springer(1998) の日本語訳です。この本でのバナッハ空間における陰関数の定理の証明をユークリッド空間の場合（こちらの方が議論が簡単です）に適用したものが本記事に載せた証明です。
[7] Walter Rudin ‘Principles of Mathematical Analysis’ (3rd ed.) McGraw-Hill(1976)
[8] 笠原皓司「微分積分学」 サイエンス社 (1974)
微積分についての定義や用語については上の二書を参考にしました。
[9] 岩堀長慶 (編)「微分積分学」 裳華房 (1983)
学部用の微積分の教科書ですが、多様体論の立場から Lagrange の未定乗数法を扱っています。
[10] 金谷健一「これなら分かる最適化数学 —基礎原理から計算手法まで—」 共立出版 (2005)
変数が３つで制約条件が２つの場合の図解が載っています。
[11] J. ソープ「微分幾何の基礎概念」 シュプリンガー (2006)
John A. Thorpe “Elementary Topics in Differential Geometry” Springer(1979) の日本語訳です。

応用上重要な Lagrange の未定乗数法ですが、軽めの微積分の教科書では２変数で制約条件が１つの問題だけを扱っていることがあります。本記事で述べたように、制約条件が２つ以上の問題にも Lagrange の未定乗数法は適用できます。ただし、その場合には３変数以上でなければなりません。このような問題を「図を使って説明する」場合には説明する側の技量だけでなく読者の深い幾何学的直観も要求するようです [10]。多変数の問題で制約条件が１つ (g とする）の場合、g が定める制約超曲面上の点 p における法線ベクトルが $\nabla g(p)$ であることを利用して、ある程度読者の幾何学的直観に訴えつつ数学的に満足の行く議論をすることもできます [11]。しかしながら、制約条件をさらに増やした場合にそのような流儀で議論できるのかどうかは定かではありません。

Lagrange の未定乗数法の最も自然で本質的な理解は、筆者の意見では「制約超曲面に視点を移すこと」から得られます。このような視点から物事を述べるための適切な枠組みの一つは多様体論です。つまり、滑らかな制約条件から「等高面」として多様体 V を考え、制約付き極値問題を、$f\colon \mathbb{R}^N \to \mathbb{R}$ の V への制限 $f\big|_V$ の制約なし極値問題に変換して論じるというわけです。微積分の教科書の中にも、Lagrange の未定乗数法を多様体論の枠組みで論じたものは存在します [9] が、学部生向けの教科書で多様体論にまで視点を広げて論じたものは少ないようです。

多様体論の立場から論じなくとも、陰関数の定理さえあれば、読者の「高度な幾何的直観」に訴えることなく Lagrange の未定乗数法を証明できます。本記事ではそのような立場で証明を行いました。そのついでに、陰関数の定理の証明も行いました。陰関数の定理の証明はいくつかの教科書を眺めて、一番わかりやすそうな Jost の本の証明を元に書いてみました。連続関数の重要事項、例えば『一点でゼロでなければその近傍でもゼロではない』だとか一様収束列の極限の性質などが登場するので、微積分の良い復習になると思います。

多変数における微分は本質的には関数を局所的に線形変換で近似する話なので、多変数の微積分

を学ぶ上で、線形代数をある程度理解していることが必要です。行列を扱う議論において、添字の海に溺れないようにするためにうまい略記法を導入することは重要です。行列を列ベクトルの集まりだとみなしたり、行列をブロック分けして扱うというようなことによって readability が保たれるのです。このような事を考慮して、本記事では一次連立方程式を例にして線形代数の簡単な復習をしています。多くの教科書では、一般の場合の一次連立方程式は斉次と非斉次に分けて扱われますが、この記事では斉次の場合の更に特殊なケースを扱いました。このケースは比較的議論の見通しが良く、後のほうで扱う陰関数定理とのつながりも良いからです。

会員名簿じゃなイカ?

`@master_q` (1 章)
: ATS たん、モフ! モフ!

`@nushio` (2 章)
: call-cc が不思議だ、というところから書き始めたのですが、分かってしまうと何が不思議だったのが自分でわからなくなって、うまく文章が書けず、残念です…。

`@osiire` (3 章)
: Alloy 最強！

`@dif_engine` (4 章)
: 次回は Haskell に戻る予定

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

「簡約(かんやく)！？ λ(らむだ)カ(か)娘(むすめ) 8」

2015 年 8 月 16 日 コミックマーケット 88 初版発行

発行：パスティヤージュ λ カ研究学院 / 参照透明な海を守る会
http://www.paraiso-lang.org/ikmsm/
ikmsm-authors@googlegroups.com

著者：@master_q, @nushio, @osiire, @dif_engine

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

印刷：ちょ古っ都製本工房 様 http://www.chokotto.jp/

パスティヤージュ λ カ研究学院/参照透明な海を守る会