Egison Journal Vol. 1

2019 年 4 月 14 日

目次

Egison 開発史

江木 聡志

Egison は 2010 年の 3 月，著者が卒業論文のためのプログラムを書いているときにアイデアを得て，2011 年 5 月，著者が修士課程の学生だったころに世に公開したプログラミング言語である．2012 年 3 月に修士課程を修了して 7 年を経た 2019 年現在も，開発を続けている．Egison には，そのパターンマッチ機能を中心に，多くの独自の機能が実装されてきた．これらの機能がどのように発見され，実装されていったのかこの開発史は紹介する．

また，著者が Egison の開発を，修士課程を修了して大学を離れたあとも現在まで，続けてこられたのは，多くの理解者・協力者と出会えたからである．この開発史では，今まで開発を助けてくださったこれらの方々への感謝も込めて Egison を中心に繰り広げられている人間ドラマの記述にも挑戦する．

この開発史を読んだ結果，Egison に入門したくなった読者がいてくれたら，うれしい．

1　2009/10-2010/03: Egison が生まれるきっかけとなった卒業研究

Egison のアイデアが生まれた学部 4 年の冬から話をはじめる．当時，私は，東京大学理学部情報科学科の萩谷研究室に配属されたばかりだった．

私は，自動で数学の理論を構築するプログラムを書きたいという夢をもっていて，卒業論文では自動で数学の定理を予想するプログラムについて書いた．このような研究の興味をもっていた理由は，高校生の頃，高木貞治『初等整数論講義』[13] を読んだ影響であった．この本は名著として有名な教科書で，ガウスの整数論についての研究がとてもわかりやすくまとまっている．私がとくに感銘を受けたのは，素数 p が 2 つの平方数の和で表されるための必要十分条件は，p が 2，または p が 4 で割って 1 余る素数であるという定理を，整数の範囲を $a+bi$ $(a,b \in \mathbb{N})$ という形の数にまで広げた素因数分解を考えることによって証明するアイデアであった．おおくの数学書がそうであるが，一度理解すると自然であるのに，発見するには超人的な思考が必要に思えるアイデアがこの本にはあふれていた．

このような歴史レベルの数学的発想にふれた高校生の私は，それらの発想の理由をすべて解明したいと考えるようになった．数学書は，基本的に数学の証明を軸に記述されている．しかし，証明を読んでも，証明で使われているアイデアの発想の源はみえないことがおおい．これらの発想の原因がすべて記述されていて，読者が数学者の思考を簡単にトレースできるのような数学書を書きたいと，高校生の私は考えていた．

このような志を持ちながら，東大に進学した私は，数学科に進もうと考えていた．しかし，教養学部の勉強に身がはいらず，東大の進学振り分けで私が数学科に進むのは不可能という状況に追い込まれた．(数学科への進学には平均点 75 点は必要であるのに，大学 1 年生が終わった時点の平均点が 43 点だった．) そのとき，偶然，理学部数学科のとなりの理学部情報科学科は進学希望者がほとんどおらず底割れしていることに気づき，運が良いことに情報科学科に進学できた．このときから，実は自分には数学よりプログラミングのほうが向いていると考えるようになり，プログラミングに本気で打ち込むようになった．

大学 2 年の冬に情報科学科に進学してからは，起きている間ずっとプログラミングについて考えている毎日だった．ターミナルに触れて 2 ヶ月の間にシェルを C 言語で実装したり，Scheme に触れたその日に徹夜で Scheme インタプリタを実装したり，その 1 年後にはフルスクラッチで Scheme コンパイラを実装したり，かなり充実した学部生活だった．

そしてついに卒業研究がはじまり，整数論とプログラミングをつなげようとしはじめたのだった．そこから，どのように Egison のアイデアが生まれたのか本節は語る．

1.1 難航した卒業研究

大学 4 年の冬学期に萩谷研究室に入って研究を始めた．整数論を自動で構築するようなプログラムについて研究したいという希望を萩谷先生に伝え，萩谷先生の指導を受けながら研究を進めた．

この卒業研究には苦労した．東大の情報科学科の卒業研究は期間が半年しかないため，基本的に既存の研究を少し拡張しただけの研究をすることが多い．私もそのように研究するため，萩谷先生から自動推論の論文を毎週紹介していただき読んでいた．ただ毎回紹介された既存研究に対して，この研究の拡張なんてしたくないといろいろな理由をつけながら文句を言っていた．そのおかげで，研究が一切進まないまま，卒業論文の締め切りまで残り一，二ヶ月という状態になってしまった．毎週，「うーん，なるほど」などと言いながら私の文句を聞いていた萩谷先生も，さすがにここまできると，「もう絶対このテーマでやってくれ」と頼みながら既存研究を紹介するようになってしまった．そのとき紹介していただいた文献が，記述論理の教科書 [9] だった．

自動推論についての既存研究の問題点と私が考えていたのは，推論の表現形式だった．論理式

による推論の表現は，論理自体の性質[*1]を論じるのには確かに便利だが，人間の推論を表現するのに適していないという認識を私は得た．たとえば，非常に初歩的な定理である素因数分解の一意性を論理式で記述すると，煩雑になる．このような表現形式の上で推論について考えても意味のある結果は得られないと私は考えた．記述論理はそのような私の指摘に対する最適案かもしれないということで萩谷先生が紹介してくださったのであろう．

記述論理の教科書を読みはじめて一週間後，四則演算，平方数を組み込みの概念として使う記述論理式を自動生成し，その中から 3000 以下の整数について成り立つ記述論理式のみを抽出することにより，数学の定理を予想するプログラムを走らせる実験をすることを思いついた．目標は，さきほどの 2 つの平方数の和で表される素数についての定理を予想することだった．時間が足りず，よい成果は出せなかったが，この実験が Egison のアイデアを得るきっかけになった．

1.2 Egison のアイデアを得た瞬間

卒業論文を提出し終えたあと，卒業論文のために書いたプログラムについて振り返っているときに Egison のアイデアを得た．このプログラムで，`mapWithBothSides` という関数を定義した．この関数は，記述論理の式の一部を同じ方法で書き換えて得られる式を生成するために定義された．`mapWithBothSides` を再帰関数としてふつうに記述すると，以下のように補助関数が必要となる[*2]．

```
mapWithBothSides f xs = mapWithBothSides' f [] xs
 where
  mapWithBothSides' f xs [] = []
  mapWithBothSides' f xs (y:ys) = (f xs y ys):(mapWithBothSides' f (xs ++ [y]) ys)
```

このような補助関数を定義した経験は，ほとんどの関数型プログラミングの経験者にあると思う．このような補助関数を必要とせずにもっとエレガントにプログラムが記述できるのではないかと考えはじめたことが，Egison の着想につながった．パターンマッチを拡張すれば，下記のようにエレガントに `mapWithBothSide` を定義できるというアイデアはすぐに思いついた．

```
mapWithBothSides f xs = matchAll xs as list something with
                          $hs ++ $x : $ts -> f hs x ts
```

「もし上記のようにプログラムを書くための研究がすでに提案されているとしたら，絶対にこのプログラミングスタイルは世の中の常識になっているはずである．それなのに，このプログラミングスタイルについて聞いたことがないのは，この着想は大発見であるからだ」という直感がこのときの私にはあった．この直感のおかげで，その後，長期間このアイデアの実現のために集中する

[*1] たとえば，健全性や完全性など．

[*2] 卒業研究のプログラムは Scheme で書いたが，ここでわかりやすさのために Haskell で記述した．

Egison による最強の CLI 活用術

ぐれさん

1　はじめに

こんにちは、Egison とシェルを愛するぐれさん (twitter:@grethlen) と言います。普段はクラウドの中のエンジニアとして仕事[*1]をする傍ら、技術雑誌や IT コミュニティで情報発信をしたり、Egison を含めた OSS 活動をしたりしています。

さて、本章では「Egison による最強の CLI 活用術」と題して、私が Egison を UNIX のシェル上で活用する際のテクニックをお教えしたいと思います。現状、筆者陣の中では研究畑とは一番遠い所で活動している人だと思うので[*2]、他の章とは少々テイストが異なり、実務寄りな内容になっております。Egison が初めてという方は、入門記事としてもぜひ楽しんで読んでいってもらえればと思います。

本章は本誌後半にある「付録 1. Egison 処理系の使い方」にあるインストール方法を事前にすませておき、かつ端末エミュレーターを開いておくと読みながら一緒に試すことができるのでおすすめです。

2　Egison でシェル芸

2.1　シェル芸とは

ところでみなさんは「シェル芸」という言葉をご存知でしょうか。下記の意味を持つ言葉です (引用元は本章末尾の参考資料 [8])。

「マウスも使わず、ソースコードも残さず、GUI ツールを立ち上げる間もなく、あらゆる調

[*1] なお、日本では働いてないです。そのため、原稿だけを江木さんに託しています。

[*2] 数年ぶりに LaTeX でがっつり文章を書いております。最後に書いたのは修士時代のジャーナルでした。

査・計算・テキスト処理を CLI 端末へのコマンド入力一撃で終わらすこと。あるいはそのときのコマンド入力のこと。」

———シェル芸人（本職はロボット工学者）[*3]・上田隆一氏 [9] による定義

要はコマンドをちゃちゃっと入力して目の前の仕事を終らせることです。あるいは入力されたコマンド自体のことを指すこともあります。また、「コマンド入力一撃」なので一行の処理（ワンライナー）を指して使われることが多いです。よく誤解されるのですがシェル芸は「古典的な UNIX コマンドのみを使ったテクニック」などということは断じてありません。目の前の仕事をコマンド入力一撃で終わらせることができれば、Windows で PowerShell を使ったテクニックも「シェル芸」と呼ばれることもあります[*4]。「シェル芸」の「シェル」はあくまでも一般的な CLI(コマンドラインインターフェイス) としての「インタラクティブシェル」であり、UNIX で動く「シェルスクリプト」に限定されるものではないためです。

私は「シェル芸」という言葉を考えた上田さんと一緒に活動をすることがあるので分かるのですが「使えるものは何でも使おう」という精神が根底にあります。2 ヶ月に一回、シェル芸勉強会という勉強会が開催されるのですが、この勉強会の参加者は `grep` や `sed` のみならず、`awk` や `perl`、`python`、`ruby`、さらには `bash` 以外のシェルである `zsh` や `fish` など、使えるものは何でも使っています[*5]。どれくらい何でもありかというと、Egison(`egison` コマンド) が使えるくらい何でもありです。

そう、Egison は「シェル芸」で使えるんです。現に私も何回も使ったことがあります。後述するように Egison は、他の一般的なコマンドでは決して一筋縄ではいかない処理を簡単に記述することができる、とても強力な武器となります。

2.2 一般的な UNIX のコマンドの弱点

広く使われる UNIX のコマンドを使っているときに筆者は、下記の 2 点が不便だと感じていました。

1. 標準入力からパターンを生成するフィルタコマンドが全然ない
2. 柔軟なパターン列挙をするコマンドがない

[*3] 本人に肩書きを確認したらシェル芸人が本職だと主張。

[*4] 特に「パワーシェル芸」と呼ばれます。

[*5] 「何でも使う」のは良いのですが、シェル芸で「何でもやろう」とするのはおすすめしません。シェル芸が向かない場面もたくさんあるので、そういう場合は素直にスクリプトを書きましょう。また、シェル芸の内容をスクリプトとして保存して何回も使い回すような方法もおすすめしません。ワンライナーは可読性を損ないやすいので、保存して継続的に保守する目的には向かない場合が多いです。「シェル芸」は基本的に「使い捨て」と思ってください。

そして、Egison をシェル芸で活用することの「うまみ」はその弱点を克服できる点だと考えています。以下、この 2 点について詳しく説明します。

標準入力からパターンを生成するフィルタコマンドが全然ない

古典的な UNIX のコマンドは組み合わせ列挙、パターンの生成が苦手です。例えばこんな課題があったとします。

問題 1: `A, B, C` という 3 つの文字を並び替えたときのパターンを全て端末に出力せよ。

いわゆる順列を列挙せよという問題です。手慣れたひとなら、ブレース展開を思いつくかもしれませんね。`echo {A,B,C}{A,B,C}{A,B,C}` というコマンドで ABC の 3 文字の重複を含んだ並び替えのパターンを全て列挙できます。そこから `grep` と拡張正規表現で重複を含むパターンを除くと、下記のようになります。

```
$ echo {A,B,C}{A,B,C}{A,B,C} | fmt -1 | grep -vE '(.).?\1' | xargs
ABC ACB BAC BCA CAB CBA
```

これを見て、「なんだ、シェル上でのパターン列挙は意外とできるじゃないか」と思った方もいるかも知れません。しかしながら、逆にこのようなことをするシェル (Bash) の一般的な機能はブレース展開くらいしかありません。ブレース展開は素晴らしい機能ではありますが、あくまでファイルやディレクトリ名を列挙する想定で利用されてきた機能であるため、少し用途が広がるとどんどん不便な点が出てきます。

UNIX のコマンドは、多くの場面でフィルタとして振る舞う、すなわち標準入力を使って新たな出力をすることが期待されます。例えば、上記の問題 1 を下記のように書き換えた途端にブレース展開の弱点が見えてきます。

問題 2: `echo A B C` コマンドにさらにパイプ (`|`) をつなげて、`echo` の出力から得られた文字の順列を全て端末に出力せよ。

この問題を先ほどと同じ方法で解くためには、標準入力の内容を更にブレース展開させる必要があります。そのためには `eval` あるいは更に `bash` コマンドを使って命令を再評価しなければいけません。これは、少々手慣れた人でなければ難しいです。下記は解答例ですが、途端に可読性が落ちる状態になりました。

```
$ echo A B C | sed 's/ /,/g;s/.*/echo {&}{&}{&}/' | bash | fmt -1 | grep -vE '(.).?\1' | xargs
ABC ACB BAC BCA CAB CBA
```

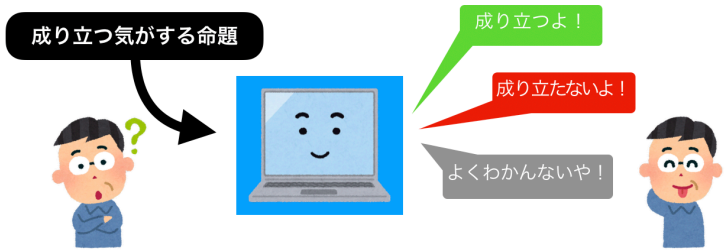
Egison で自動定理証明

西脇 友一 / わさびず

この記事では Egison を使って定理の自動証明を行います．内容はカタめですが，なるべく柔らかく説明していきたいと思います．どうぞお付き合いください．

1　自動証明って？

これから作成する物を絵にすると以下のような感じです．



つまり，命題をいれるとそれが正しいかどうかを返してくれる箱をつくります．

具体例を一つあげてみましょう．数学に触れたことのある方なら馴染みがあるであろう例を取り上げた方がいいと思うので，ここでは群を例にとってみます．

定義 1. **群**とは，集合 $|G|$，写像 $\cdot : |G| \times |G| \to |G|$，元 $e \in |G|$，写像 $(-)^{-1} : |G| \to |G|$ からなる四つ組 $G = (|G|, \cdot, e, (-)^{-1})$ であって，以下の 3 条件を満たすものである．

1. 任意の $|G|$ の元 x, y, z について，$(x \cdot y) \cdot z = x \cdot (y \cdot z)$ が成り立つ．
2. 任意の $|G|$ の元 x について，$x \cdot e = x$ が成り立つ．
3. 任意の $|G|$ の元 x について，$x \cdot x^{-1} = e$ が成り立つ．

なお，e を**単位元**，x^{-1} を x の**逆元**と呼ぶ．

普通の数学に慣れている人には少し見慣れない定義かもしれません．しかし，この定義のほうがコンピュータで扱うには何かと都合が良いです．条件 2 と条件 3 では単位元や逆元を右から掛けた場合の条件だけを要求していますが，通常の群の定義ではそれらに加えて左側から掛けた場合の条件も要請します．そう考えると上の定義は条件が足りないように見えるのですが，実はそれらは定理として導くことができます．

定理 1. *G を (定義 1 の意味での) 群とする．以下が成り立つ．*

(A) 任意の $|G|$ の元について，$e \cdot x = x$ である．
(B) 任意の $|G|$ の元について，$x^{-1} \cdot x = e$ である．

せっかくなのでこの証明をちょっと考えてみてください．かなりいい頭の体操になります．わかった人か諦めた人は次のページに進んでください．

...

証明できましたか？

Proof. まず (B) を示します．それを用いて (A) を示します．

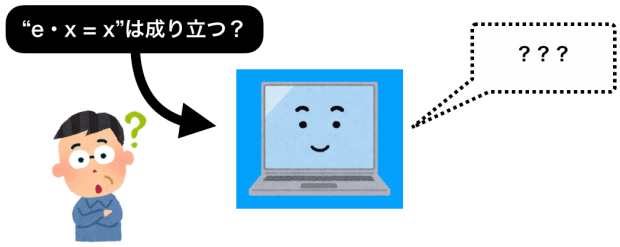
(B)
$$\begin{aligned} x^{-1} \cdot x &= (x^{-1} \cdot x) \cdot e \\ &= (x^{-1} \cdot x) \cdot (x^{-1} \cdot (x^{-1})^{-1}) \\ &= (x^{-1} \cdot (x \cdot x^{-1})) \cdot (x^{-1})^{-1} \\ &= (x^{-1} \cdot e) \cdot (x^{-1})^{-1} \\ &= x^{-1} \cdot (x^{-1})^{-1} \\ &= e \end{aligned}$$

(A)
$$\begin{aligned} e \cdot x &= (x \cdot x^{-1}) \cdot x \\ &= x \cdot (x^{-1} \cdot x) \\ &= x \cdot e \\ &= x \end{aligned}$$

□

これで無事，定義 1 が通常の群の定義と同値なことがわかりました．[*1]が，上の証明を読むとわかるとおり，**証明がだいぶテクいです**．パズルとしては楽しいですが，急いでいる人にはちょっと大変ですね．ではもし**この定理が全自動で解けたら**どうでしょうか．急いでる人にはとても助かるでしょうし，それになんだか楽しそうですよね．

ということで，今回の目標は**定理 *1* を自動で証明する**ことです．さきほどの絵に合わせて書くと以下のような感じです．



[*1] 山下くんが証明を少し改善してくれました．感謝です．

Egison の型システムの設計と型推論器の実装

a_kawashiro

この記事では Egison の型システムとその型推論器を設計した話をします.

1　型とは?

そもそもプログラミング言語における型とは何でしょうか? C++ では `int` や `string`, Haskell では `Int->Int` や `Maybe a` などがあります. 端的に言えば, 型とはプログラム中のデータを分類する種類のことです.

データを型を用いて分類する利点は大きく分けて 2 つあります.

1 つ目はバグの実行前検出です. 例えば, Haskell で数値計算を行うプログラムを書いて, その結果を表示することを考えましょう. Haskell に不慣れなプログラマは`"The result is " ++ n` のような整数型と文字列型を混同したプログラムを書いてしまうことがよくあります. Haskell の型システムはこのようなエラーを実行前に防いでくれます.

2 つ目は型のドキュメントとしての利用です. 例えば Haskell の `zip` 関数の型は `zip::[a]->[b]->[(a,b)]` ですが, これを見れば `zip` について何も知らない人でもその挙動を推測することができます. 更に型は `zip` の定義から自動的に導出されるため, 実装とドキュメントが乖離することがありません.

2　型の導出・検査

このような利点のある型ですが, どのように導出・検査するのでしょうか? プログラムへの型の導出は型付け規則の組み合わせで行われます. 型付け規則とはあるプログラム e に型 T を導出できる条件を定めるものです.

あるプログラム e に型 T が導出されるとは，型付け規則の組み合わせであってその結論が「プログラム e に型 T が付く」となるものが存在する，と定義されます．

例を使って解説しましょう．文字列に対する型付け規則は T-Str です．横棒の上が必要条件，横棒の下が結論です．s は任意の文字列を表します．Γ はとりあえず無視しましょう．T-Str を日本語に翻訳すると，「文字列には無条件で `String` 型を付けて良い」となります．

$$\frac{}{\Gamma \vdash s : \texttt{String}}\ \text{T-Str}$$

この規則を`"Hello"`に使うと `String` 型が導出できます．

$$\frac{}{\Gamma \vdash \texttt{"Hello"} : \texttt{String}}\ \text{T-Str}$$

型付け規則は複数組み合わせて使われることが多いです．`(lambda [$x] (b.+ x 10))` に対する導出木はこのようになります．

$$\dfrac{\dfrac{\dfrac{}{\cdots \vdash \texttt{b.+} : \texttt{[Int Int]} \to \texttt{Int}}\ \text{T-Var} \quad \dfrac{}{\cdots \vdash x : \texttt{Int}}\ \text{T-Var} \quad \dfrac{}{\cdots \vdash 10 : \texttt{Int}}\ \text{T-Num}}{\{\texttt{b.+} : \texttt{[Int Int]} \to \texttt{Int}\} +\!\!+ \{x : \texttt{Int}\} \vdash \texttt{(b.+ x 10)} : \texttt{Int}}\ \text{T-App}}{\{\texttt{b.+} : \texttt{[Int Int]} \to \texttt{Int}\} \vdash \texttt{(lambda [\$x] (b.+ x 10))} : \texttt{Int} \to \texttt{Int}}\ \text{T-Abs}$$

3　型推論器

`(lambda [$x] (b.+ x 10))` の型付け導出木を見ればわかるように，型付け導出木は非常に大きくなることが多く，プログラマが手で書くのは困難です．このため型付け導出木を自動で構築するソフトウェアが必要になります．このようなソフトウェアは型推論器と呼ばれています．

型推論器はプログラムを受け取り，その型を推論します．型付け導出木が構築可能ならそのプログラムの型を，構築不可能ならその旨を伝えるエラーを返します．

4　Egison の型システムの設計と型推論器の実装

Egison に型をつける作業は大きく 3 つに分けられます．

- Egison の構文の把握

- 型付け規則の設計
- 型付け規則に基づく型推論器の実装

Egison の構文を正確に定義するところから型システムの設計が始まります．まず Egison のインタプリタのコードを解読し，どのような構文要素が存在するかを調べ，必要不可欠なものだけを書き出します．こうして書き出した構文要素のリストが 7.1 節にあります．

次は型付け規則の設計です．Egison に必要なプリミティブな型を書き出し，その型付け規則を書き起こします．型付け規則はできるだけ既存のものを再利用しましたが、マッチャーやパターンに関する規則は当てはまるものがなかったので、筆者が新たに考案したものを使っています。書き起こした規則は 7.2 節にあります．

最後に型付け規則に沿って型推論器を実装します．型推論器は Hindley–Milner のアルゴリズム ([1] の 22 章) を使うことが多く，筆者もこのアルゴリズムを基に実装しました．

5 Egison 特有の型付け規則

この章では Egison 特有の型付け規則について細かく解説します。

各型付け規則に入る前に一つ注意点があります。通常型付け規則は環境 Γ，項 t，型 T の 3 項関係の組み合わせとして表現されますが，Egison では型をつけたあとに，パターン中の変数束縛によって環境 Γ が変化する場合が多いので，環境 Γ，項 t，型 T と変化した環境 Γ' の 4 項関係の組み合わせで表現します．$\Gamma \vdash t : T \Rightarrow \Gamma'$ は，「環境 Γ のもとで項 t に型 T がつき環境が Γ' に変化した」と読みます．このアイデアは Egison Workshop 2018 で`@gfngfn` さんに頂いたものです．この場をお借りして感謝します．

`match-all` に対応する T-MatchAll から解説していきます．この規則の特徴的な点はパターン p 内で束縛した変数を M_3 で使えるようにするために，p の型付け後の環境 Γ' を M_3 で使っている点です．

$$\frac{\Gamma \vdash M_1 : T_1 \Rightarrow \Gamma \qquad \Gamma \vdash M_2 : \texttt{(Matcher } T_1\texttt{)} \Rightarrow \Gamma \qquad \Gamma \vdash p : \texttt{(Pattern } T_1\texttt{)} \Rightarrow \Gamma' \qquad \Gamma' \vdash M_3 : T_3 \Rightarrow \Gamma'}{\Gamma \vdash \texttt{(match-all } M_1\ M_2\ \texttt{[}p\ M_3\texttt{])} : \{T_3\} \Rightarrow \Gamma}\ \text{T-MatchAll}$$

Γ' の有用性を確認するために，双子素数の各組の最初の素数だけを出力する次のようなプログラムを考えます．

```
(match-all primes
  (list integer)
```

Egison による因数分解

梅崎直也@unaoya

Egison による多項式の因数分解の実装をご紹介します。通常、中学校などでやるような因数分解は

$$x^2 + 6x + 5 = (x + 1)(x + 5)$$

のように整数の範囲で行うものですが、今回は素数 p で割ったあまりの世界、つまり**有限体**上での因数分解を行います。例えば $p = 3$ とすると

$$x^2 + 6x + 5 = x^2 + 2 = (x + 1)(x + 2)$$

のようになります。

整数の範囲での因数分解は、今回実装した有限体上での因数分解に加え、Hensel 持ち上げというアルゴリズムを組み合わせることで実装可能です。

この記事ではまず簡単に有限体やその多項式の計算を説明した後、因数分解のアルゴリズムと Egison での実装を紹介します。

1　有限体

最初に準備として、有限体とは何かについて、また有限体上の多項式の演算について紹介します。有限体はガロア体などとも呼ばれます。数学では体というのは通常の数と同じように四則演算が可能な数の集まりを言います。例えば有理数全体は有理数体、実数全体は実数体、複素数全体は複素数体と呼ばれます。これらの例は要素の個数が無限にあります（有理数も実数も複素数も無限にあります）が、要素の個数が有限であってその範囲内で四則演算が可能であるような数の集まりを考える事ができます。

例えば、$\mathbb{F}_2$ というのは要素が 2 個の集合 $\{0, 1\}$ に $1 + 1 = 0, 0 - 1 = 1$ とし、それ以外の計算は通常の数と同じように定めることで四則演算が可能になります。ここでは通常の数と同様に 0

で割ることはできないのでその点に注意してください。この四則演算は、整数を 2 でわったあまりのみに注目して四則演算を行うことに相当します。

同じように $\mathbb{F}_3 = \{0,1,2\}, \mathbb{F}_5 = \{0,1,2,3,4\}$ などもそれぞれ 3 で割ったあまり、5 で割ったあまりでの四則演算として定義できます。体の条件として 0 以外の全ての数について逆数が存在する事が必要です。$\mathbb{F}_3$ では $2 \times 2 = 4 = 1$ なので 2 の逆数は 2 であり、$\mathbb{F}_5$ では $2 \times 3 = 6 = 1, 4 \times 4 = 16 = 1$ なので、2 と 3 は互いに逆数、4 の逆数は 4 自身となります。より一般に、素数 p に対して $\mathbb{F}_p$ を $\{0,1,2,\ldots,p-1\}$ に p で割ったあまりとしての四則演算を定める事で、これは体になります。

一方で $\mathbb{F}_4$ ですが、これは同じような方法では体になりません。$\{0,1,2,3\}$ とすると、$2 \times 2 = 0$ となることから 2 の逆数は存在しません。代わりに $x^2 + x + 1 = 0$ の解を a, b として、$\{0,1,a,b\}$ とする事で体 $\mathbb{F}_4$ を定義する事ができます。詳細は最後の節に説明してあります。

有限体は要素の数が必ず素数の冪乗 p^e になります。また要素の数を決めると、一意的に決定されます。

今回は要素の数が素数の場合を使いますが、アルゴリズムの背景には素数冪のものがあることに注意しておきます。

1.1 $\mathbb{F}_p$ の演算

まずは有限体の演算を実装してみましょう。体なので、四則演算を定義します。Egison には `modulo` という整数の割り算での余りを返す関数があるので、それを使えばよいです。例えば足し算 `p.+` であれば次のように実装します。

```
(define $p.b.+
  (lambda [%x %y] (coef-mod (+ x y))))

(define $p.+
  (cambda $xs (foldl p.b.+ (car xs) (cdr xs))))
```

同様に引き算 `p.-`や掛け算 `p.*`および冪乗 `p.**`も定義します。Egison の割り算は分数になってしまうので、その点には注意が必要です。逆数を返す関数を `p.inv` と定義しておいて、これを用いて除算を定義します。

```
(define $p.inv
  (lambda [%x]
    (car (filter (lambda [$a] (eq? (p.* a x) 1)) (take p nats)))))

(define $p.b./
  (lambda [%x %y] (p.* x (p.inv y))))

(define $p./
```

付録 1. Egison 処理系の使い方

江木 聡志

本書を読んで，実際に Egison を試したくなった読者のために，Egison 処理系のインストール方法と基本的な使い方を紹介する．

1　インストール方法

まず，Egison 処理系をインストールする方法を紹介する．コンパイル済みのバイナルパッケージからインストールする方法と，ソースコードをコンパイルしてインストールする方法の 2 種類がある．パッケージからインストールする方法がおすすめであるので，そちらをメインで紹介する．

1.1　Mac OS X (Homebrew)

Mac OS X のユーザーは Homebrew を使うと以下のコマンドをターミナルで実行すると Egison をインストールできる．

```
$ brew update
$ brew tap egison/egison
$ brew install egison
```

1.2　Debian 系の Linux Distribution

dpkg または apt コマンドを使って以下のようなコマンドでインストールできる．

```
$ wget https://git.io/egison-3.7.14.x86_64.deb
$ sudo dpkg -i ./egison-*.deb
( or $ sudo apt install ./egison-*.deb)
```

付録 2. Egison プログラミング入門

江木 聡志

Egison は，(i) バックトラッキング[*1]による効率的な非線形パターンマッチ[*2]，(ii) ユーザーによるパターンマッチアルゴリズムの拡張，(iii) パターンの多相性[*3]，の 3 つの機能を同時に満たすパターンマッチ機能をもつプログラミング言語である．このような強力な表現力をもつパターンマッチのうえでは，これまで提案されていなかった新しいプログラミングのテクニックを考えることができる．このテクニックを使えば，たとえば，従来の関数型プログラミングでは再帰を使って定義されていた関数を，再帰を明示的に使わず，より直接的に定義しなおせる．このチュートリアルの目的は，これらのテクニックを紹介し，強力な表現力をもつ Egison のパターンマッチの魅力を読者に感じてもらうことである．

1　パターンマッチ指向プログラミングとは

「`map` 関数とは何をする関数であるか？」という質問に読者はどのように答えるであろうか？

多くの読者は，「関数とリストを引数にとり，リストの各要素に関数を適用して，それらの結果をリストとして返す関数である」と答えるであろう．「関数とリストを引数にとり，リストが空であった場合は，空のリストを返す．そうでない場合は，リストの先頭の要素に関数を適用した結果と，のこりの要素に再帰的に `map` 関数を適用した結果を `cons` した結果を返す関数である」と答える読者は少数であると思われる．

しかし，関数型プログラミングでは後者の説明による `map` 関数の定義が一般的である．たとえば，下記左側は Haskell による `map` 関数の定義である．一方，複数の結果をもつパターンマッチのための `match-all` 式をもつ Egison では，下記右側のように，“`<join _ <cons $x _>>`” のようなコレクションのすべての要素にマッチするパターンを使って，前者の説明により `map` 関数を

[*1] バックトラッキングとは，探索木を適切に枝刈りするアルゴリズム

[*2] 非線形パターンとは，パターン内に同じパターン変数が複数あらわれるパターン

[*3] 同じパターンコンストラクタ (たとえば，`cons`) を複数のマッチャーについて使えること

付録 3. Egison パターンマッチ論文の紹介

江木 聡志，西脇 友一

APLAS 2018 に採択された Egison のパターンマッチについての江木と西脇による共著論文 “Non-linear Pattern Matching with Backtracking for Non-free Data Types” (『正規形を持たないデータ型に対するバックトラッキング付き非線形パターンマッチング』)[*1]を本記事は紹介する．本論文の原稿は arXiv.org にても無料で公開されている．本合同誌を読んで興味を持てくださった読者はぜひ読んでほしい．

https://arxiv.org/abs/1808.10603

この論文を読みたくなる読者をすこしでも増やすために，この論文の概要を日本語訳したものをここに載せておく．

> *Non-free data type* とは正規型を持たないデータ型のことをいう．たとえば，多重集合は non-free data type である．なぜなら，多重集合 $\{a, b, b\}$ は同値であるが表現が異なる 2 つの表現形式 $\{b, a, b\}$ と $\{b, b, a\}$ を持つからである．**パターンマッチ**は，このようなデータ型を扱うための便利な構文を提供することが知られている．Non-free data type を扱うためのパターンマッチについて多くの研究と実用的なプログラミング言語への実装が提案されている．しかし，以下に我々が提案する “*Non-free datat types* **に対する実用的なパターンマッチのための** *3* **つの基準**” を満たしている既存研究がないことに我々は気づいた．この 3 つの基準とは，(i) 非線形パターンを効率的に処理するバックトラッキング，(ii) パターンマッチアルゴリズムの拡張性，(iii) パターンの多相性である．

[*1] https://doi.org/10.1007/978-3-030-02768-1_1