Theoretical Science Group

理論科学グループ

部報 310 号
── 駒場祭パンフレット号 ──

目　次

展示企画

企画紹介 スリザーリンク自動解答

semiexp

はじめに

3 年の semiexp です．駒場祭で展示する (予定の) スリザーリンク自動解答について紹介します．

スリザーリンクとは

スリザーリンク [1] は，ペンシルパズルの一種 (例えば「数独」のようなもの) です．ルールは，

1. 点と点の間にタテヨコに線を引き、全体で 1 つの輪っかを作りましょう。
2. 4 つの点で作られた正方形の中にある数字は、その正方形の辺に引く線の数を表しています。数字のない正方形には、何本の線を引くかわかりません。
3. 線を交差させたり、枝分かれさせたりしてはいけません。

というものです．(http://www.nikoli.com/ja/puzzles/slitherlink/rule.html より)

使い方

パズルのデータを与えての自動解答はインターネット上を探すと多数見つけることができますが，今回は問題の画像データをそのまま自動解答できるようにしてみました．

パソコンにカメラが接続されていて，カメラからの映像が画面に表示されるようになっています．映像にスリザーリンクの問題が写るようにして (問題の書かれた紙が用意されています [2])，

[1]「スリザーリンク」「数独」は (株) ニコリの登録商標です．
[2]問題は自作プログラムで自動生成させたものです

ウィンドウ上で Enter キーを押すと自動解答を行います．自動解答に成功すると，写真の問題部分に解答が書き込まれて表示されます．失敗した場合 (主にうまく写真中の問題を検出できなかった場合です) はその旨表示されます．

原理説明

内部動作は，「問題の認識」「自動解答」「解答を写真に合成」の 3 段階からなっています．

問題の認識

問題の認識も，以下のように複数の段階からなっています：

- 二値化
  入力されたカラーの画像を，扱いやすいように白黒の画像に変換する操作です．これは，OpenCV ライブラリを利用して行っています．
- 問題の「点」の検出
  問題の画像のどこに「点」があるかを検出します．これは，画像の黒いピクセルからなる縦横のつながり (連結成分) を検出した後，適切な大きさの連結成分を選び出し，さらに点の候補の中で最も「問題」を表していそうな部分のみを選び出すことで行っています．
- 「正方形の辺」の計算
  「点」が正しく検出できれば，「辺」は各点について，それに近い点たちを調べ，最もグリッドの一部らしくなりそうな辺の選び方を行うことで計算することができます．
- 「正方形」の計算
  「辺」がすべてわかっていれば，「正方形」たちの構造を調べるのは，グラフ理論における「双対グラフ」を求めることで行えます．実際には，グリッドグラフに特化して双対グラフを求めるアルゴリズムを使用しています．
- 数字認識
  各「正方形」が画像中のどこにあるのかがわかっているので，中にある数字の画素情報もわかります．ただし，「正方形」は画像中では傾いたり歪んだりしてそのままでは扱いづらいので，適切に変換することで，一定の大きさの (本当に正方形の) 画素情報を得ます．
  その後，数字認識器にかけてそこに含まれる数字が何なのかを認識します．問題全体が回転している可能性もあるので，数字の「向き」も同様に認識します．この数字認識器は，OpenCV に含まれる SVM (サポートベクターマシン) ライブラリを用いて実装しました．特徴量抽出など行わず，画素情報をそのまま SVM に与える単純な方法ですが，活字の $0, 1, 2, 3$ を認識するだけなので，高精度に認識できているようです．
- 問題全体の回転

この認識器は，回転した問題についても正しく向きを判定し，正しい向きの問題データを得られるようになっています．向き判定の情報としては，数字認識の際に計算した「数字の向き」を利用しています．0 を除く (180 度回転させても区別がつかないため) 各数字の向きについて多数決を行い，問題全体の向きを判定します．それに従い，問題全体を正しい向きまで回転させます．実際には，数字の向きはほとんどすべて一致することが多いようです．

自動解答

スリザーリンクの可解性の判定は NP-完全 であることが示されており，どんな問題にも通用するような多項式時間の解法はないと考えられています．しかし，現実の (人間が解くことを意図した) パズル問題では，大抵は「手筋」をうまく当てはめることを繰り返して解けるようになっているため，さまざまな手筋をプログラムに組み込むと多くの問題は解けるようになります．

展示のプログラムでも，このような手筋ベースの解法により問題を解いています．そのため，あまりに難しい問題は解くことができない場合があります．

解答の合成

問題データの上での各点が，画像でのどの点に対応するか，というのは問題認識の段階ですでにわかっています．そのため，解答が得られてしまえば，その対応データをもとに解答の線を適切に描画すれば，写真に解答を合成することができます．

機械学習

kivantium

はじめに

2 年生の@kivantium です。自分が最近やっている機械学習について簡単に紹介します。

機械学習

僕がメインで勉強しているのは機械学習と呼ばれる分野です。TSG でも去年・今年と機械学習を勉強する分科会（特定の分野に関心がある人が集まって勉強する TSG の活動単位）が開催されるなどメンバーの関心も高いようです。

機械学習はその名の通り機械に学習させる技術の総称でその応用範囲は多岐に渡ります。機械学習を学習の手法で大まかに分類すると

教師あり学習 入力データとそれに対する正解の出力データの組が与えられ、入力に対して正しい出力を行えるように学習する手法です。画像が与えられた時に何が映っているのかを判定するタスクや、過去の傾向から将来の予想を行うタスクなどが該当します。

教師なし学習 出力データが特にないデータが与えられ、そのデータの特徴・性質などを学習する手法です。データを特徴から N 個のグループに分類するクラスタリングなどが該当します。

強化学習 手に入れる報酬を最大化するために何をするべきかを学習する手法です。教師あり学習とは異なり何が正解なのかを報酬から判断する必要があります。ロボットの歩き方を自分で学習させるなどのタスクが該当します。

のようになっています。この分類はあくまで便宜上のもので、それぞれの手法が入り混じって使われることもあります。

教師あり学習は最も機械学習らしい学習手法で自分は主にこれを学んでいます。教師あり学習では SVM やニューラルネットワーク、Random Forest などのアルゴリズムがよく使われています。極端なことを言うと基本思想はどれも同じで、入力と出力をうまく近似する関数を作ることを目標に、どのような関数の形にするか関数をどのように最適化するかというところにアルゴリズムごとの違いが現れます。

自分が特に強い関心を持っているのは画像の教師あり学習です。画像認識は Deep Learning で

性能が飛躍的に向上したため面白い研究が次々と登場しており今後も注目していきたいと思っています。 最近は強化学習にも関心を持つようになり、今学期は強化学習分科会を開催して勉強しています。画像を入力として強化学習を行うことができれば面白いだろうなと思い研鑽を積もうとしているところです。

おわりに

機械学習は少し難しい理論が必要になることもありますが、出てくる結果が分かりやすいので書いていて面白いのが魅力だと思っています。理論上そうなることが分かっていても実際にプログラムを書いてきれいな結果が得られると何となく不思議な感じがするのも機械学習の特殊性のように感じます。短い文章でしたがこれを読んで機械学習に関心を持っていただければ幸いです。

T-rex Player

satos

概要

もし、あなたがブラウザとして GoogleChrome を使っているなら、インターネットに接続できないときに出てくる恐竜を見たことがあるかもしれません。そう、あいつです。あれはゲームになっていまして、迫ってくるサボテンを適切なタイミングでジャンプすることで飛び越えるゲーム、を遊ぶことができるのです。

操作はシンプルで、スペースキー/上十字キーでジャンプ/リスタート、下十字キーでしゃがみ、です。パソコン以外にも、お手持ちのスマホの GoogleChrome でもできるはずですので、やってみてください。

このゲームを、人間ではなくコンピュータに自動でやってもらおう、というのがこの企画です。

技術

まず、ゲームを自動操作する方法ですが、画面を認識したり、自動でキー入力を行ったりするのには、全て WindowsAPI を使っています。 FindWindowEx で Chrome のウィンドウハンドルを得て、 GetDC で画面のデバイスコンテキストを得て、 BitBlt で画面を別のビットマップにコピーして、 GetPixel で画面の色を得て、 SendMessage で適切なタイミングで WM_KEYDOWN を Chrome のウィンドウに送る。といった感じですね。

さて、後は、得られた画面から、跳ぶべきか否かを判断する AI を作るだけです。... どうやって作ったもんでしょうか。

まず、全部ルールを書きだして AI を作るテがあります。すなわち、「10mm 手前のここが黒くなったら、障害物が接近しているのでスペースを押す」とか、そういう規則を書きだしていって AI を作るのです。この方法だと、AI の改良がすぐに反映されるのでデバッグなどはやりやすくなりますが、改良してやらないと AI は強くなりません。そのため、AI をどんどん強くしていくために、人間が新たな改善点を次々と見つけて、それをどんどん実装していってやる必要があります。

そこでもうひとつ、AI に自分から学習していってもらうテがあります。AI は、「ゲームが終了したか否か」というのは確実に判断できるので、まず AI に適当に動いてもらいます。そこで、例

えば障害物が迫っていたのにジャンプせずにぶつかったりすると、「今ゲームオーバーになったのは、0.5 秒前にジャンプをしなかったためだな。ということは、0.5 秒前のこの画像みたいなのを見たときは、ジャンプする必要があるな」というように AI が判断して、次に来たときはちゃんと跳べるようにする、みたいなことを、AI 自身で学習していってもらうのです。この方法だと、AI を勝手に走らせておくだけでどんどん強い AI ができてくれるので、人間側としては非常にありがたいです。ただ、バグが直接的に表れないのでデバッグしにくかったり、AI が速く学習してくれるように改良する必要があったりするなど、AI をちゃんとつくるまでがなかなか大変です。

当日にどちらを展示できるかはわかりませんが [1] 、人間を超えた AI を作れるように頑張りたいと思います。

[1] 僕がプログラマ展示 (駒場祭までにプログラムが組みあがらなかった部員が駒場祭中にプログラミングするのを展示する企画) と化している可能性も大いにありますが...

メガネ男子かわいい

yamaguchi

概要

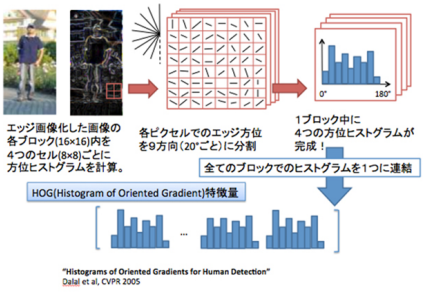
カメラに正対して近づくと、目を判定してメガネをかけてくれるプログラムです。キーボードの３９と w,e,r,t を押すことでかけるメガネを選択でき、s を押すとイメージが保存されます。

どうやって動くの？

dlib というライブラリを用いて、画像に対して HOG 特徴量を取って SVM を用いて目の認識を行っています。教師用の写真を自前で用意して一枚一枚目を囲って教師用データを作成し、dlib の検出器に突っ込んで判定してもらっています。

HOG 特徴量ってなぁに

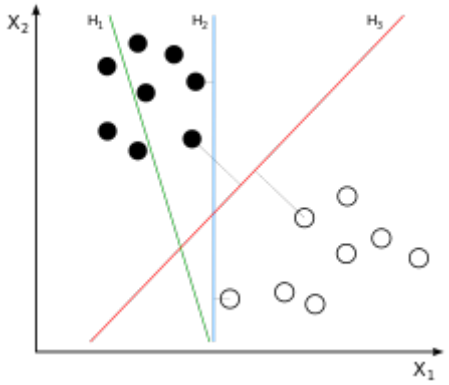
HOG 特徴量は入力画像から計算したエッジ画像に対して、各ブロック領域ごとの方位ヒストグラムを計算したものです。このサイト (http://news.mynavi.jp/series/computer_vision/022/) の画像がわかりやすかったので貼っておきます。



SVM ってなぁに

画像の分類には「君が目的の物体、君は違う子だね」と言ってくれる識別器が必要です。SVM(support vector machine) は識別器の一種です。SVM の特徴として、学習データの中で特徴空間上で最

も他クラスと近い位置にいるものを基準として、学習データと識別境界とのユークリッド距離が最も大きくなるような位置に識別境界を設置するマージン最大化があります。うぃきぺでぃあ (https://en.wikipedia.org/wiki/Support_vector_machine) から画像を取ってきました。H3 がマージンを最大にします。



プログラムについて

たぶん展示の近くにスクリプトを印刷して置いておくので見てください。検出器を読み込み、検出器ちゃんから目を四角で囲った座標が与えられるので、四角で囲われた目の中心の二点間の距離と生のメガネの画像のレンズの中心の二点間の距離が一致するようにメガネを縮小 or 拡大し、そのままだと初期位置（0，0）から動いてくれないので目の位置に合うようにずらしています。

性能について

誤判定はほぼ無いです。もっと性能が向上するように駒場祭中も教師用写真を収集したいです。（原稿を書いている時点で）謎のバグが取れなくて強制終了してしまうことがあるのと、近づくと座標変換がうまくいってないのかずれた場所にメガネがかかってしまうことが稀にあります。がんばってなおすぞい！

もう少し詳しいことが私のプログに書いてあるので、もしよかったら覗いていってください。

http://yamaguchi-1024.hatenablog.com/

Special Thanks

Cookies さん（なんでも聞いた）kivantium さん（アドバイスもらったり部誌見てもらったりした）

Python を使ったアニメーション

lip_of_cygnus

概要

周囲の状況に合わせてパソコン画面に映し出される物体の動きや色が変化します。カメラ周辺の音にも反応します。(予定)

技術

Video Capture で Web カメラから入力された画像を読み込みます。その後 PIL(Python Imaging Library) というライブラリを使ってアニメーションを生成します。時間ごとに各画素及び音の大きさなどの変化を記録しておき、変化の割合が多いと暖色系でかつ物体の動きが激しくなるようになり逆に変化が乏しいと寒色系で物体があまり変化しないようにプログラムを書いています。従ってカメラに映る景色や音が急に変化すると画面も大きく変化します。

おわりに

このプログラムでは多種多様な関数を使用しており、どの関数を利用して動きをつけようかという所が一番楽しく興味深い所です。元々映像のエフェクトには興味があったので、今後は 3D エフェクトやより複雑な動きを考えていきたいです。

一般記事

計算形而上学入門

nolze

> . . . このことが成されたならば、何か論争が生じても、二人の哲学者の間で議論することに、二人の計算者 *Computistas* の間で議論すること以上の必要はなくなる。ただペンを手に取って計算台 *abacus* に向かって座り（望むなら、友も呼びだしておいて）こう言うのだ—「計算しよう」、と。
>
> —ライプニッツ断片 (Gerhardt 1890)

本稿では Fitelson & Zalta の論文「『計算形而上学』への歩み *Steps Toward a Computational Metaphysics*」(2007) を紹介します[1]。

著者の一人で取り組みの中心人物である哲学者のザルタ E. Zalta はスタンフォード哲学百科事典 (SEP) の創設者で、今も主席編集者を務めているので哲学や論理学に関心のある人なら一度は名前を目にしたことがあるかもしれません。「計算形而上学 *Computational Metaphysics* (以下 CM)」のプロジェクトを簡潔に説明すると、彼が考案した抽象的対象論[2] *Theory of Abstract Objects* の枠組みでさまざまな形而上学的主張を公理化し (曰く「公理的形而上学」) 自動定理証明系に投げることで、主張の一貫性をチェックしたり帰結を見たりしようという研究の試みです。

Zalta の抽象的対象論

対象論とは

「対象論」は、端的に言えば、存在するかどうかを問わずに「対象」を認める立場の理論です。今日「マイノング主義」と呼ばれる思潮の直接の源流は、名前の通り 19 世紀末の哲学者で心理学者としても知られるマイノング A. Meinong に遡ることができます。マイノングは、ブレンターノ F. Brentano の「志向性」の理論に影響を受けつつ、心的な次元から出発して独自の対象論

[1] 修正などあれば `http://www.tsg.ne.jp/nolze/` 以下で公開します。ご指摘は `https://twitter.com/nolze` までお願いします。

[2] 単に対象論 *Object Theory, Theory of Objects* とも。他の「対象論」とのバランスから本稿では Zalta (1983) を背景に「抽象的対象論」で統一します。

Gegenstandtheorie を確立しました。マイノング主義においては、よく挙げられる例として「黄金の山」や「丸い四角」が「存在しない対象」として認められることになります。彼の後を継いだのがマリー E. Mally で、彼はマイノングの対象論を精緻化するとともに、その論理学的な分析によっても成果を残しました。マイノング-マリーの対象論を特徴づける原理として、以下の二つがあります。

- 対象の「かくあること *Sosein*」は、その対象の「存在しないこと *Nichtsein*」に依存しない (独立性の原理)
- 対象の「存在すること *Sein*」か「存在しないこと」のどちらかが対象について成り立っているが、存在は対象に本質的には関与しない (無差別性の原理)

とはいえ、そのようなよくわからないものを何でも—「丸い四角」のような矛盾したものは特に—どんな形であれ認めたりするのは「素朴に」やばい感じもします。ラッセル B. Russell はマイノングに批判的な立場を取るようになった一人で、Russell (1905) における強力な確定記述の理論の確立はマイノング主義に対する反論として大きな影響力を持ち、以後マイノング主義は有名どころだけでも Quine (1948) で隠喩的に叩かれる、「対象論は [...] 死に、埋葬され、復活することはないだろう」(Ryle 1973) と書かれるなどあまり評価されてこなかったようです。

しかしながら、こうした情勢にあって Parsons、Routley、Castañeda、Rapaport などなどの論者の尽力で徐々にマイノング主義の再評価と論理学的な形式化が進みます。Zalta の抽象的対象論もこの流れの中で (直接的には Parsons の影響で) 登場した理論のひとつです。他の理論に関しては、最近は日本でも Priest (2005) の邦訳が出たりしています。

もちろん、Zalta の抽象的対象論も含め対象論が存在を説明する唯一の理論であるというわけではなく、内在的な問題 (特に種々のパラドックス) がいろいろと指摘されているほか、マイノング主義を前提せずにこのような「対象」を説明できるとする Lewis (1978) のような議論もあります。

Zalta の抽象的対象論については、スタンフォード大学言語・情報研究センターの「形而上学研究室」の Web サイト [3] に情報が集約されています。書籍の Zalta (1983) が最も丁寧な文献ですが、形式的な内容については Web サイト内の Principia Metaphysica というドラフトに網羅的にまとまっています。

言語

- 対象変数と対象定数：$x, y, z, ...; a, b, c, ...$
- 関係変数と関係定数 [4]：$P^n, Q^n, R^n, ...;\ n = 0$ のとき $p, q, r, ...;\ n = 1$ のとき $P, Q, R, ...$
- 特別な 1 項関係：$E!$ (読みは「具体的 *concrete*」)
- 原子論理式：
  - 例化：$F^n x_1 ... x_n$

[3] https://mally.stanford.edu/

[4] いわゆる n 項述語ですが、述語に関しては Oppenheimer & Zalta (2010) で展開されているような微妙な背景がありそうなのでそのまま関係と訳します。

  - エンコーディング：xF

- 複合論理式：$(\neg\varphi)$...
- 複合項：
  - 確定記述：$\iota x\varphi$
  - λ 式：$[\lambda x_1 ... x_n \varphi]$
- 通常 *ordinary*：$O! =_{df} [\lambda x \Diamond E!x]$
- 抽象的 *abstract*：$A! =_{df} [\lambda x \neg\Diamond E!x]$
- 同一性：$x = y =_{df} x =_E y \vee (A!x \,\&\, A!y \,\&\, \Box\forall F(xF \equiv yF))$
- 関係の同一性：$F = G =_{df} \Box\forall x(xF \equiv xG)$

0 項関係は命題、1 項関係は性質 *property* と呼ばれます。例化 *exemplify* とエンコード *encode* は区別される「叙述の様態」で、Fx は「x は F (という性質) を例化する」というおなじみの叙述であるのに対し xF は「x は F (という性質) をエンコードする」と読みます。対象論的なのは後者であって、これは性質が対象を「決定する *determinieren*」というマリーの言い回しを Zalta 流に解釈したものです。後述の理論の公理から、抽象的対象だけが性質を「エンコードする」というあり方をします。よって例えば、「丸い四角」があるということを「丸さ」を R、「四角さ」を S として以下のように書くことができます。

$$\exists x(A!x \,\&\, \forall F(xF \equiv F = R \vee F = S))$$

論理

基本的には様相論理の体系 S5 な定領域の二階様相述語論理です。後で使うので、様相に関する公理だけ提示しておきます。

- 公理 K：$\Box(\varphi \to \phi) \to (\Box\varphi \to \Box\phi)$
- 公理 T：$\Box\varphi \to \varphi$
- 公理 5：$\Diamond\varphi \to \Box\Diamond\varphi$

これにエンコーディング・λ 式・確定記述に関する公理が追加されます。

- エンコーディングの論理：$\Diamond xF \to \Box xF$
- 同一性の論理：$\alpha = \beta \to [\phi(\alpha, \alpha) \equiv \phi(\alpha, \beta)]$
- λ 式の論理：β 簡約・η / α 変換
- 確定記述の論理：$\psi_z^{\iota x\varphi} \equiv \exists x(\varphi \,\&\, \forall y(\varphi_x^y \to y = x) \,\&\, \psi_z^x)$

理論の公理

- 通常対象の公理：$O!x \to \Box\neg\exists F(xF)$

- 抽象的対象の公理：$\exists x(A!x \,\&\, \forall F(xF \equiv \phi))$ (ϕ は自由変項を含まない式)

ハンズオン：イデアの計算理論

CM の取り組みの実例として、Fitelson & Zalta (2007) でも触れられている「プラトンのイデアの計算理論」[5] を取り上げます。この取り組みの直接の元ネタは、 Meinwald (1992) に即していわゆる「第三の人間」論を取り上げる Pelletier & Zalta (2000) であるものの、公理的形而上学としてのイデア論については Zalta (1983) に既にまとまっています。自動定理証明器に対する入力は CM の Web ページで Fitelson & Zalta (2007) に載っていないものも含め公開されていますが、以下ではなるべく抽象的対象論に忠実な形になるよう若干異なった定式化の仕方をしている場合があるので参照する際は注意してください。

Prover9/Mace4 のインストール

論文や Web ページにならって、本稿でも自動定理証明器 Prover9 とモデル探索器 Mace4 を使うことにします。公式ページ [6] から Linux/Mac OS X 向けのソースコードと Windows 向けのバイナリがダウンロードできます。コマンドライン版 (LADR) と GUI 版がありますが、以下ではコマンドライン版を前提します。

抽象的対象論の定式化

導出原理を基盤とする Prover9 で直接扱えるのは一階述語古典論理に限られるため、`Object(x)` や `Property(x)` といった述語を導入することで抽象的対象論の二階述語論理を多ソートな *many-sorted* 一階述語論理に書き換えるのが基本的な方針になります。

様相に関しても relational translation というテク (Ohlbach 1993, Manzano 1996) で Kripke モデルをソートによって表わします。[7]

- 公理 T：$\Box\varphi \to \varphi$

```
% AXIOM T, FROM OBJECT THEORY
Point(W).
```

$Point(w)$ は、直感的には「現実世界 W から世界 w へ到達可能である」と読むことができます。[8] 残るは λ 式の処理で、今回は使わないので割愛しますが、これも別のテクで表現できます。

[5] `https://mally.stanford.edu/cm/forms/`

[6] `http://www.cs.unm.edu/~mccune/mace4/`

[7] 自動定理証明界隈ではこの辺の技術は発達しているようで、例えば同じく自動定理証明系である SPASS は様相を内部的に変換してくれるらしいです。

[8] 抽象的対象論では、可能世界もひとつの対象として（メタ言語に対する）対象言語で定義されるので、*Point* というソートの導入はあくまで公理の形式的な変換として理解するのが正確なところです。

同一性の定義は具体的対象の同一性と抽象的対象の同一性の選言ですが、後で必要になる抽象的対象の同一性だけ定式化します。

- 同一性：$x = y =_{df} x =_E y \vee (A!x \,\&\, A!y \,\&\, \Box\forall F(xF \equiv yF))$

```
% ABSTRACT IDENTITY, FROM OBJECT THEORY
all x all y ((Object(x) & Object(y) & Ex1(A,x,W) & Ex1(A,y,W)) ->
  (x=y <-> (all F all w ((Property(F) & Point(w)) -> (MEnc(x,F,w) <-> MEnc(y,F,w))))))).
```

- エンコーディングの論理：$\Diamond xF \rightarrow \Box xF$

```
% LOGIC OF ENCODING, FROM OBJECT THEORY
all x all F ((Object(x) & Property(F)) ->
  ((exists w (Point(w) & MEnc(x,F,w))) -> (all w (Point(w) -> MEnc(x,F,w))))).
```

また論文や Web ページの略記に合わせて `Enc(x,F)` を `MEnc(x,F,W)` として定義しておきます。

```
% ENCODING ABBR., FROM OBJECT THEORY
all x all F ((Object(x) & Property(F)) -> (Enc(x,F) <-> MEnc(x,F,W))).
```

イデア論の定式化

原理　Pelletier & Zalta (2000) では、抽象的対象論における「x は抽象的である」を「x はイデア的である」に読み替えて [9]、以下の原理を設定しています。

- イデア的対象の原理：
- 内包原理：$\exists x(A!x \,\&\, \forall F(xF \equiv \phi))$ (ϕ は自由変項を含まない式)
- 同一性原理：$A!x \,\&\, A!y \,\&\, \forall F(xF \equiv yF) \rightarrow x = y$

内包原理は抽象的対象の公理と同じです。次の定義から内包原理の帰結に関する具体的な例が定式化できます。

- (必然的) 帰結 *entailment*：$G \Rightarrow F =_{df} \Box\forall x(Gx \rightarrow Fx)$

```
% ENTAILMENT, FROM THEORY OF FORMS
all F all G ((Property(F) & Property(G)) ->
  (Implies(F,G) <-> (all x all w ((Object(x) & Point(w)) -> (Ex1(F,x,w) -> Ex1(G,x,w)))))).
```

- 帰結による内包：$\forall G\exists x(A!x \,\&\, xF \equiv G \Rightarrow F)$

```
% ENTAILMENT INSTANCE OF COMPR. PR., FROM THEORY OF FORMS
all G (Property(G) ->
  (exists x (Object(x) & Ex1(A,x,W)
    & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
```

同一性原理は抽象的対象論における同一性の定義から導出できます。

[9] イデアの実在性を保持するため、Zalta (1983) はプラトニックな存在 Platonic Being $\overline{E}! =_{df} A!$ (!) を導入していますが、Pelletier & Zalta (2000) では実在性に直接言及しないで、読み替えによって通常の存在とプラトニックな存在の「差異を捉える」という戦略から出発しています。

```
formulas(assumptions).
  % AXIOM T, FROM OBJECT THEORY
  Point(W).
  % ABSTRACT IDENTITY, FROM OBJECT THEORY
  all x all y ((Object(x) & Object(y) & Ex1(A,x,W) & Ex1(A,y,W)) ->
    (x = y <-> (all F all w ((Property(F) & Point(w)) -> (MEnc(x,F,w) <-> MEnc(y,F,w)))))).
  % LOGIC OF ENCODING, FROM OBJECT THEORY
  all x all F ((Object(x) & Property(F)) ->
    ((exists w (Point(w) & MEnc(x,F,w))) -> (all w (Point(w) -> MEnc(x,F,w))))).
  % ENCODING ABBR., FROM OBJECT THEORY
  all x all F ((Object(x) & Property(F)) ->
    (Enc(x,F) <-> all w (Point(w) -> MEnc(x,F,w)))).
end_of_list.

formulas(goals).
  % IDENTITY, FROM THEORY OF FORMS
  all x all y ((Object(x) & Object(y) & Ex1(A,x,W) & Ex1(A,y,W)) ->
    ((all F (Property(F) -> (Enc(x,F) <-> Enc(y,F)))) ->  x = y)).
end_of_list.
```

イデア　次にイデアまわりを定義します。

- 「x は G のイデアである」：$Form(x,G) =_{df} A!x \,\&\, \forall F(xF \equiv F \Rightarrow G)$

```
% A FORM, FROM THEORY OF FORMS
all x all G (IsAFormOf(x,G) -> Object(x) & Property(G)).
all x all G ((Object(x) & Property(G)) ->
  (IsAFormOf(x,G) <-> (Ex1(A,x,W) & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F)))))))).
```

- G のイデア：$\Phi_F =_{df} \iota x(Form(x,G))$

先述の理由から Prover9 では確定記述を直接表現できないので、次のように変形します。

- G のイデア：$\forall x \forall G(\Phi_F \equiv A!x \,\&\, \forall y(Form(y,G) \rightarrow x = y))$

```
% THE FORM, FROM THEORY OF FORMS
all x all G (IsTheFormOf(x,G) -> Object(x) & Property(G)).
all x all G ((Object(x) & Property(G)) ->
  (IsTheFormOf(x,G) <-> (IsAFormOf(x,G)  & (all y (IsAFormOf(y,G) -> y=x))))).
```

いわゆるイデアの「分有」は「第三の人間」の議論を取り上げる上での中心テーマであり、Meinwald (1991) で提出された「他のものとの関係において *pros ta alla* (PTA) 」と「自分自身との関係において *pros heauto* (PH) 」の区別が取り入れられています。Pelletier & Zalta (2000) の説明を補って簡単に述べると、ある述語 (関係) F に関して、前者は「ある性質 F」についてそれを持っているという普通の個物における分有を、後者は性質 F をその本質として持っているというイデアにおける分有を意味します。ここではこの両者をそれぞれ Fx と xF に関連させて以下のように定義します。

- PTA 型の分有：$Participates_{PTA}(y,x) \equiv \exists F(x = \Phi_F \,\&\, Fy)$

```
% PTA-PARTICIPATION, FROM THEORY OF FORMS
all y all x all w ((Object(x) & Object(y) & Point(w)) ->
  (PartPTA(y,x,w) <-> (exists F (Property(F) & IsTheFormOf(x,F) & Ex1(F,y,w))))).
```

- PH 型の分有：$Participates_{PTA}(y,x) \equiv \exists F(x = \Phi_F \,\&\, yF)$

```
% PH-PARTICIPATION, FROM THEORY OF FORMS
all y all x ((Object(x) & Object(y)) ->
  (PartPH(y,x) <-> (exists F (Property(F) & IsTheFormOf(x,F) & Enc(y,F))))).
```

イデア論の定理

- 定理 1 (イデアの存在と唯一性)：$\forall G \exists! x Form(x,G)$

左右方向の含意でふたつの定理に分けて証明します。公理はすべて前提に入れても差し支えないのですが、見やすさと紙幅の都合から CM の Web ページにある入力を参考に証明が成り立つ範囲で適当に前提を省いています。

- 定理 1a (イデアの存在)：$\forall G \exists x Form(x,G)$

```
formulas(assumptions).
  % ENTAILMENT INSTANCE OF COMPR. PR., FROM THEORY OF FORMS
  all G (Property(G) ->
    (exists x (Object(x) & Ex1(A,x,W)
      & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
  % A FORM, FROM THEORY OF FORMS
  all x all G (IsAFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsAFormOf(x,G) <-> (Ex1(A,x,W) & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
end_of_list.

formulas(goals).
  % THEOREM 1A
  all G (Property(G) -> (exists x (Object(x) & IsAFormOf(x,G)))).
end_of_list.
```

証明器に投げてみましょう。

```
$./LADR-2009-11A/bin/prover9 < theorem1a.in
...
============================== PROOF =================================

% Proof 1 at 0.02 (+ 0.01) seconds.
% Length of proof is 33.
% Level of proof is 8.
% Maximum clause weight is 25.000.
% Given clauses 25.

1 (all G (Property(G) -> (exists x (Object(x) & Ex1(A,x,W) & (all F (Property(F) ->
  (Enc(x,F) <-> Implies(G,F)))))))) # label(non_clause).  [assumption].
```

```
2
...
105 Implies(c1,f2(f1(c1),c1)).
[resolve(103,a,71,d),unit_del(a,34),unit_del(b,58)].
109 $F.  [ur(75,a,34,a,b,58,a,d,103,a),unit_del(a,105)].

============================== end of proof ==========================
...

THEOREM PROVED
...
```

証明が見つかりました。同様にして以下の定理も証明できます。

- 定理 1b (イデアの唯一性) : $\forall G \forall x \forall y (Form(x,G) \& Form(y,G)) \rightarrow x = y)$

```
formulas(assumptions).
  % A FORM, FROM THEORY OF FORMS
  all x all G (IsAFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsAFormOf(x,G) <-> (Ex1(A,x,W) & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
  % IDENTITY, FROM THEORY OF FORMS
  all x all y ((Object(x) & Object(y) & Ex1(A,x,W) & Ex1(A,y,W)) ->
    ((all F (Property(F) -> (Enc(x,F) <-> Enc(y,F)))) ->  x=y)).
end_of_list.

formulas(goals).
  % THEOREM 1B
  all G (Property(G) -> (all x all y ((IsAFormOf(x,G) & IsAFormOf(y,G)) -> x=y))).
end_of_list.
```

以上から定理 1 が証明できました。この「イデアの計算理論」からは、どのような性質 G についてもイデアがただひとつ存在するという基本的なテーゼが問題なく導けます。

- 定理 3 (PTA 型の分有) : $Fx \equiv Participates_{PTA}(x, \Phi_F)$

PTA 型の分有は例化を使って定義されましたが、より直裁的に例化との同値関係も満たします。

```
formulas(assumptions).
  % AXIOM T, FROM OBJECT THEORY
  Point(W).
  % ENTAILMENT, FROM THEORY OF FORMS
  all F all G ((Property(F) & Property(G)) ->
    (Implies(F,G) <-> (all x all w ((Object(x) & Point(w)) -> (Ex1(F,x,w) -> Ex1(G,x,w)))))).
  % A FORM, FROM THEORY OF FORMS
  all x all G (IsAFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsAFormOf(x,G) <-> (Ex1(A,x,W) & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
  % THE FORM, FROM THEORY OF FORMS
  all x all G (IsTheFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsTheFormOf(x,G) <-> (IsAFormOf(x,G)  & (all y (IsAFormOf(y,G) -> y = x))))).
```

```
  % PTA-PARTICIPATION, FROM THEORY OF FORMS
  all y all x all w ((Object(x) & Object(y) & Point(w)) ->
    (PartPTA(y,x,w) <-> (exists F (Property(F) & IsTheFormOf(x,F) & Ex1(F,y,w))))).
end_of_list.

formulas(goals).
  % THEOREM 3
  all x all y all G ((Object(x) & Object(y) & Property(G)) ->
    (IsTheFormOf(x,G) -> (Ex1(G,y,W) <-> PartPTA(y,x,W)))).
end_of_list.
```

- 「定理」4 (PH 型の分有) : $xF \equiv Participates_{PH}(x, \Phi_F)$

PH 型はどうでしょうか。この定理については、Pelletier & Zalta (2000) では左から右方向の証明が与えられ、右から左方向は「読者への課題」となっています。怠惰な読者の代わりに Prover9 にやってもらいましょう。

```
formulas(assumptions).
  % TRIVIAL PREMISES
  Property(A).
  all x (Point(x) -> -Property(x)).
  % AXIOM T, FROM OBJECT THEORY
  Point(W).
  % ENTAILMENT, FROM THEORY OF FORMS
  all F all G ((Property(F) & Property(G)) ->
    (Implies(F,G) <-> (all x all w ((Object(x) & Point(w)) -> (Ex1(F,x,w) -> Ex1(G,x,w)))))).
  % A FORM, FROM THEORY OF FORMS
  all x all G (IsAFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsAFormOf(x,G) <-> (Ex1(A,x,W) & (all F (Property(F) -> (Enc(x,F) <-> Implies(G,F))))))).
  % THE FORM, FROM THEORY OF FORMS
  all x all G (IsTheFormOf(x,G) -> Object(x) & Property(G)).
  all x all G ((Object(x) & Property(G)) ->
    (IsTheFormOf(x,G) <-> (IsAFormOf(x,G)  & (all y (IsAFormOf(y,G) -> y = x))))).
  % PH-PARTICIPATION, FROM THEORY OF FORMS
  all y all x ((Object(x) & Object(y)) ->
    (PartPH(y,x) <-> (exists F (Property(F) & IsTheFormOf(x,F) & Enc(y,F))))).
end_of_list.

formulas(goals).
  % THEOREM 4
  all x all y all G ((Object(x) & Object(y) & Property(G)) ->
    (IsTheFormOf(x,G) -> (PartPH(y,x) -> Enc(y,G)))).
end_of_list.
```

なぜか証明が見つからなかったと思います。驚くべきことに (?) 実は Pelletier & Zalta (2000) が間違っていたのでした。同じ入力を Mace4 に投げると反例モデルを見つけてくれるので、それを手がかりに具体的な反例を構成することができます。

Fitelson & Zalta (2007) では左から右方向の証明を Prover9 で確認して「定理」4 を次のように弱めています。証明を試してみてください。

- 定理 4*：$xF \rightarrow Participates_{PH}(x, \Phi_F)$

おわりに

CM の Web ページには他にも、可能世界の定式化や、アンセルムスやライプニッツの形而上学に関する取り組みが掲載されています。自動定理証明器に対する入力には、本稿では参考文献での参照のしやすさと人間による読みやすさを重視して Prover9 固有のシンタックスを採用しましたが、汎用のものとしては TPTP シンタックスが普及しています。新しく出たより高速な証明器を利用することができるなどのメリットがあるので、CM の最近のプロジェクトはこのシンタックスを採用しているようです。

なぜより表現力のある Coq などの定理証明支援系ではなく自動定理証明系なのか？ という点に関しては、冒頭のライプニッツからの引用に表れているような普遍学 *scientia generalis* の理念を Zalta は意識したようです。夢があっていいですね。

「計算形而上学」は、Zalta の抽象的対象論で理論的には尽きていて「コンピュテーショナル」な部分が本質的な役割を担っているわけではないので、今日盛んに研究されている哲学と情報・計算の間の本質的な関係に迫る感じのやつを期待した人は若干がっかりしたかもしれませんが、期待されているポジションとしてはむしろ生物学に対するバイオインフォマティクスのそれに近いのかな？ と思います。コンピュテーションが本質的でないというのは利点でもあって、なーにがエンコードじゃと思うなら Zalta の抽象的対象論以外の理論を基盤にすることもできます。情報と哲学という組み合わせは一筋縄ではいかないところはありますが、踏み込んだ議論から一歩引いてもなお新しい発見のある領域なのではないかと思います。

参考文献

[1] Fitelson, B., & Zalta, E. N. (2007). Steps toward a computational metaphysics. *Journal of Philosophical Logic*, 36(2), 227-247.

[2] Leibniz, G. W. (1890). Die philosophischen Schriften von Gottfried Wilhelm Leibniz, hrsg. v. Carl Immanuel Gerhardt, Hildesheim, Olms, 7.

[3] Lewis, D. (1978). Truth in fiction. *American Philosophical Quarterly*, 37-46.

[4] Manzano, M. (1996). *Extensions of first order logic* (Vol. 19). Cambridge University Press.

[5] Meinwald, C. C. (1991). Plato's Parmenides.

[6] Meinwald, C. (1992). Good-bye to the Third Man.

[7] Ohlbach, H. J. (1993). Translation methods for non-classical logics: an overview. *Logic Journal of IGPL*, 1(1), 69-89.

[8] Oppenheimer, P. E., & Zalta, E. N. (2010). Relations versus functions at the foundations of logic: Type-theoretic considerations. *Journal of Logic and Computation*, exq017.

[9] Pelletier, F. J., & Zalta, E. N. (2000). How to say goodbye to the third man. *Noûs*, 34(2), 165-202.

[10] Priest, G. (2005). Towards non-being: The logic and metaphysics of intentionality.

[11] Quine, W. V. (1948). On what there is. *The Review of Metaphysics*, 2(1), 21-38.

[12] Russell, B. (1905). On denoting. *Mind*, 479-493.

[13] Ryle, G. (1973). Intentionality-theory and the nature of thinking. *Revue internationale de philosophie*, 255-265.

[14] Zalta, E. (1983). *Abstract objects: An introduction to axiomatic metaphysics* (No. 160). Springer Science & Business Media.

計算についての走り書き

satos

概要の前に

以下は、あくまで走り書きです。各部分での証明などは省かれておりますので、興味のある方は、下の参考文献やインターネットなどを参照してみてください。

概要

わたしたちの身の回りには、いろいろなプログラミング言語が存在しています。たとえば、C++,Java,Ruby,Haskell,Scheme などは、皆さんも使ったことがあるのではないでしょうか。

いままで述べた有名な言語以外にも、難解プログラミング言語 (esolang) と呼ばれる言語があります。

たとえば、「Brainf*ck」は、＋ - ＜ ＞ [ ] , . の 8 記号だけで構成された言語ですし、「Lazy_k」は、S K I ( ) の 5 記号だけですし、「Grass」は、w W v の 3 記号だけです。

これらの言語は、上で述べたメジャーなプログラミング言語と比べてとても貧弱に思えますが、実は例示した言語と同等の能力を持っています。いわゆる「チューリング完全」というやつです。

これらの言語と、その背景にある (のかもしれない) 理論について、ざっと紹介していきます。

チューリング完全

さて、上で挙げたような有名なプログラミング言語の間に優劣は存在するのでしょうか。例えば、Ruby だと記述できるけれども Haskell だと記述できないようなアルゴリズムは存在するのでしょうか。これに対する答えは、今のところ「ノー」だといわれています。すなわち、言語の能力に優劣はなく、ある言語で書かれたプログラムに対して、別の言語を使って、それと同様な動きをするようなプログラムを書くことができるということです。[1]

[1]ただ、言語によって「書きやすさ」や「計算速度」はある程度違い、科学計算をするのに適している言語や、ウェブページを作るのに適している言語、ゲームを作るのに適している言語など、いろいろな用途があるので、これだけ多くの

これが正しい (と考えられている) 理由として、「チャーチ・チューリングの提唱 (thesis)」というのがあります。これは、「すべての”計算”と呼ばれるような手順は、チューリングマシンで表現可能である」という提唱です。これはあくまで”提唱”であって、実際に定理として証明されているわけではありませんが、(というか、”計算”のちゃんとした定義がまだできていません)、これまでのさまざまな研究から、この提唱は正しいと信じられています。この提唱は、「プログラミング言語」にとどまらず、ある一定の手続きに従って計算するようなことは、すべてチューリングマシンで表現可能である、ということを言っています。

また、上に挙げた言語たちは、全て「チューリング完全」であることが示されています。ある言語、計算手順、手続きが「チューリング完全」であるとは、「その言語 (計算手順、手続き) がチューリングマシンと同等の計算能力を持っている」ということとして定義されています。つまり、あるチューリングマシンが与えられると、それと同等な計算をするようなその言語で書かれたプログラムを構成できる、ということです。このとき逆に、「チャーチ・チューリングのテーゼ」より、その言語で書かれたプログラムが与えられると、それと同等な計算をするようなチューリングマシンを構成できることが保証されます。

すなわち、プログラミング言語は、ある程度の計算能力を得ると、これ以上の機能を追加しても、他の機能の糖衣構文にしかならない、という状況に陥るのです。こうして、チューリング完全性を保ちつつ、言語の機能をそぎ落としていくことによってできるのが、最初に言及した難解プログラミング言語 (esolang) です。

Brainf*ck

必要最小限の機能だけを残した手続型言語[2] です。

8 記号の + - ＜ ＞ [ ] , . は、それぞれ、C 言語でいうところの

’+’ array[index]+=1;
’-’ array[index]-=1;
’＞’ index+=1;
’＜’ index-=1;
’[’ while(array[index]!=0){
’]’ }
’,’ array[index] = getchar();
’.’ putchar(array[index]);

に対応しています。

これだけで C 言語などと同じ能力を持っているのか、と驚くかもしれませんが、たとえば ”for” や ”if” は ”while” で代用できますし、 加減乗除は、1 を足し引きすることの繰り返しでどうにか

言語が世の中にはあるのです。いろいろな形の工具があるようなもんです。

[2] C++とか Java とか Ruby とかは、この系列

なります。また、関数呼び出しや再帰関数などは、同様の計算をする while ループに展開できることが知られています。[3] そのため、これだけの言語機能で十分なのです。

さて、これをさらに抽象化した計算概念として、チューリングマシンというのがあります。

チューリングマシン

「チャーチ・チューリングのテーゼ」で 言及された機械です。有限オートマトンに、記憶機能として一本の長いテープをつけたものです。テープ上の文字を書き換えていくことにより、計算を行います。メモリの上限がなく、1 命令ごとに、次にどの命令に跳ぶかを判断するようなプログラム、みたいな感じですかね。[4]

Lazy_k と Grass

必要最小限の機能だけを残した関数型言語 [5] です。

ふつうの言語には、例えば、数字や文字列、ペアやリストなどのいろいろな”型”がありますが、Lazy_k と Grass は、どちらも型が関数しかありません。[6] すなわち、すべて「関数を受け取って関数を返す関数」だけでプログラムを組むのです。これらは「型無しλ計算」という理論にもとづいており、これだけの機能で十分強力な計算ができます。

型無しλ計算

kuromunori さんの記事をご覧ください。(まるなげ)

チューリング完全なほかのひとたち

・帰納的関数

計算能力の定式化としては、先ほど挙げた「チューリングマシン」と「型無しλ計算」以外に、

[3]実際、アセンブリ言語などでは、引数や返り値をスタックに積むことによって関数を表現しています
[4]http://snuke.main.jp/turing/ チューリングマシンを、実際にゲームとして遊べるサイトです。
[5]Haskell とか Scheme とかは、この系列
[6]Grass の方は、正確には、in や out に'w' を succ したもの以外を渡すとエラーで落ちますが

「帰納的関数」を使った定式化が、ものの本にはよく書いてあります。どちらかといえば、プログラミング言語よりも、数学的証明のほう寄りの定式化です。[7]

(1) で原始的な関数を定義した後、(2),(3),(4) の規則に従って関数どうしを組み合わせることによって、計算したい複雑な関数をつくります。

(1)

1. ゼロ関数 ($zero(x) \stackrel{\text{def}}{=} 0$)
2. 後者関数 ($succ(x) \stackrel{\text{def}}{=} x+1$)
3. 射影関数 ($P_i^n(x_1, x_2, \cdots, x_n) \stackrel{\text{def}}{=} x_i$)

としたとき、1.2.3. は帰納的関数である。

(2) n 変数帰納的関数たち $g_1, g_2, \cdots, g_m$,m 変数帰納的関数 h に対して、

$f(x_1, x_2, \cdots, x_n) \stackrel{\text{def}}{=} h(g_1(x_1, x_2, \cdots, x_n), g_2(x_1, x_2, \cdots, x_n), \cdots, g_m(x_1, x_2, \cdots, x_n))$

として定義した n 変数関数 f は、帰納的関数である。(関数合成)

(3) n 変数帰納的関数 g,n+2 変数帰納的関数 h に対して、

$f(0, x_1, x_2, \cdots, x_n) \stackrel{\text{def}}{=} g(x_1, x_2, \cdots, x_n)$

$f(k+1, x_1, x_2, \cdots, x_n) \stackrel{\text{def}}{=} h(k, f(k, x_1, x_2, \cdots, x_n), x_1, x_2, \cdots x_n)$

として定義した n+1 変数関数 f は、帰納的関数である。(これを、原始帰納法による定義と言ったりします)

(4) n+1 変数帰納的関数 g に対して、

$f(x_1, x_2, \cdots x_n) \stackrel{\text{def}}{=} g(y, x_1, x_2, \cdots x_n) = 1$ となるような、最小の y

として定義した n 変数関数 f は、帰納的関数である。(これを、最小化による定義と言ったりします。y=0 から始めて、y=0,1,2,3 ... について g(y, ...) の値を順々に計算していき、g(y, ...)=1 となるような y が見つかった時点で、その y の値を返す、みたいな感じで計算します。ここで、g(y, ...)=1 となる y が見つからない場合、計算は一種の無限ループに陥ります。)

このように「帰納的関数の集合」を定義してやると、全ての計算可能な関数は、帰納的関数の集合に含まれます。

例えば、add(足し算) は、

$add(0, b) \stackrel{\text{def}}{=} P_1^1(b)$

$add(a+1, b) \stackrel{\text{def}}{=} succ(add(a, b))$

と定義できます。[8]

このため、「帰納的関数」もチューリング完全となります。

・ライフゲーム

「グライダー」と呼ばれる物体どうしを適切にぶつけることによって、AND,NOT などが構成できます。「ライフゲイムの宇宙」では、自己複製が可能であろうということのみが触れられてい

[7] 帰納的関数に似た「RECU 関数」という関数の集合がペアノ算術で表現しやすいので、不完全性定理とかの証明にもかかわってくるらしいです。

[8] まあ、正確には $add(a+1, b) \stackrel{\text{def}}{=} succ(P_2^3(a, add(a, b), b))$ ですが、「与えられた」引数を使わずに捨てることは、射影関数を使えば簡単にできるので、ここでは省略します。

ましたが、実際に自己複製するパターンが最近発見されました。また、「メタピクセル」という任意のルールのセルオートマトンをシュミレートするパターンも発見されました。

・タグシステム, 循環タグシステム, マルコフアルゴリズム

文字列を一定の規則に従って書き換えていくことにより計算する仕組みです。

・ルール 110

一次元的セルオートマトンのうちのひとつ。曼荼羅のようなきれいな模様を作ってくれますが、なんと計算も可能だそうです。

・ラグントンの蟻

AND,NOT などの論理回路を構成できるので、チューリング完全となるそうです。

・2-3 チューリングマシン

3 状態、2 記号だけでできた万能チューリングマシンです。

「全てのチューリングマシンを、適切にエンコーディングして入力してやることによってシュミレートすることができる」、ようなチューリングマシンのことを、「万能チューリングマシン」と言います。さて、このとき、どれだけ少ない状態数 (Brainf*ck でいう、プログラムのソースコード長さ)、記号数 (テープ上に書ける記号の種類数) で万能チューリングマシンを作れるだろうか、ということが気になってきます。これに対して、3 状態、2 記号だけで万能チューリングマシンを構成することができる、ということが最近証明されました。2 状態、2 記号だと無理だということが分かっているので、このチューリングマシンが最小の万能チューリングマシンのうちの一つ、ということになるのです。

計算可能性と不可能性

世の中の、たいていの真偽の定まる事柄は、プログラムを使って判別、もしくは計算することができますが、ときたま、「計算が不可能な関数」(すなわち、その関数を計算するプログラムを書けないような関数) というものが存在します。[9]

ここで前もって言っておくべきこととして、「プログラムを引数に取るようなプログラム」を作ることができる、ということがあります。これは、全てのプログラムに対して、それに対応する番号をつけることができるので、その番号をプログラムに引数として渡すことができるためです。[10] [11]

さて、計算が不可能な関数の有名な例として、「停止性判定」があります。これは、「与えられたプログラムが停止するかどうかを判定する関数」を計算する停止するプログラムは存在しない、

[9] 本項内では、「プログラム」を「ソースコードやλ項、帰納的関数などの、具体的に構成できるもの」として、「関数」を「その定義 (例えば、入力値の総和を返す、とか、入力によらずすべて 42 を返す、とか) は決まっているが、実際に「プログラム」を構成できるかどうかはわからないもの」として書いていきます。

[10] たとえば、プログラムファイルをアスキーコードの 256 進数とみなして、対応する数字を番号とする、とか。

[11] ちなみに、「”あるプログラムの番号”と”そのプログラムに渡す引数”を受け取ると、そのプログラムをシュミレートしてくれるようなプログラム」(スクリプト言語の eval にあたるようなもんです) を構成できることが証明されています。

というものです。停止しないプログラムとしては、「while 文が無限ループになってしまっている」とか「いつまでたっても再帰呼び出しが終わらない」とか、λ 式の場合、「β 簡約が終わらない」とか、帰納的関数の場合、「最小化のところで、g(y,...)=1 を満たすような y が存在しない」だとかが、それぞれの定式化において存在します。[12] プログラミングをする身としては、無限ループが発生するのは勘弁してほしいので、「無限ループ判定器」みたいなのが欲しいのですが、残念ながらそんなものはないことが証明されます。[13]

証明をおおざっぱにします。

もし、プログラム halt(f,x) $\stackrel{\text{def}}{=}$「「f が 1 つの引数をとるプログラムの番号で、f に x を与えると無限ループに陥る」なら 0, そうでないなら 1 」
なる halt(f,x) (halt(f,x) は、必ず停止するとします）が存在すると、
s(f) $\stackrel{\text{def}}{=}$「$halt(f,f)=1$ なら無限ループに陥り、$halt(f,f)=0$　なら停止するようなプログラム」というような プログラム s(f) が作れます。ここで、 s(s)　について考えると、 s(s) は、「無限ループに陥る」か、「一定時間で停止する」かのどちらかです。

さて、
(1) s(s) が無限ループに陥るとすると → s(s) の定義より、halt(s,s)=1 である → halt(s,s) の定義より、「s(s) は、無限ループに陥る」ではない → 「s(s) が無限ループに陥る」という最初の仮定と矛盾
(2) s(s) が停止するとすると → s(s) の定義より、halt(s,s)=0 である → halt(s,s) の定義より、「s(s) は、無限ループに陥る」 → 「s(s) が停止する」という最初の仮定と矛盾

よって、(1),(2) のどちらにせよ矛盾してしまいます。こうなったのは、最初に、「プログラム halt(f,x) が存在する」としたことが間違っていたからです。

よって、「与えられたプログラムが停止するかどうかを判定する関数」 を計算する停止するプログラムは存在しないことが分かります。 (証終

さて、「与えられたプログラムが停止するかどうかを判定する関数」を計算する停止するプログラムは存在しないことが分かりましたが、もうちょっと一般化して、「与えられたプログラムが 1 を返すどうかを判定する関数」とか、「与えられた 2 つのプログラムが同じ答えを返すどうかを判定する関数」などが存在しないことも証明できます。[14]

---

[12] そもそも、無限ループに陥らないようにプログラミング言語を定義してやればよいではないか、という作戦も考えられ、例えば、帰納的関数の場合、(4) の最小化を定義から外してやった「原始帰納的関数」というプログラムの集合があったりするのですが、(最小化が無ければ、確実に計算が停止します) この場合、逆にチューリング完全ではなくなってしまうのです。(「アッカーマン関数」という確実に停止する関数があるのですが、この関数を表現するような「原始帰納的関数」は存在しないことが証明されています) このような、チューリング完全であって、必ずすべてのプログラムが停止するようなプログラミング言語の定義は、いまのところ見つかっていません。

[13] まあ、「無限ループ判定器」が存在すると、例えば、「 $x^n + y^n = z^n$ となるような $x, y, z, n \geq 3$ の組を列挙していき、見つかればそれを出力して停止する」みたいなプログラムを組んだあと、そのプログラムを「無限ループ判定器」に投げることによってフェルマーの最終定理の証明ができてしまうので、なかなかえらいことになってしまうのですが。

[14] 例えば、「 1 を返す」のほうは、$e(f,x) \stackrel{\text{def}}{=}$「「$f$ が 1 つの引数をとるプログラムの番号で、$f(x)$ は 1 を返す」なら 0, そうでないなら 1」なる プログラム e(f) の存在を仮定したあと、いったん $q(f,x) \stackrel{\text{def}}{=}$「$f(x)$ をシュミレートした後、1 を返す」という q(f,x) を作ります。ここで、$r(f,x) \stackrel{\text{def}}{=}$「[番号 p に対応するプログラム]$(t) = q(f,t)$ と、なるような番号 p を求め (この計算は停止します)、$e(p,x)$ を返す (この計算も停止します)」という プログラム r(f,x) が作れます。(つまり、p(x) は、q(f,x) に対応しています) すると、halt(f,x) が、 r(f,x) に対応していることがわかると思います。後は先ほど

もっと一般的に、「与えられたプログラムの挙動に関する自明でない性質を判定するような関数」を計算する停止するプログラムは存在しない、という定理 (ライスの定理) があります。

あと、他の「計算が不可能な関数」として、

・「与えられた文字列のコルモゴロフ複雑性 (その文字列を出力するようなプログラムのうち、最小のプログラムの大きさ) を計算する関数」

・「ビジービーバー関数 (2 記号 n 状態のチューリングマシンのうち、一番多くの 1 を出力して停止するチューリングマシンが出力する 1 の数、を返す関数)」

などがあります。どちらも、いわば「「○○を出力する最小文字数 (状態数) のプログラム」を求める関数」なのですが、その関数に対応するプログラムが作れるとすると、そのプログラム自身の文字数 (状態数) が返り値よりも小さくなってしまうような引数が存在する、ということを利用して、存在しないことの証明をしています。

参考図書など

アンダースタンディング・コンピュテーション
ライフゲイムの宇宙
計算理論とオートマトン言語理論
プログラミングによる計算可能性理論
計算可能性入門

の証明を使えば矛盾が導けます。

編集後記

★ TSG へお越しくださってありがとうございます.
★ 今年度 compiler になりました lip_of_cygnus です.
★ 今年はどうやら記事が多いみたいです.
★ 編集作業も結構大変で、ディスプレイの壊れた PC でやっています.
★ 1 年間よろしくお願いします.

理論科学グループ　部報 第 310 号

2015 年 11 月 21 日　発行
発行者　佐藤聡太
編集者　秋本慎弥
発行所　理論科学グループ
〒 153–0041　東京都目黒区駒場 3–8–1
東京大学教養学部内学生会館 313B
Telephone: 03–5454–4343

理論科学グループ部報 第 **310** 号

— 駒場祭パンフレット号 —

2015 年 **11** 月 **21** 日

THEORETICAL SCIENCE GROUP