
簡約!?
人力娘
9

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. 九冊目なんて、こんなの絶対おかしいよ！
B. 僕達の役目はね。関数型言語とイカ娘の設計、実装、原理、応用に関する妄想を抜き取って、文章に変えることなのさ

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、コミックマーケット 80 での「簡約! λ カ娘」、コミックマーケット 81 での「簡約!? λ カ娘 (二期)」、さらにコミックマーケット 82 での「簡約!? λ カ娘 (算)」、に続く、さらにさらにコミックマーケット 83 での「簡約!? λ カ娘 4」、に続く、もーさらにさらにコミックマーケット 84 での「簡約!? λ カ娘 Go!」、に続く、そしてコミックマーケット 85 での「簡約!? λ カ娘 Rock!」、コミックマーケット 86 での「簡約 λ カ娘 巻の七」、コミックマーケット 88 での「簡約!? λ カ娘 8」、に続く、九冊目の関数型イカ娘の本でゲソ。コミック連載終了に安部真弘氏に「おつかれさまでした!」の気持ちをこめて、関数型言語で地上を侵略しなイカ!

この本の構成について

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

第1章

λ力娘探索ファイナル

— @ark_golgo

Contents

1.1 プロローグ

「1 勝 4 敗か…。内容を見る限り、『よく 1 勝した』と言うべきなのじゃろうが…。」
「そうでゲソね…。まさかこれ程とは思わなかったでゲソ…。」
大きなニュースになったので、読者の皆様もおそらく何のことかご存知であろう。そう、
「イ・セドル対 AlphaGo 戦」
のことである。λカ娘 6 で述べた通り、大半の研究者が、
「囲碁で AI が人間トップを超えるにはまだまだかかる。」
と思っていた。しかし、2016 年 3 月、人間のプレイヤーとして世界トップクラスのイ・セドルが、Google が作った囲碁 AI の AlphaGo と 5 番勝負をして、わずか 1 勝しか出来なかった。研究者の予想が見事に裏切られた訳である。
「しかし、お主は確か以前、モンテカルロ囲碁ソフトの強さは頭打ちになっていて、それを打ち破るには『攻め合い』や『死活』などの細い一本道を読み切る能力が必要だと言っておったな。それをモンテカルロ囲碁に組み込むことはとても難しいことで、誰も成功していないとも。この AlphaGo を作った DeepMind の人達は、この短期間にそれを成し遂げてしまったのかね？」
「私は AlphaGo の論文を読んだでゲソが、そういう訳ではないでゲソ。爺さんも対局の内容を見て気づいたんじゃなイカ？ 実は、AlphaGo は、論文を読む限りは読みの力は大したことがないと思われるでゲソ。どうやら、大局観だけでイ・セドルに圧勝してしまったらしいんでゲソ。」

「確かに、対局の内容を見て何となく分かってはおったが…。ただ、すまんが儂は AlphaGo の論文を読んで理解するだけの英語力も AI に対する知見も持ち合わせておらん。よければ、AlphaGo とはいかなる物なのか、儂に説明してくれんかね？」

「勿論、喜んで説明するでゲソ。ただ、AlphaGo の凄さを理解するためには、囲碁 AI の歴史を知らなければならないでゲソ。それに、λカ娘　6 を読んでいない読者もいると思われるでゲソ。さらに言えば、囲碁のルールを知らない読者も居ると思われるでゲソ。だから、長くなるでゲソが、まず AlphaGo の概要を述べた後、囲碁のルールとゲーム AI の基礎及び囲碁 AI の歴史も説明して、改めて AlphaGo の詳細を述べるでゲソ。長くなるでゲソが、付き合ってくれなイカ？」

「勿論構わんよ。茶と菓子を用意したから、休みながらで構わん。ゆっくり聞かせてくれ。」

　さて、AlphaGo とはいかなる物なのか、λカ娘にじっくり説明してもらおう。

1.2 AlphaGo 概要

「AlphaGo は、凄くざっくり言ってしまえば、モンテカルロ木探索と Deep Learning を組み合わせた囲碁 AI でゲソ。モンテカルロ木探索はλカ娘 6 でも説明したでゲソが、この後でも説明するでゲソ。Deep Learning は、階層の深いニューラルネットワークのことで、AlphaGo の場合は、画像認識の世界で有効性が確認されていた Deep Convolutional Neural Network(DCNN) と言う種類のものを用いているでゲソ。DCNN が囲碁 AI に有効らしいということは 2015 年頃から分かっていたでゲソが、AlphaGo は潤沢なマシンリソースを利用することで、DCNN を用いた高精度の候補手絞り込みと、DCNN を用いた高精度の局面評価を可能にしたでゲソ！」

1.3 囲碁のルールおさらい

「再度囲碁のルールを説明するでゲソ。囲碁のルールは細かいものまで説明していると結構大変なので、今回の記事に関係ある部分だけ説明するでゲソ！ 以下の 4 つを把握しておけば十分でゲソ。」

1. 黒白交互に盤の交点に打つ。1 手目は黒から (図 1.1)。[*1]

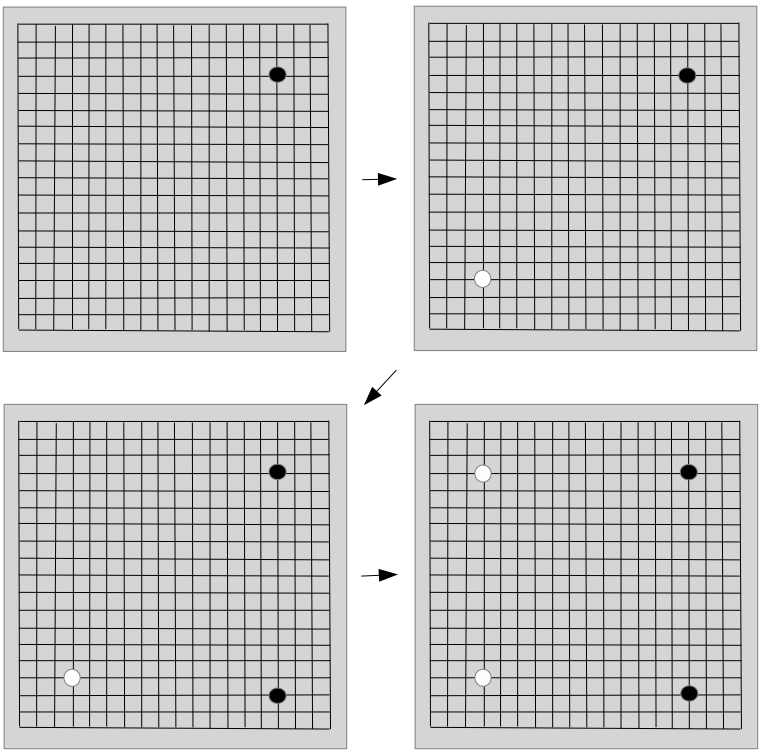

図 1.1: 交互に打つ

[*1] ただし置き碁 (ハンデ戦) の場合は黒があらかじめ石を何個か置き、その後白から打ち始める。

2. 相手の石の四方を囲むと取れる (図 1.2)。

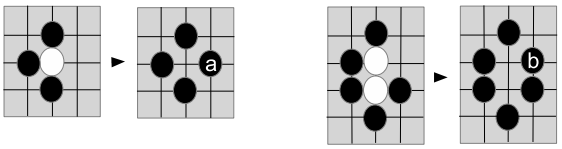

図 1.2: 左図の状態から、a で示した石を黒が打つと、真ん中の白石は上下左右囲まれるので取られる。b のように、複数石あっても同様である。

3. 自分から石を取られるような手は打てない (自殺手の禁止) が、それ以外は空いている場所のほぼどこでも打てる。そのため、自由度が非常に大きい。ただし一見自殺手に見えても、先に周りの相手の石を取れる場合は打てる (図 1.3)(『コウ』という概念があって、厳密には打てない場所もあるが、今回は説明しない)。

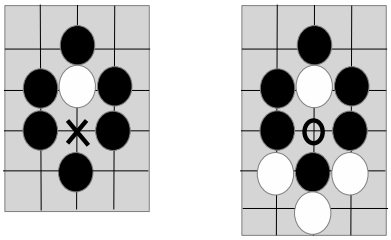

図 1.3: 白は×には打てないが○には打てる

4. 相手より大きな陣地を囲えば勝ち (図 1.4)。陣地とはそれぞれの石または盤端で四方を囲まれた石の無い領域のこと。図は数えやすいように 9 路盤 (練習用の小さな盤) で示している。黒と白の丸は実際の石で、四角はそれぞれの陣地を示す。四角の所には実際には何も置かれていない。この場合、黒の陣地は 36 目 (目(もく)とは地の単位)、白の陣地は 23 目で黒の 13 目勝ち。

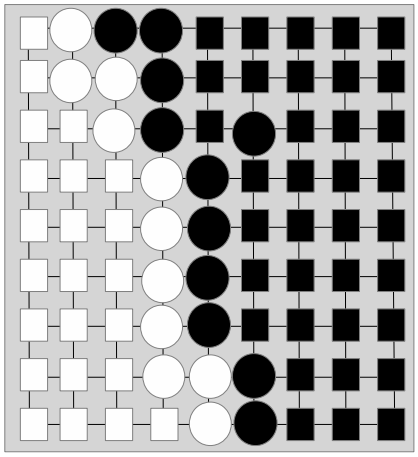

図 1.4: ■が黒の陣地で 36 目。□が白の陣地で 23 目。黒 13 目勝ち

1.4 AND-OR 探索

「AND-OR 探索、Min-Max 探索、α-β 探索をまず順に説明するでゲソ。この節では AND-OR 探索を、次の節で Min-Max 探索を、その次の節で α-β 探索を説明するでゲソ。これは、囲碁 AI では上手く行かなかった手法でゲソが、ゲーム AI の基礎とも言える部分なので、是非とも理解してほしいでゲソ！」

「ゲームには色々あるでゲソが、囲碁はゲーム研究の世界では『二人零和有限確定完全情報ゲーム』と呼ばれるものでゲソ。『二人で戦い』『片方が勝てば他方は負け』『状態空間が有限であり』『サイコロのような不確定な要素が無く全て自分の意志で決めることが出来』『伏せられたカードのような隠れた情報が無い』ゲームはこう呼ばれるでゲソ。将棋、チェス、リバーシなどもこれに分類されるでゲソ。バックギャモンや麻雀やポーカーは違うでゲソ。この『二人零和有限確定完全情報ゲーム』の AI において人間の『読み』に相当する機能は基本的に『AND-OR 木』というものがベースになっているでゲソ。ここでは盤面が小さな Tic-Tac-Toe で説明するでゲソ。勿論、Tic-Tac-Toe も『二人零和有限確定完全情報ゲーム』でゲソ。」

「三目並べとか○×とか呼ばれているゲームのことじゃな。」

「図 1.5 を見るでゲソ。図 1.5 は最大あと 4 手で決着する、Tic-Tac-Toe のある局面 (a) からの変化を網羅したものでゲソ。(a) での手番は×でゲソ。×は次にどこに打てるでゲソか ?」

「上、左、左下、下の 4 マスじゃな。」

「その通りでゲソ！ その 4 つの可能性があるでゲソ。人間がこのゲームをする時には、それぞれの箇所に打ったらその後何が起こるか考えるんじゃなイカ？ 例えば、上のマスに打ったとするでゲソ。そうすると、局面は (b) のようになるでゲソ (上の局面から下の局面への線は、この一手打ったことによる変化を示しているでゲソ)。

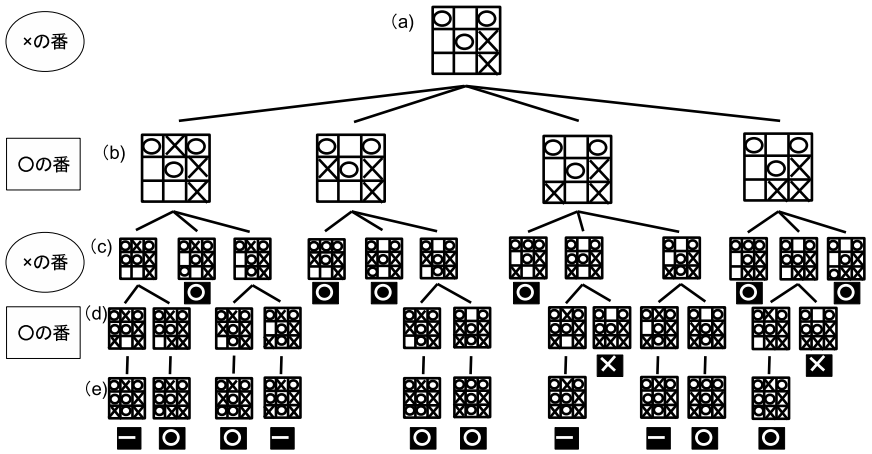

図 1.5: Tic-Tac-Toe の変化を網羅したゲーム木。ノードの下にあるマークは、どちらが勝ったかを表す。「-」は引き分けを表す。

局面 (b) では相手、○の手番でゲソ。この局面 (b) ではまだ決着がついていないのでゲームを続行するでゲソ。○は左、左下、下の三か所に打てるでゲソ。
○がもし左のマスに打ったとしたら……次は局面 (c) になるでゲソ。×がその次に左下に打つと局面 (d) に、そうすると次に○は下にしか打てなくて、下に打つと局面 (e) になって、引き分けになるでゲソ。これが一つの可能性じゃなイカ！
もちろん、可能性を一つ考えるだけではダメでゲソ。例えば、局面 (b) で○は左ではなく左下に打つこともできるでゲソ。そうすると……○の勝ちでゲソ!つまり私の負けじゃなイカ！　局面 (b) は○の手番だから○の好きな場所に打つことが出来るので、当然一番○にとって良い、つまり×にとって一番悪い手、『左下』を打つでゲソ！　この可能性を見逃して最初の局面 (a) で×を上に打ってしまうと、次に○に左下を打たれて負けてしまうでゲソ。
だから、局面 (a) では上、左、左下、下のすべての可能性を、その次の局面 (b) でも左、左下、下のすべての可能性を……という風に、すべての可能性を網羅したものが図 1.5 でゲソ！　」
「なるほどだんだん思いだしてきたわい。そして、勝ちか負けか、どこに打てばいいのか分かるために、『AND-OR 木』という考え方を使うんじゃったな。」
「そうでゲソ！ 図 1.6 を見るでゲソ！　これは、さっきの網羅図に対応する『AND-OR 木』でゲソ！
　AND-OR 木を作る目的は、『自分が勝てる手』を探すことでゲソ (見つかったら当然それを打ちたいからでゲソ)。単純化すると、ゲームには『双方最善を尽くした時自分が勝てる局面』と『双方最善を尽くした時自分が負ける局面』しかないので、これを仮に『True(真)』と『False(偽)』で表すことにするでゲソ。引き分けがあるゲームについては、引き分けをどうするかは難しい問題でゲソが、今回は引き分けは自分の勝ち True とするでゲソ！
図 1.6 の場合は、自分が×だとして、×が勝てる局面　=True にたどり着けるような手を探すでゲソ。自分の手番の時 X は、次の一手で True になるような局面が一つでもあれば、他の手の結果がどうなろうと関係なく、True になる手を選べるので、X も True になるでゲソ。どの手を選んでも False になる場合は X も False になるでゲソ。これは要するに次の局面の集合に対する『OR』演算と同じでゲソ！

相手の手番の時 Y は、逆に、次の一手で False になるような局面が一つでもあれば、他の手の結果がどうなろうと関係なく、False になる手を選んでくるので、Y も False になるでゲソ。どの手を選んでも True になる場合は Y も True になるでゲソ。これは要するに次の局面の集合に対する『AND』演算と同じでゲソ！

そうやって、図 1.7 のように、下から順に、つまり確定しているところから順に、各局面の勝敗を判定していくことが出来るでゲソ！　こうすると、今の局面で勝ちか負けか分かる上、勝ち (True) の場合は子ノードから True になっている物を選ぶのが最善手であることが分かるでゲソ！　自分が負け (False) の場合は子ノードはすべて False なのでどこ打っても負けになるでゲソ…」

「なるほどのう。そういえば、全く同じ局面が何度も登場しているようじゃが、それを全て調べるのは無駄だという話もあった気がするのう。例えば、引き分けになっている局面は全て同じじゃな。まあこれは AND-OR 木の末端じゃから、調べなおしても大した負荷にはならんかもしれんが、探索の途中にも同じ局面があるのう。これはかなりの負荷になる場合もあるのではないかね？」

「おっしゃる通りでゲソ！　実は、いくつかのゲームではゲーム木中で全く同じ局面に到達する場合があるでゲソ。そのような場合、何度も探索し直すのは無駄なコストがかかるので、『置換表』というものに一度探索した局面とその結果を記録しておくのが普通でゲソ。」

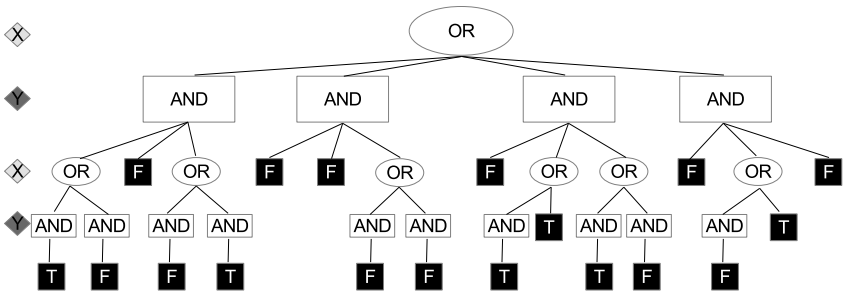

図 1.6: 図 1.5 のゲーム木を「AND-OR 木」で表したもの

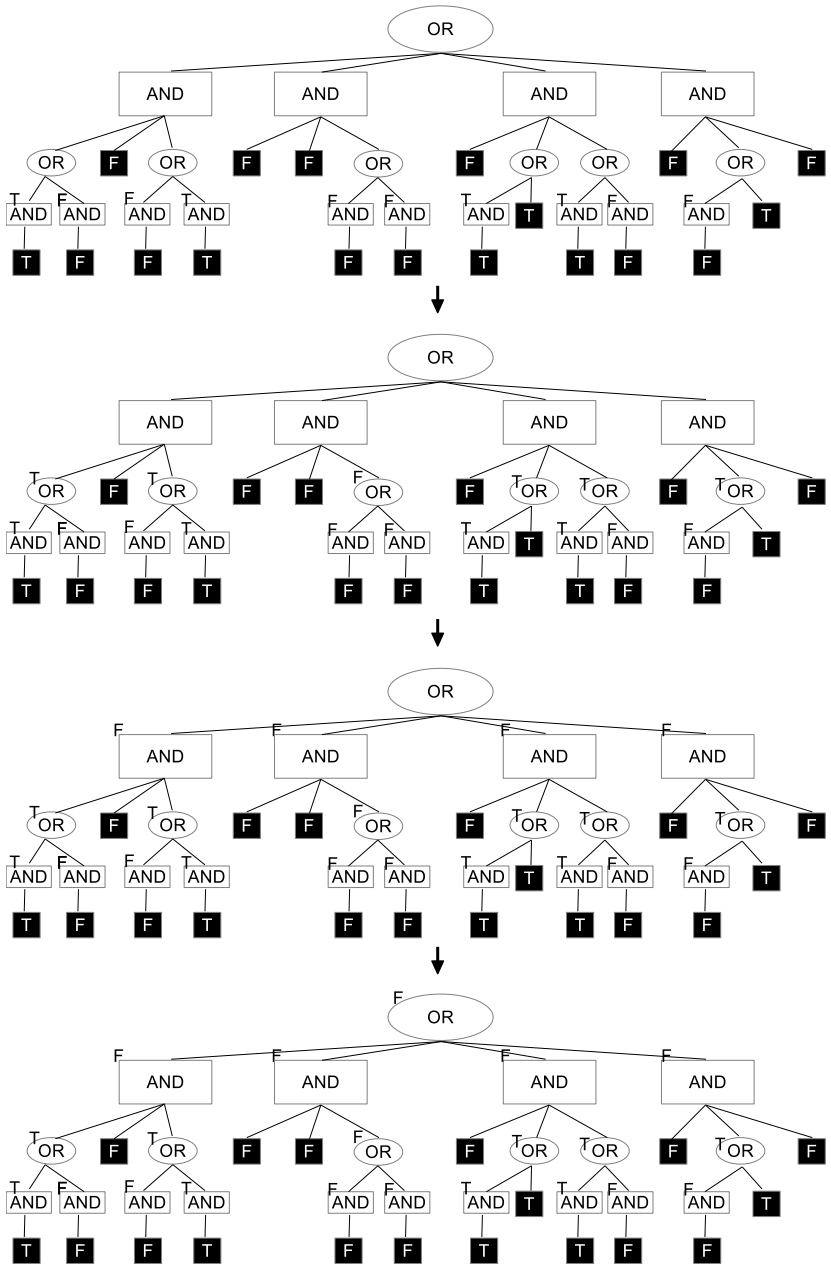

図 1.7: AND-OR 木のルート局面の勝敗が決まる様子

1.5 Min-Max 探索

「上記の AND-OR 探索では、ゲームの決着がつくまで探索できることを想定していたでゲソ。でも、現実に楽しまれているゲームは、選択肢の幅が広すぎて、とてもゲームの決着がつくまで探索は出来ないことが多いでゲソ。あるいは、例えばリバーシならスコアは-64〜64 の整数になるでゲソが、勝つか負けるかだけではなく決着時に出来るだけ大差で勝ちたい/負けるにしても小差にしたいという場合もあるんじゃなイカ？ そのような場合は、『評価関数』と『Min-Max 探索』を組み合わせて使うでゲソ。順に説明するでゲソ！

評価関数はゲームの途中でどちらがどれだけ有利かを計算する関数で、自分の手番が有利なほど大きな値 (通常は正の値) を、相手の手番が有利なほど小さな値 (通常は負の値) を返すでゲソよ。例えば、絶対負け = −10000、絶対勝ち=10000、完全に互角=0、やや有利=1000 のようになるでゲソ。何せゲームの途中で判断するので、評価関数を作るのはなかなかに難しく、それだけで本何冊も書けそうでゲソ。でもここでは、人間のプレイヤーがゲームの途中で『勝ちそう！ 』とか『負けそう… 』とかを判断しているのをプログラムにやらせるというイメージだけあればいいでゲソ！ これを、AND-OR 木で最後まで読み切った時の True/False だった部分の代わりに使うでゲソ。

そして、さっきは True と False だったので、AND-OR で良かったでゲソが、評価関数を使うともう AND-OR では計算できないでゲソ。そこで Min-Max でゲソ！ Min-Max 探索は、先ほどの『OR ノード』が『Max ノード』に、『AND ノード』が『Min ノード』になったものでゲソ。

自分の手番の時 X は、次の一手で好きな手を選べるので、次の局面の中から出来るだけ大きな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Max』演算と同じでゲソ！

相手の手番の時 Y は、逆に、次の一手で相手が好きな手を選べるので、次の局面の中から出来るだけ小さな評価値を選んだ時の評価値になるでゲソ。これは要するに次の局面の集合に対する『Min』演算と同じでゲソ！

図 1.8 が、そうやって下から順に評価値が決まっていく様子を示したものでゲソ！ これで、現在の局面の評価値、つまりどれくらい有利/不利そうか、が分かる上、Max ノードなので子ノードのうち一番評価値が大きかった子ノードに対応する手を打つのが最善手であるというのも分かるでゲソ！ 」

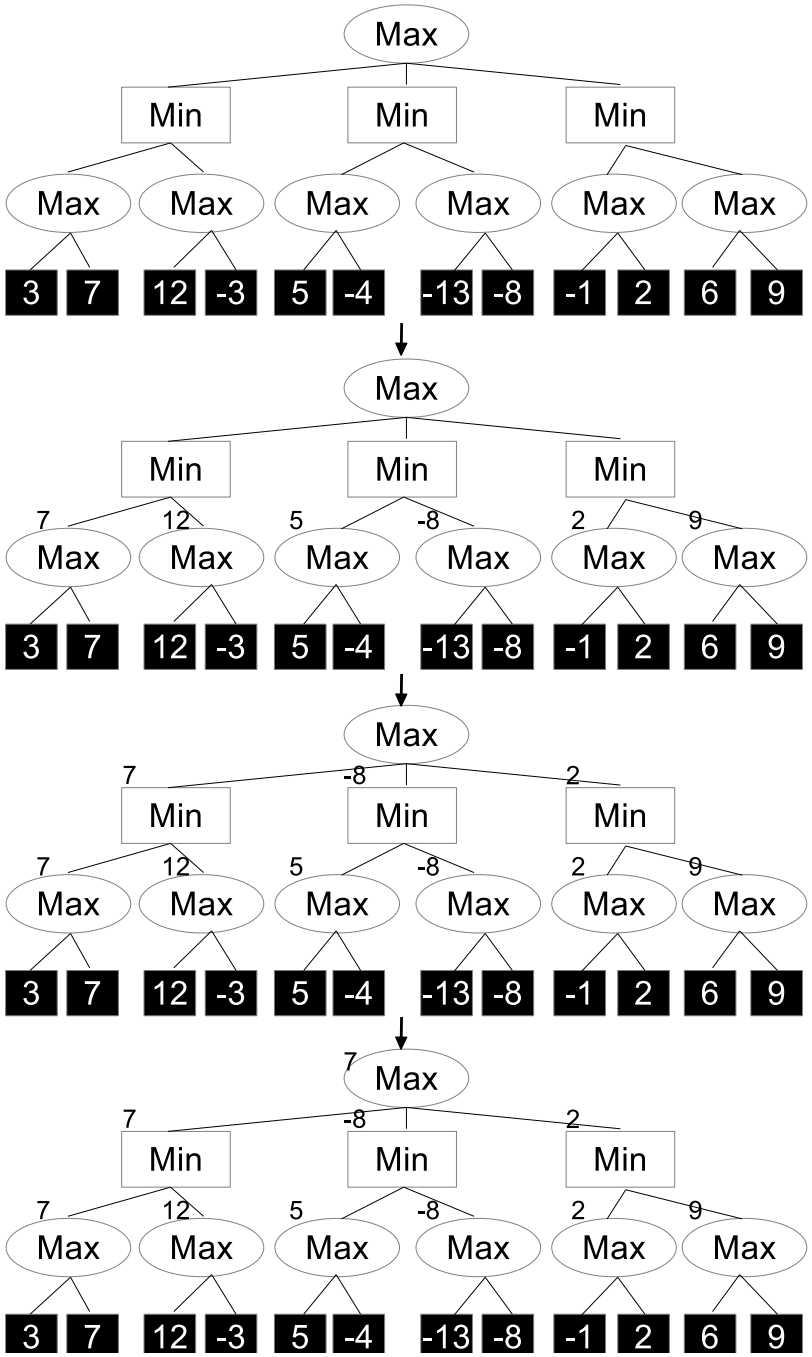

図 1.8: Min-Max でルート局面の評価値が決まる様子

1.6 α-β 探索

「実は、Min-Max 探索と同じ結果を保証しつつ、Min-Max 探索よりも大幅に少ない探索量で済む探索手法があるでゲソ。それが α-β 探索でゲソ！　どれくらい探索量が減るかというと、元の探索量の平方根くらいになるでゲソ。図 1.9 を使って説明するでゲソ。
(現在は並列化などのために厳密にそうとは言えないが) 通常は図の左にあるノードから評価関数が呼ばれるでゲソ。

- (a)start でゲソ。
- (b) 一番左の Min ノード (i) までの探索が終了した所でゲソ。ここで『よし、この一番左の手を選べば最低でも +7 で勝てる。でも、もっといい手はないのかな? 他の二つの可能性も調べてみて、+7 よりも私に有利な手ならそっちにしよう』というわけで次の探索に行くでゲソ。
- (c) というわけで二番目の手の探索を開始し、Max ノード (j) までの探索が終了した所でゲソ。
- (d)『ふむ、二番目の手を打つと、相手には評価値 +5 という手が発生する。相手はなるべく評価値の低い手を打ってくる (Min) から、二番目の手を打ったときのノード (k) の評価値が +5 より大きいということはありえない。さらに悪いことにもっと評価値が低い手、マイナスの手がある可能性もある。しかし、冷静に考えると、(b) で +7 より有利な手を探していたんだから、(k) が +5 以下であることが分かっただけで十分だな。これ以上 (k) について調べる必要はない。一部のノードは見てすらいないのだけど、三番目の手の探索に移れるな。』

という感じでゲソ！　」
「ふむ、なるほどのう。少し分かりづらいが、考えなくて良い局面があるわけじゃな。でも、今の囲碁プログラムではこの『あるふぁべーた探索』は使われていないのじゃったな？」
「その通りでゲソ！ 昔の囲碁プログラムは使っていたでゲソ。でも、すごく弱いのしか作れなかったでゲソ！　その理由を説明するでゲソ。」

1.7 α-β 探索が囲碁ではうまく機能しなかった理由

「α-β 探索が囲碁でうまく機能しなかった理由は大きく二つあるでゲソ！」「一つは囲碁では良い評価関数が作れないからでゲソ。すごくざっくり言うと、リバーシでは石のパターンを学習することで良い評価関数を作ることが出来、チェスや将棋では駒や駒組みの評価値を学習することで良い評価関数を作ることが出来たでゲソ。しかし、囲碁は、一つ一つは個性の無い石が無限とも言える組み合わさり方で様々な価値を生み出すので、良い評価関数を作ることが非常に難しいでゲソ！ 人間の強い人がどのようにして形勢を判断しているかを言葉で説明しても、最低でもアマチュア初段くらいの力のある人でないと理解できないくらい、囲碁の形勢評価は難しいでゲソ!」
「……と、私は前回説明したでゲソが、実は DeepMind の人たちは高精度の評価関数を作ることに成功してしまったでゲソ！ 後程詳しく説明するでゲソが、それが AlphaGo の一つの肝でゲソ！」
「なん…じゃと…！？ そんなことが出来てしまったのか！？ ああ、すまん、少し動揺してしまったようじゃ。話を続けてくれ。」
「爺さんの動揺も分かるでゲソが、とりあえず話を戻すでゲソ。もう一つは、囲碁では打てる手の数が多すぎて、深く読めないからでゲソ。囲碁はルール上打てる手が、9 路盤 (練習用の小さい盤) の最初の状態で 81 通り、19 路盤 (公式の盤) の最初の状態で 361 通りあり、中盤付近でも 9 路盤で 60 通りくらい、19 路盤で 250 通りくらいもあるでゲソ！ ちなみに、リバーシだと石のひっくり返せる所しか打てないので、打てる場所は 10 通りくらいのことが多いでゲソ。もちろん手を絞り込む研究もされているでゲソが、少々絞り込んだだけでは焼け石に水でゲソ！ また、絞り込むと重要な手を見落とすでゲソ！」

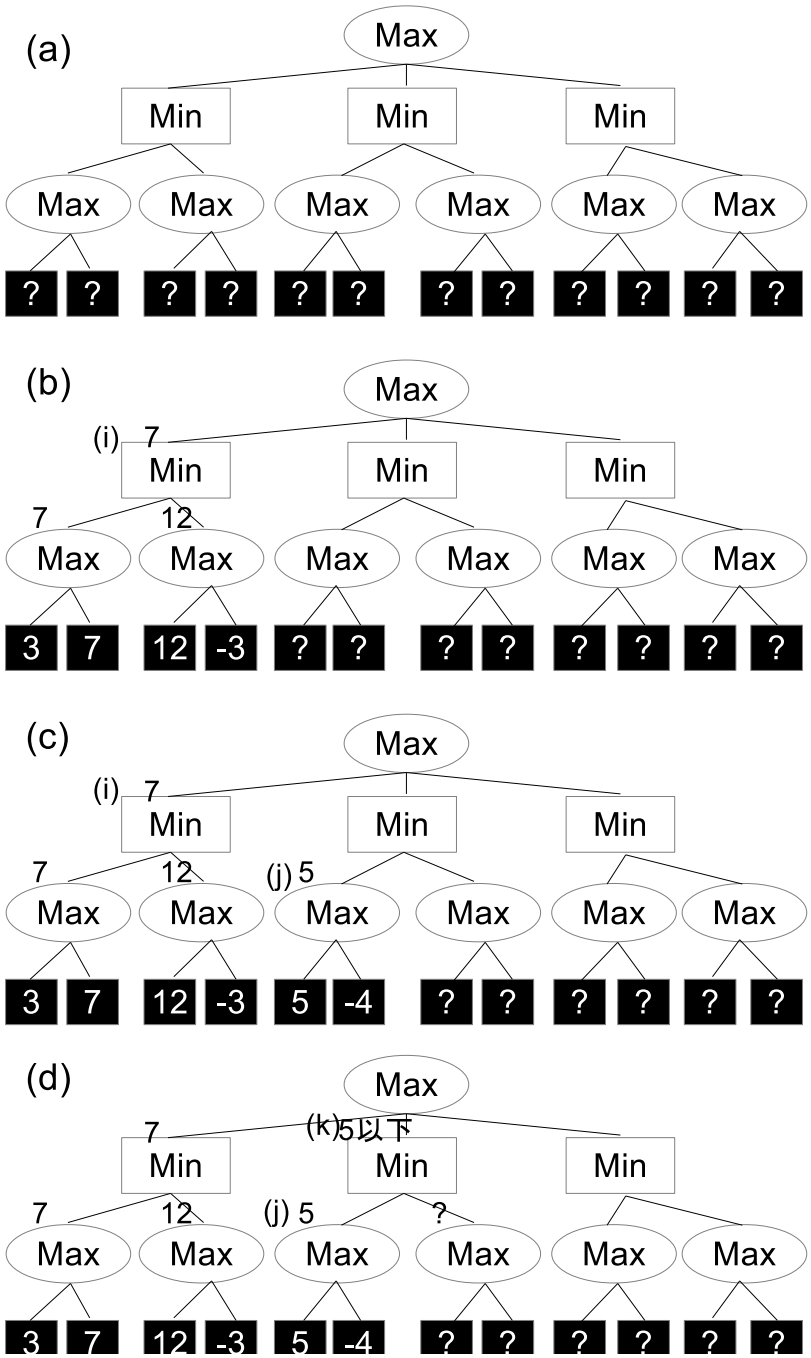

図 1.9: α-β 探索の様子

「……と、私は前回説明したでゲソが、実は DeepMind の人たちは高精度の手の絞り込みに成功したでゲソ！ それが AlphaGo のもう一つの肝でゲソ！ 厳密に言うと、手の絞り込みの方が先に出来て、それから評価関数を作ったでゲソ！」
「囲碁は、有段者になると、手を瞬間的に数か所に絞り込めると思うのじゃが、コンピュータにもそれが出来るようになったということじゃな。いやはや恐ろしい時代になったもんじゃわい！」
「全くでゲソ！ で、このまま AlphaGo の説明に行きたいのは山々でゲソが、もう少しコンピュータ囲碁の歴史の話に付き合ってもらうでゲソ！」

1.8 原始モンテカルロ法

「『モンテカルロ法』という名前くらいは聞いたことがあるでゲソか？ 乱数を使って近似値を求める方法でゲソ！ 囲碁におけるモンテカルロ法、厳密に言うと原始モンテカルロ法とは、『次の一手として有力な手がどれかを探すために、まず候補手を打ってみて、その後無作為に最後まで、自殺手以外打つ場所が無くなるまで、打つということを何回も繰り返し、結果の平均、通常は何目勝ったかではなく何回勝ったか、すなわち勝率をその候補手の点数として、点数が最大の候補手を次の一手として選ぶ』手法でゲソ。なお、『無作為に最後まで打つ』シミュレーションのことを『プレイアウト』と呼ぶでゲソ。図 1.11 を見ると分かると思うでゲソ。この図の場合は、一個の候補手あたりのプレイアウトが 100 回ということでゲソね。単純に考えると候補手が 74 箇所あるでゲソから合計で 7400 回になるでゲソ。実際には、プログラムにもよるでゲソが、合計 1 万〜100 万回くらいプレイアウトするでゲソ！ 一見すごく多そうでゲソが、αβで全部の手を深いところまで調べるのに比べたらはるかに少ないでゲソ！ 」
「そんな無茶苦茶な方法がベースでも、最終的にはアマ有段レベルの囲碁ソフトが作れたという話じゃったな。」
「モンテカルロ法は 1990 年代には提案されていたでゲソが、実は当初は弱いソフトしか作れなくて、皆もこんな方法で強いソフトが作れるわけがないと思っていたでゲソ！ でも、2006 年頃に再び注目され、多くの工夫が加えられたことで、強くなったでゲソ！ それを次の節で説明するでゲソ。」

1.9 モンテカルロ木探索

「原始モンテカルロ法だけでも弱い囲碁プログラムが出来るでゲソが、原始モンテカルロ法に木探索を組み合わせるともっと強くなるんじゃなイカ？ 図 1.12 を使って、モンテカルロ法における木探索を説明するでゲソ。まずは普通に、各候補手に対して何度かプレイアウトを行って結果を集計するでゲソ。それから、ある候補手に対してのプレイアウト数が閾値 (1〜数百のことが多い) を超えたら、そこにノードを作るでゲソ (それまではプレイアウト数を集計するための仮のノードを作っておく)。そうすると、深さ 1 のところにノードがたくさん出来るでゲソ。そうなったら、以降のプレイアウトを行う際に、そのノードの勝率とプレイアウト数に注目するでゲソ。そして、『勝率が高く、まだプレイアウト数が少ない』ノード (候補手) を選んで、そこからまたプレイアウトをするでゲソ。これを何段も繰り返すと、モンテカルロとして有望なところは深いところまでノードが作られるでゲソ！ そして、上記の $\alpha-\beta$ 探索と同様の効果が得られるようになるでゲソ！ これがモンテカルロ木探索でゲソ。」
「実はあと二つ、大きな工夫が殆どの囲碁ソフトで加えられていたでゲソ！ それが、『プレイアウト』部分を賢くするということと、『モンテカルロ木探索にバイアスをかける』とうことでゲソ！ ある局面を評価するのに、無作為打ち同士で最後まで打って結果の平均を取るより、ある程度囲碁の強いプログラム同士で最後まで打って結果の平均を取った方が良さそうだというのは、自然な考えだと思うでゲソ。もちろん、これをやると『プレイアウト』は遅くなってしまうんでゲソが、そ

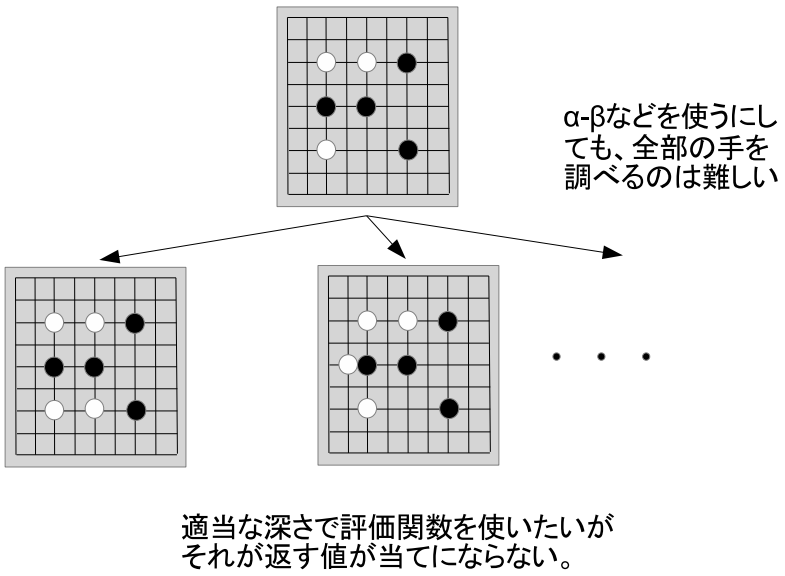

図 1.10: 囲碁で α-β は上手く機能しない

れでも、『プレイアウト』を賢くすることで、囲碁ソフトは凄く強くなったでゲソ！ また、モンテカルロ木探索を行う時に、囲碁の知識を使って『良さそうな手はより深く調べる』というバイアスをかけることでも、囲碁ソフトは凄く強くなったでゲソ！ これら二つについては前回は触れなかったでゲソが、実は AlphaGo の肝として後々出てくることに関連するので、頭の隅に置いておいて欲しいでゲソ！」

1.10 AlphaGo 登場前の囲碁ソフト

「モンテカルロ木探索と賢いプレイアウト、そして適度なバイアスを用いることで、囲碁ソフトは飛躍的に強くなったでゲソ！ 当初は 9 路盤でしか有効でないと思っていた人もいるようでゲソが、たくさんの工夫によって、19 路盤でもかなり強くなったでゲソ。AlphaGo 登場前の最強の囲碁ソフトは『Zen』と『CrazyStone』だと言われていたでゲソが、どちらも以前はネット碁の KGS というところで 5d〜6d[*2] くらいにランクされていたでゲソ。これは日本基準だとアマチュアの 7〜8 段くらいでゲソ。Zen は 4 子置いて (Zen 側が 4 段級差有利) トッププロの武宮正樹プロに勝ったことがあるでゲソ。CrazyStone も 4 子置いてトッププロの石田芳夫プロや依田紀基プロに勝ったことがあるでゲソ！ また、Zen は 3 子置いて高校チャンピオンの大表拓都さん (その後プロに転向) に勝ったこともあるでゲソ。また、AlphaGo 登場直後くらいに、この二つのソフトは 7d になったでゲソ！ 7d になった Zen は武宮陽光プロ (武宮正樹プロの息子) になんとたった 2 子置きで完勝したでゲソ！ さらにさらに、Zen は 2016 年 6 月頃に 8d になったでゲソ！」

「そうじゃったな、AlphaGo 登場前はトッププロと 4 子でいい勝負とのことじゃったな。儂くらい

[*2] KGS では対戦成績によって段級位がつく。5d というのは 5 段ということ。ただし、日本のアマチュアの 5 段よりも評価が厳しいと言われている。なお、日本と同様に級もあり、例えば 1 級なら 1k となる。

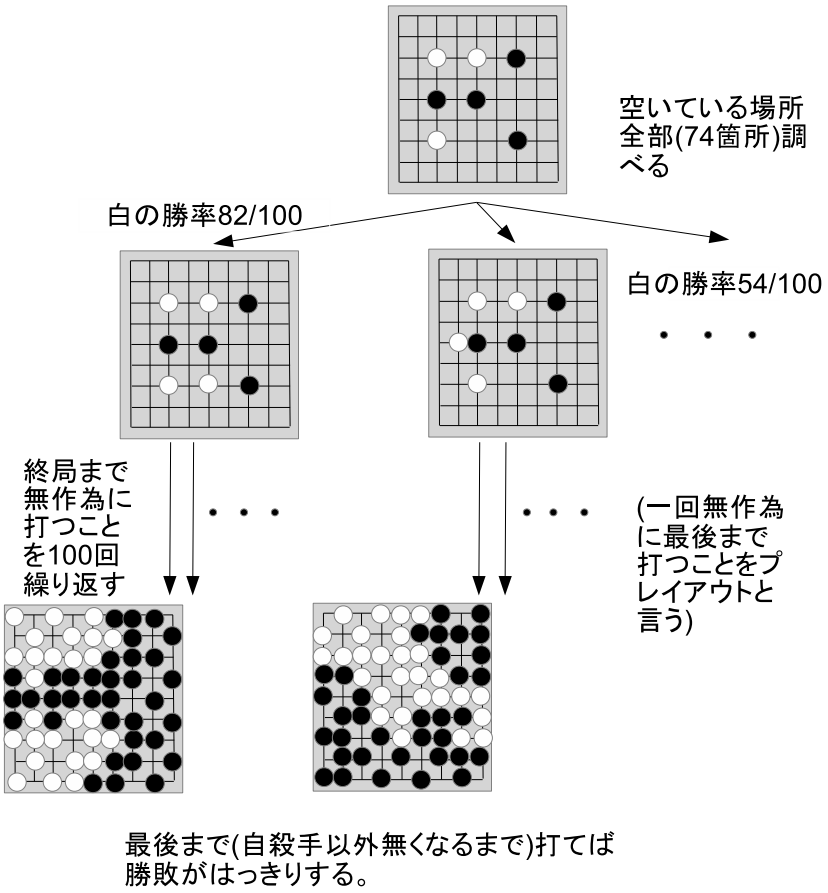

図 1.11: 囲碁における原始モンテカルロ法。

長い間囲碁をやっているとよく分かるが、たった 4 子のハンデでトッププロに勝つのは並のアマチュアではむりじゃな。」

「でも、AlphaGo が登場するしばらく前に、アマチュアトップレベルの多賀さんとハンデなしで打ってみたこともあるでゲソが、その時はボコボコにされていたでゲソ……。その後も、2015 年 3 月の UEC 杯エキシビションマッチで、そこそこ強いけどトップアマとは差があるくらいのアマチュアの人に完敗していたでゲソ…。また、2015 年 11 月に、中国で行われたコンピュータ囲碁の大会で優勝した『DolBaram』というプログラムは、当時 Zen や CrazyStone とほぼ変わらない力を

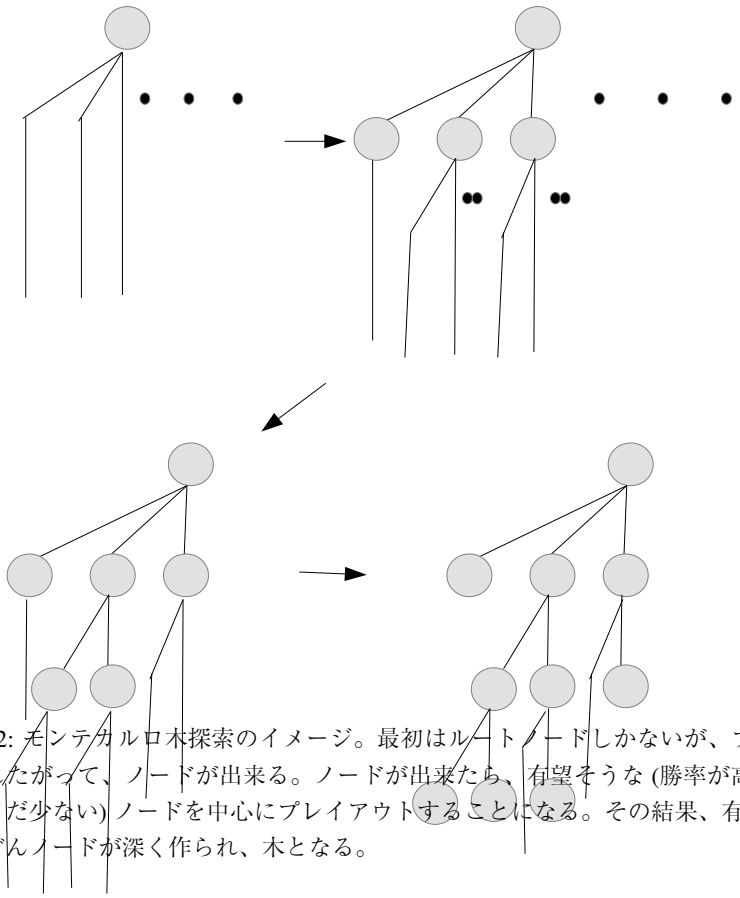

図 1.12: モンテカルロ木探索のイメージ。最初はルートノードしかないが、プレイアウトを繰り返すにしたがって、ノードが出来る。ノードが出来たら、有望そうな (勝率が高く、プレイアウト回数がまだ少ない) ノードを中心にプレイアウトすることになる。その結果、有望そうなところだけ、どんどんノードが深く作られ、木となる。

持っていたと思われるでゲソ。しかし、その時のエキシビションで、中国のプロの名人だった連笑さん 4 子置きで負け、5 子置きでも負け、6 子置きでかろうじて勝ったでゲソ。最初にも爺さんがちょっと触れていたでゲソが、『攻め合い』や『死活』等の、細い一本道だけが正解の状況に非常に弱かったでゲソ。」

「平均を取るという操作が入る以上、そういうのに弱いのは本質的に解決が困難なことで、解決するために様々な人達が研究しておるということじゃったな。」

「そうでゲソ。でも今思うと、そこにこだわってしまったことが、この文章の著者も含む多くの研究者の道を誤らせたのかも知れないでゲソ…。では、そろそろ AlphaGo の話をするでゲソ。まず、AlphaGo の肝となる『Deep Learning』の話からでゲソ！」

1.11 Deep Learning

「Deep Learning という言葉は最近広く知られるようになったでゲソ。定義は色々あるでゲソが、普通は『階層の深いニューラルネットワークを使った機械学習』のことを指すでゲソ。機械学習と言うのは、『大量の訓練データ、即ち入力と正解ラベルの集合から、そのデータに潜む傾向をプログラムが自動的に掴み、新たに入力が与えられた時に正しいと思われるラベルを返せるようになる技術』でゲソ。ニューラルネットワークと言うのは、そのような機械学習が出来る仕組みの一つで、『人間の神経網を模して、簡単な関数を大量に繋げることにより、複雑な処理が出来るようにしたもの』でゲソ。イメージを図 1.13 に示すでゲソ。そして、実はそのようなネットワークで、階層の深いものが扱えるようになったのはつい最近なんでゲソ。」

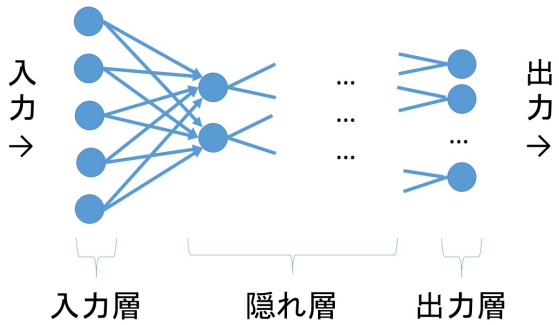

図 1.13: ニューラルネットワークのイメージ。入力ノード (関数) の数、隠れ層の数、それぞれの隠れ層のノードの数、出力ノードの数は任意。各ノードのつなぎ方にも工夫の余地がある。隠れ層の数は従来は 1 が主流だったが、最近 10 以上の物も扱えるようになってきた。

「なるほどのう。そのような技術の進歩があって、最近『Deep Learning』という言葉が聞かれるようになった訳か。」
「その通りでゲソ。そして、Deep Learning の中でも特に『Deep Convolutional Neural Netowork(DCNN)』と呼ばれるものが、画像認識の分野でブレイクスルーを起こしたでゲソ！ 図 1.14 を見てほしいでゲソ！ これは 2012 年に開かれた ILSVRC というコンペティションの優勝チームのスライドの一部でゲソ。何と、DCNN を使うと、この図にあるように『画像にどんな物体が映っているか』を高精度で認識するプログラムを作ることが出来るでゲソ！ このチームが 2012 年の時に、正答率約 85% と、2 位に 10% 以上の精度差を付けたことで、DCNN の威力が広く知られるようになったでゲソ。なお、この分野の研究が進んだ今では、正答率は 96% くらいになって、人間を超えてしまったでゲソ！」
「ほう、そんな技術革新があったのかね。郵便番号などの手書きの数字を認識するのが精一杯だった時代とはえらい違いじゃな。」
「それでは、いよいよ AlphaGo の核心に迫っていくでゲソ！」

1.12 AlphaGo

「DCNN が画像処理でブレイクスルーを起こしてから、囲碁にも DCNN を使おうという流れが起きたでゲソ！ 囲碁は絵画に例えられることもあるでゲソ。囲碁を知っている人なら、DCNN を使おうという発想は自然な物でゲソ。まず、19×19 の盤で、アマチュア高段〜プロレベルの人の次の一手を予測する研究に、DCNN が使われたでゲソ！ この研究は専門的には『Move Prediction』と呼

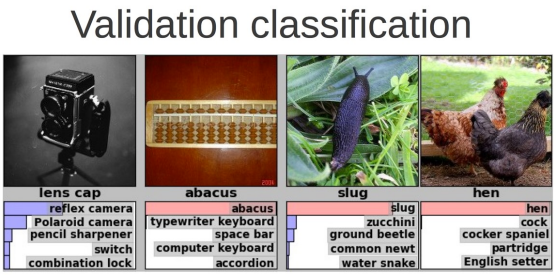

図 1.14: DCNN により、画像に何が映っているかを高精度で認識出来るようになった。(http://www.image-net.org/challenges/LSVRC/2012/supervision.pdf)

ばれるものでゲソ。」
「なるほどのう。棋力認定問題で、次の一手を選択肢から選ばせるようなものもあるからのう。」
「その通りでゲソ。ただ、棋力認定問題では通常選択肢は数個で、ある程度囲碁が打てる人にとっては分かりやすい局面を使うでゲソが、『Move Prediction』の研究では、通常『テストデータの棋譜のすべての局面において』『ルール上打てる手全て』から次の一手を選ばせるでゲソ。つまり 19×19 の盤だと、最初の一手は 361 通り、序盤で 300 通り以上、終盤でも 100 通り近くから選ばないといけないでゲソ！」
「そんなことがプログラムに出来るのかね？」
「実は、DCNN が使われる以前でも、そこそこの精度が出ていて、41% くらいの正答率だったでゲソ。でも、DCNN が使われると、一気に 55-57% くらいになったんでゲソ！」
「55% 以上じゃと！ 確かに、囲碁には定石の途中など、選択肢がかなり限られる場面もあるからのう…。いや、それでもその数値は凄いのう。」
「そして、先ほど述べた『木探索のバイアス』にこの DCNN を使うソフトがいくつか現れたでゲソ。これは AlphaGo の論文発表前から行われていたことでゲソが、それにより、ソフトは比較的容易に KGS5d-6d レベルに到達出来るようになったでゲソ！ また、元々それくらいのレベルがあった Zen や CrazyStone は、AlphaGo の論文発表に前後して DCNN を導入し始めて、KGS7d になったでゲソ！ また、AlphaGo も同様の使い方をしていて、論文中では『Policy-Network』と呼ばれているでゲソ！」
「なるほどのう。確かに、これだけ精度が上がれば、木探索のバイアスに使うだけでも強くなりそうじゃな。だが、ちょっと待ってくれ。DCNN を『プレイアウト』にも使うことは出来ないのかね？」
「爺さん鋭いでゲソね！ 実は、DCNN を『プレイアウト』に使おうと考えた研究者は多かったでゲソ。でも、実は DCNN って遅いんでゲソ…。一つの盤面から次の一手を予測するのに、GPU という装置を使っても 3 ミリ秒くらいかかるんでゲソ。囲碁のプレイアウトは 300 手以上になることもあるでゲソから、単純計算で一回プレイアウトするのに 1 秒かかるんでゲソ。囲碁ソフトは通常次の一手を決めるために万単位でのプレイアウトをするでゲソ。でも一手にかけられる時間は、通常の対局だとせいぜい数十秒でゲソ。全然間に合わないんでゲソ…。でも、実はそれを解決したのが AlphaGo なんでゲソ！」
「AlphaGo は凄まじいハードウェアで動いていると聞いたが、まさかハードウェアの物量で解決したのかね？」

「それもあるでゲソが、それだけではないでゲソ！ 順に説明するでゲソ！ まず、DCNN で作った『Policy-Network』を『強化学習』という技術を使って、『囲碁ソフトとして強く』するでゲソ！『強化学習』を詳しく説明すると本が書けてしまうので、簡単に説明するでゲソが、『強化学習』というのは『行動の結果を基に学習していく』手法でゲソ。今回の場合は、『Policy-Network』が出した次の一手の確率分布に基づく、ある程度の無作為性を持った囲碁ソフト同士を何度も対局させて、どちらが勝ったかという結果を『Policy-Network』にフィードバックさせることで強くしていくんでゲソ。」
「人間が、対局した後検討して、強くなっていくのと同様のメカニズムと思えば良さそうじゃな。」
「イメージとしてはそれでいいでゲソ。そして、その強くした DCNN 同士で改めて、3000 万局面から対局するんでゲソ！ 3000 万局面は、それぞれ別の対局のものでゲソ！ そして、また別途 DCNN を用意して、その 3000 万の局面から、対局の結果を予測出来るように学習していくんでゲソ！ これにより、『あたかも、強くした Policy-Network でプレイアウトを行ったかのような評価が、15000 倍の速さで出来る』評価関数が出来たんでゲソ！ これが AlphaGo の論文で『Value-Network』と呼ばれているものでゲソ！」
「数字を聞いただけで気が遠くなってきたわい…。つまり、AlphaGo は 3000 万回の対局、あるいは 3000 万回の棋譜並べと言うべきかもしれんが、それを行って、正確に形勢判断する能力を手に入れたわけじゃな。そういえば誰かは忘れたが、プロ棋士がそんな解説をしておった気がするのう。確かその棋士は「自分はプロになるまでに 15 万局並べた」と言っておったような…。まあ儂は、15 万局でも十分化け物じゃと思うが、3000 万か…。」
「コンピューターは、一局の棋譜から学習できる情報の量は人間より少ないかもしれないでゲソが、量をこなすのは得意でゲソ！ そして、AlphaGo 自体は、モンテカルロ木探索に、簡単なプレイアウトと、Policy-Network と、Value-Network を組み合わせて動いているでゲソ。
「なるほどのう。DCNN の威力を最大限使っている訳じゃな。」

1.13　今後の囲碁ソフト

「しかし、AlphaGo の論文は公開されたわけじゃから、誰でも強いソフトが作れるようになったのかね？」
「残念ながらそうではないでゲソ。そもそも真似するのが容易ではないでゲソ。学習に GPU を 50 個使って数週間以上の時間がかかるでゲソ。個人でやったら電気代だけで破産するでゲソ！ イ・セドルに勝った対局時も 1202CPU、176GPU という大規模構成で挑んでいるでゲソ。DeepMind を買収した Google という企業の財力があってこそ出来る AI でゲソ！ しかも、AlphaGo のイベントのしばらく後に発表されたでゲソが、実は Google は Deep Learning に適したハードウェアを自前で作って、それを AlphaGo でも使っていたそうでゲソ…。個人でそれに追いつこうとするのは、B29 に竹やりで挑むようなものでゲソ…。でも、Zen の人たちはドワンゴと協力して、追いつこうと努力しているでゲソ。また、この文章の著者も、PEZY という会社と協力して、追いつこうと努力しているでゲソ。」
「なるほどのう。それと、もう一つ確認したいのじゃが、最初に言っていた『読みの力は大したことが無い』と言うのは本当なのかね？」
「AlphaGo の公式の棋譜が少なすぎて断言は出来ないでゲソが、ベースはモンテカルロ木探索なので、モンテカルロ木探索が苦手な長手数の読みは今回の手法では解決できていないと思われるでゲソ。また、実は DCNN を使っても、少なくとも Zen や CrazyStone は、長手数の読みが苦手と言う弱点は解決出来ていないでゲソ。おそらく AlphaGo も同様でゲソ。実際、イ・セドルとの 4 局目の乱れ方や 5 局目の石塔絞りの見損じは、長手数の読みが出来ていなかったためと思われるでゲソ。その代わり、大局観は人間を凌駕していると思われるでゲソ。」

「ふむう。まあ確かに、人間の場合でも、囲碁や将棋は上達するほど『読まずに判断できる部分が増える』ものじゃからのう。まあこれは囲碁や将棋がアマ高段以上の人しか理解できない感覚かもしれんがのう。しかし、イ・セドルが負けてしまった以上、囲碁ソフトは研究対象にはもうならないのかね？」
「そんなことは無いでゲソ！ 今後囲碁ソフトの研究は『普通の PC 一台で動く強い AI』『AlphaGo でも克服できていないと思われる弱点を克服した AI』『強さ以外の利点を強調した AI』の開発に向かうと思われるでゲソ！ 別に人間を超えたというだけで、研究する価値が無くなる訳ではないでゲソ。まあ、人間超えを目指して頑張っていた開発者達の中には、やる気を無くしてしまった人も居るでゲソが…。」
「ふむぅ…。だが、ソフトがどれだけ強くなろうとも、人間同士が囲碁を打つ楽しさは、人間が存在する限り未来永劫変わらんと儂は思うよ。チェスや将棋の世界では、ソフトを使ったカンニングを防ぐための対策等が真面目に議論されておるようで、確かに囲碁もそういう対策が必要な時代になってきたのかもしれん。だが、こういうゲームは、自分で考えてこそ楽しいものじゃと思う。それに、囲碁のように難しいことで、ソフトが人間を上回るくらいになってきたということは、囲碁以外のもっと一般的なことでも、より人間の仕事を減らし、より仕事の精度を高めるようなソフトも恐らく作れるようになるじゃろう。それは人間社会の発展のために喜ばしいことだと儂は思うぞ。ブラック企業の激務で過労死する若者がいるような世の中よりは、適度に仕事をソフトに任せて、人間は適度な忙しさで暮らせる世の中の方が良いと儂は思うな。」
「その通りだと私も思うでゲソ！ じゃあ爺さん、また一局教えてくれなイカ？」
「勿論構わんよ。お主にもそろそろ追いつかれるかもしれんのう！」

1.14 参考文献

- コンピュータ囲碁-モンテカルロ法の理論と実践-(松原仁　編集、美添一樹、山下宏共著 共立出版)
- ゲーム計算メカニズム (小谷善行 編著、岸本章宏、柴原一友、鈴木豪 共著　コロナ社)
- Combining Online and Offline Knowledge in UCT(Sylvain Gelly, David Silver, Proceedings of the 24th International Conference on Machine Learning 2007)
- Computing Elo Ratings of Move Patterns in the Game of Go(Rémi Coulom, ICGA Computer Games Workshop, Amsterdam, The Netherlands, June 2007)
- http://www.image-net.org/challenges/LSVRC/2012/supervision.pdf
- Training Deep Convolutional Neural Networks to Play Go(Clark Christopher, Storkey Amos, Proceedings of the 32nd International Conference on Machine Learning 2015)
- Move Evaluation in Go Using Deep Convolutional Neural Networks(Chris J. Maddison, Aja Huang, Ilya Sutskever, David Silver, Proceedings of the 3rd International Conference on Learning Representations 2015)
- Better Computer Go Player with Neural Network and Long-Term Prediction(Yuandong Tian, Yan Zhu, Proceedings of the 4th International Conference on Learning Representations 2016)
- Mastering the game of Go with deep neural networks and tree search(David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, Demis Hassabis, Nature 529 2016)
- http://nitro15.ldblog.jp/archives/46716737.html

第2章

変態する昆虫の観察

— @master_q

2.1 今日の宿題

「ぱっと見ただけだと昆虫は不規則な動きをしているだけのようでも、よーく観察すると規則が見つかることがあるじゃなイカ。」
ウチのクラスの担任はいつも授業にノリノリだ。
「このように一見動的に見えるモノでも、変わらない規則があることがあり、これを**静的な意味**と呼ぶことがあるでゲソ。」
変わらないこと... そんなものがあるのだろうか。全ての生物は死んでしまうし、統一理論は一向に定まる気配はない。これまで人類が組み上げてきた数学だって、根本的な矛盾が発見されたら、その全体を修正するしかないんじゃないか？
「そういうわけで、今日の宿題は昆虫の観察でゲソ! 来週は皆に昆虫に見る静的な意味を発表してもらおうじゃなイカ。」
今は 21 世紀。火星への移住が計画されるような現代において、昆虫を観察することにはたして意義があるのだろうか... ヤレヤレだ。

2.2 観察

昆虫といえば以前とても小さな OS、**ChibiOS/RT** [*1] を見つけた。この OS はバイナリサイズが小さく、コンテキストスイッチにかかる時間が短かいことが特徴 [*2] で、8-bit MCU である AVR や 32-bit MCU である ARM Cortex-M シリーズ上で動く。また良くドキュメントも整備されていて好感がもてる。僕はこの OS を日々の工作にも使っている。

ところが、この ChibiOS/RT 上で動くアプリケーションを作るのは簡単ではない。その理由はいくつもあるが、その 1 つはシステム API の呼び出しマナーがわかりにくいことだ。ChibiOS/RT のような OS をリアルタイム OS と呼ぶが、このリアルタイム OS に習熟していれば、ChibiOS/RT に入門するのはたやすいかもしれない。しかし、デスクトップ OS 上でのアプリケーションだけの知識だと以下のような概念に馴染みがないことがあるのだ。

- POSIX スレッドではない独自のスレッド機構
- システムグローバルのロック
- システム状態
- 再入可能
- 割り込みの許可/無効

[*1] http://www.chibios.org/
[*2] http://wiki.chibios.org/dokuwiki/doku.php?id=chibios:metrics

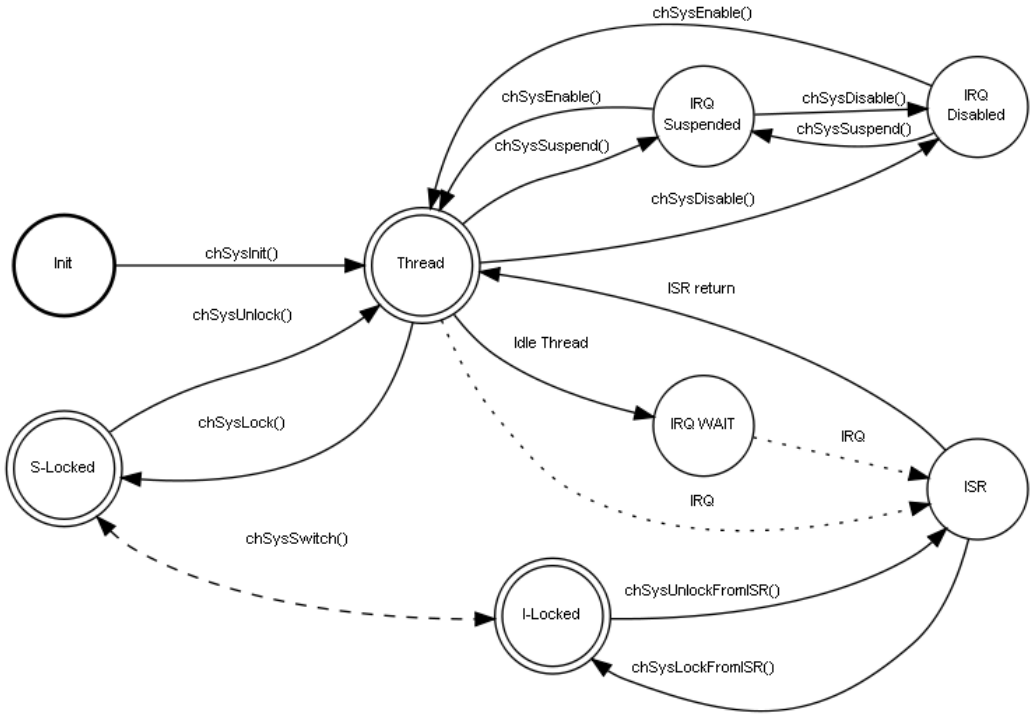

図 2.1: ChibiOS/RT のシステム状態

ChibiOS/RT の場合、これらの概念は具体的にどのようになるだろうか。まずこの OS には図 2.1 [*3] のような**システム状態**がある。ChibiOS/RT では `main()` 関数は Init 状態で開始される。`main` 関数の中で自発的に OS の初期化関数 `chSysInit()` を呼び出すことで、この OS 全体は Thread 状態に遷移する。この状態においてシステムグローバルのロックを獲得する関数 `chSysLock()` を呼び出すと、システム状態は S-Locked に遷移する。ロック排他すべき処理が終わったら `chSysUnlock()` 関数でロックを解放するとシステム状態は Thread に戻る。

もし Thread 状態の最中にハードウェア割り込み (IRQ) が入るとどうなるだろうか？ この場合、実行中のプログラムは中断し、強制的に割り込みハンドラが実行される。そしてシステム状態は ISR 状態に遷移する。つまり割り込みハンドラは常に ISR 状態から開始されるのだ。割り込みハンドラの処理が終わって関数から返るとハードウェア割り込みは終了し、Thread 状態に戻る。そして同時に中断されていたプログラムが再開されることになる。

比較のために、POSIX 上でのプログラミングを思い出してみよう。システム状態と呼べるようなものはメインコンテキストとシグナルコンテキストの二種類しかなかった。シグナルコンテキストから呼び出せる関数、つまり再入可能な関数は POSIX で定義されているが、実装によりけりなのでシグナルコンテキストであまり凝った処理をすることは稀だろう。システムグローバルのロックもない。考慮しなければならないことは少なかったはずだ。ChibiOS/RT のようなシステム状態は POSIX を実現している OS が吸収してくれているのだ。

ところが、ChibiOS/RT 上のアプリケーションは上記のシステム状態を把握しながらプログラミングする必要があるのだ。しかも各システム状態から呼び出せるシステム API は限定されていて、間

[*3] `http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel#system_states` から引用

違った作法でシステム API を呼び出すと未定義な挙動を引き起こす。

納得しやすい例からはじめよう。先の Thread 状態においてシステムグローバルなロックを獲得する関数 chSysLock() は当然だが Thread 状態でのみ使用することができる。もしこの関数を ISR 状態で使った場合の挙動は未定義だ。デバッグ機能を有効にすることで実行時に検出できるが、OS 全体のパフォーマンスが落ちてしまう。それでもこのような作法は先のシステム状態のグラフを常に参照しながらプログラミングすれば間違えることは少ないかもしれない。

しかしその他の全てのシステム API [*4] もまた使用して良いシステム状態が定められている。これらのシステム API は以下のような種類に分類 [*5] できる。

Normal Functions Thread 状態からのみ呼び出し可能
S-Class Functions S-Locked 状態からのみ呼び出し可能
I-Class Functions I-Locked もしくは S-Locked 状態からのみ呼び出し可能
X-Class Functions Thread、S-Locked、I-Locked 状態からのみ呼び出し可能
Special Functions 個々の関数によって呼び出し可能条件がある
Object Initializers どの状態からでも呼び出し可能

例えばシステム API には chEvtBroadcastI() という関数 がある。ドキュメント [*6] によるとこの関数は I-Class に分類される。つまりこの関数は I-Locked もしくは S-Locked 状態でのみ呼び出し可能で、例えば Thread 状態から呼び出した場合の挙動は未定義だ。しかもシステム API には星の数ほどの関数があり、それぞれ種別が異なる。

> This is an I-Class API, this function can be invoked from within a system lock zone by both threads and interrupt handlers.

システム状態とシステム API の呼び出し作法の徹底は明らかに人為的なミスを引き起こしやすい。にもかかわらず、このシステム状態における特性は明らかに規則的だ。このような作法を人間が検査するのではなく、機械的に検査することはできないのだろうか？ この特性は担任が言っていた静的な意味にあたるのだろうか...

2.3 静的な意味とは何か

ATS [*7] というプログラミング言語がある。この言語は ML 風の文法を持ち、依存型と線形型をそなえている。ATS における型は静的な値に依存することができ、動的なコードの静的な意味を表現することができる。ATS には数多くの機能があるが、今日は線形型について理解しよう。ATS における線形型は**観**と呼ばれ、大きく以下の 2 種類に大別される。

- メモリ上の実体と紐付いた**駐観**
- メモリ上の実体を持たない**抽象観**

まず理解しやすそうな駐観からはじめよう。言ってしまえば、駐観とは「ポインタ先のメモリへのアクセス権限を付与するチケット」のことだ。ポインタとこのチケットは一対一に対応し、ATS

[*4] http://chibios.sourceforge.net/docs3/rt/
[*5] http://www.chibios.org/dokuwiki/doku.php?id=chibios:book:kernel#api_classes
[*6] http://chibios.sourceforge.net/docs3/rt/group__events.html#ga3ad84567336d57a5de5d804b01fa8cdb
[*7] http://www.ats-lang.org/

プログラムの文脈がこのチケットを所持している間はポインタの先のメモリにアクセス (**デリファレンス**) できる。もしこのチケットを持っていない文脈からはポインタを所持していてもデリファレンスできない。もしチケットを所持していないにもかかわらず、無理矢理デリファレンスしようとした場合、コンパイルエラーになる。

具体例を見てみよう。まず main.dats というファイルを作り、次のような関数を定義してみよう。[*8]

```
fun show_int {l:addr} (pfat: !int@l | p: ptr l): void =
  println! ("Int value = ", !p)
```

show_int 関数はポインタ p を受け取って、それを !p でデリファレンスしてその内容をコンソールに println! 関数で印字する。このポインタ p の型は ptr l 型だ。この型は静的なアドレス値 l に依存している。高階な解釈をすると、ATS において依存した型というのは静的な値によって特徴づけられた型だと解釈できる。ここでは ptr l 型はそのアドレスが l であるような型だと解釈できる。

では静的なアドレス l は何なのかというと、{l:addr} の全称量化によって静的に導入された値だ。ここでは具体的な番地はさだまっていないが、どのような静的な値 l においても関数 show_int は呼び出せることを意味している。

この静的なアドレス l に依存している型はもう一つあり、それは命題 pfat の型である int@l だ。この型を ATS では駐観と呼び、先の説明におけるチケットに相当する。つまり pfat の命題が有効な範囲でしかポインタ p をデリファレンスをすることはできない。例えば、以下のように関数を修正するとコンパイルエラーになる。なぜならポインタ p をデリファレンスする権利を主張する駐観が見つからないためだ。

```
fun show_int {l:addr} (p: ptr l): void =
  println! ("Int value = ", !p)
```

この仕組みの目的はポインタのデリファレンスできる範囲をコンパイル時に特定することだ。このようなデリファレンスの権利は古典論理の命題で表わすこともできるが、古典論理では、そのような権利を一度導入してしまうと削除することができない。しかしポインタのデリファレンスする権利は範囲を指定したい。例えば malloc でメモリを確保した直後では当該メモリを指すポインタはデリファレンスしたいが、free でメモリを解放した後は当該ポインタのデリファレンスは禁止したい。このような目的には線形論理の命題である線形型が利用できる。線形論理の命題は生成と消費ができるためだ。先の例だと線形型を使う free の後では当該ポインタに対応する駐観を削除することができる。

このような駐観と実際のポインタとは静的な値への依存によって結びついている。つまりポインタは静的なアドレスに依存し、線形論理の命題である駐観も同じ静的なアドレスに依存している。静的なアドレスでは何の特性も指定されいない。にもかかわらずポインタのデリファレンスできる範囲を指定できるのは、ポインタと駐観が静的なアドレスを橋にして一対一に結び付いているためだ。これは動的なポインタの特性を線形論理の命題によって仕様記述しているとも考えることができる。

次に抽象観について理解しよう。先の駐観は静的なアドレスに依存していた。しかし実際のメモリとは関連がない観も作ることができ、それは抽象観と呼ばれる。抽象観も線形型なので静的な値

[*8] main.dats のコード全体と Makefile は https://github.com/jats-ug/practice-ats/tree/master/at-view から入手できます。

に依存することができ、生成と消費をすることができる。

この抽象観の利用例として、静的な状態マシンがある。例えば 3 つの状態 A, B, C があったとき、A から B に遷移し、さらに B から C に遷移して、状態 C からしか呼び出せない関数を呼び出すプログラムを以下のように書くことができる。[*9]

```
#include "share/atspre_define.hats"
#include "share/atspre_staload.hats"

#define A 0
#define B 1
#define C 2
absvtype state(s:int)

fun a_to_b (pf: !state(A) >> state(B) | ): void = {
  val () = println! "Change state to B from A."
  prval () = __change (pf) where {
    extern praxi __change (pf: !state(A) >> state(B)): void
  }
}

fun b_to_c (pf: !state(B) >> state(C) | ): void = {
  val () = println! "Change state to C from B."
  prval () = __change (pf) where {
    extern praxi __change (pf: !state(B) >> state(C)): void
  }
}

fun do_in_c (pf: !state(C) | ): void =
  println! "Call from C."

fun run (pf: !state(A) >> state(C) | ): void = {
  val () = a_to_b (pf | )
  val () = b_to_c (pf | )
  val () = do_in_c (pf | )
}
```

上記のコードはコンパイル可能だ。しかし、以下のようにエントリ関数 `run` を書き換えて、状態 B から `do_in_c` 関数を呼び出すとコンパイルエラーになる。

```
fun run (pf: !state(A) >> state(C) | ): void = {
  val () = a_to_b (pf | )
  val () = do_in_c (pf | )
}
```

[*9] コード全体と Makefile は `https://github.com/jats-ug/practice-ats/tree/master/state` から入手できます。

どうしてこのように状態のコンパイル時検査が可能なのだろうか？ 先のコードを詳しく見ていこう。まず absvtype state(s:int) で、静的な整数に依存した抽象観を定義している。駐観の場合はポインタと関連付いていたが、この抽象観は依存した静的な値に依存するどのような動的な実体も持たないので、そのような関連はない。しかし、抽象観が依存した静的な整数には意味があり、その値が 0 ならば状態 A を、1 ならば状態 B を、2 ならば状態 C をそれぞれ表わす。

run 関数は状態 A からはじまり状態 C で終わる。これを表現するために run 関数は抽象観 state(A) を取り、関数の終了時にはその観を state(C) に変化させなければならないことが型シグニチャに明記されている。run 関数は外部から受け取った抽象観を a_to_b 関数、b_to_c 関数、do_in_c 関数に順に渡す。

a_to_b 関数は状態 A ではじまり状態 B で終わることが関数シグニチャに明記されている。この関数は他のどのような関数にも頼らずに実際に抽象観を消費/生成して、静的な状態を変化させる責務があるが、そのような組み込みの関数はない。そこで、a_to_b 関数からしか呼び出せない __change 関数を作り、実際に抽象観を変化させる。この __change 関数は危険な関数で、もし他の関数から呼び出し可能になってしまうと意図しない箇所で状態が変化してしまう。これを防ぐために __change 関数はローカルな関数として定義する必要がある。この作法は b_to_c 関数でも同様だ。

こうして状態 C に到達できたら、do_in_c 関数を呼び出すことができる。なぜなら state(C) の抽象観を手にしているはずだからだ。この do_in_c 関数の型シグニチャによると、状態 C を表わす抽象観を引数として渡す必要があり、この関数はその抽象観を消費せず、変化させることもない。

これらの観を使うことで、ATS は変化する動的なコードの中の静的な意味を捕まえて、コンパイル時に検査することができるのだ。

2.4 捕まえよう

ChibiOS/RT における問題に戻ろう。システム状態によって呼び出し可能な関数が異なり、その徹底には人為的ミスが混入しやすい、ということが問題なのだった。システム状態は明らかに状態マシンそのもので、ATS の抽象観によって表現できそうだ。そこで、ChibiOS/RT 上で動作している具体的なアプリケーションを例にして、そのアプリケーションの一部を ATS で記述し、システム状態を抽象観によって表現することで、そのような人為的ミスをコンパイル時検査してみよう。[*10]

参考にするアプリケーションは STM32 F7 Discovery ボードのデモコードの一部である以下の C 言語コードだ。

```
static void tmrfunc(void *p) {
  BaseBlockDevice *bbdp = p;

  chSysLockFromISR();
  if (cnt > 0) {
    if (blkIsInserted(bbdp)) {
      if (--cnt == 0) {
        chEvtBroadcastI(&inserted_event);
      }
    }
    else
```

[*10] この章で作るコード全体は https://github.com/fpiot/chibios-ats-2 から入手できます。

```
      cnt = POLLING_INTERVAL;
  }
  else {
    if (!blkIsInserted(bbdp)) {
      cnt = POLLING_INTERVAL;
      chEvtBroadcastI(&removed_event);
    }
  }
  chVTSetI(&tmr, MS2ST(POLLING_DELAY), tmrfunc, bbdp);
  chSysUnlockFromISR();
}

static void tmr_init(void *p) {

  chEvtObjectInit(&inserted_event);
  chEvtObjectInit(&removed_event);
  chSysLock();
  cnt = POLLING_INTERVAL;
  chVTSetI(&tmr, MS2ST(POLLING_DELAY), tmrfunc, p);
  chSysUnlock();
}
```

tmr_init は Thread 状態から呼び出されて、イベントの初期化をした後、タイマー割り込みのハンドラとして tmrfunc を登録する。タイマー割り込みから呼び出されるため tmrfunc 関数は ISR 状態から呼び出され、ボードに挿入された SD カードの状態が変わっていないか調べた後、chEvtBroadcastI 関数を呼び出してそのイベントをメインコンテキストに伝える。

先の静的な状態マシンの例と同様に、上記の C 言語コードを ATS の抽象観を使って書き直すために、まずはシステム状態を表わす抽象観を導入しよう。

```
#define chss_init       0
#define chss_thread     1
#define chss_irqsusp    2
#define chss_irqdisable 3
#define chss_irqwait    4
#define chss_isr        5
#define chss_slock      6
#define chss_ilock      7
absvtype chss(s:int)
vtypedef chss_any = [s:int | chss_init <= s; s <= chss_ilock] chss(s)
vtypedef chss_iclass = [s:int | s == chss_slock || s == chss_ilock] chss(s)
```

上記のコードは静的な整数に依存した抽象観 chss(s:int) を導入している。この抽象観が依存した整数は個々のシステム状態を表わす。このままでも抽象観を使ってシステム状態を表現できているが、複数の状態を許すことが難しい。そこで、状態の範囲を指定した観型を定義する。観型 chss_any は存在量化で導入された静的な整数に依存することで、どのようなシステム状態でも許

されることを表わしている。同様に観型 chss_iclass も存在量化で導入された静的な整数に依存しており、I-Class すなわち I-Locked もしくは S-Locked のシステム状態を表わす。

抽象観としてシステム状態を定義できたので、システム API の呼び出しは以下のように型シグニチャで表現できる。

```
extern fun chSysLock (!chss(chss_thread) >> chss(chss_slock) | ): void
  = "mac#"
extern fun chSysUnlock (!chss(chss_slock) >> chss(chss_thread) | ): void
  = "mac#"
extern fun chSysLockFromISR (!chss(chss_isr) >> chss(chss_ilock) | ): void
  = "mac#"
extern fun chSysUnlockFromISR (!chss(chss_ilock) >> chss(chss_isr) | ): void
  = "mac#"
extern fun chEvtBroadcastI (!chss_iclass | cPtr0(event_source_t)): void
  = "mac#"
extern fun chEvtObjectInit (!chss_any | cPtr0(event_source_t)): void
  = "mac#"
extern fun chVTSetI (!chss_iclass | cPtr0(virtual_timer_t), systime_t,
  vtfunc_t, cPtr0(BaseBlockDevice)): void = "mac#"
```

上記の型シグニチャによると、chSysLock 関数はその引数に抽象観として表わされたシステム状態を取る。この関数は Thread 状態においてのみ呼び出すことができ、返った後にはシステム状態が S-Locked になる。

また複数の状態において呼び出されるシステム API も表現できる。chEvtBroadcastI 関数はシステム状態が I-Class であれば呼び出すことができる。つまり I-Locked 状態でも S-Locked 状態でもこの関数は呼び出すことができる。この関数はシステム状態を変更しない。

すると先の C 言語コードは以下のように ATS 言語で記述できる。

```
extern fun tmrfunc (!chss(chss_isr) | ptr): void
implement tmrfunc (pss | p) = {
  val bbdp = $UN.cast{cPtr0(BaseBlockDevice)}(p)

  val () = chSysLockFromISR (pss | )
  val cnt = $extval(int, "cnt")
  prval pss2 = pss
  val () = if cnt > 0 then
             if blkIsInserted (bbdp) then {
               extvar "cnt" = cnt - 1
               val cnt = $extval(int, "cnt")
               val () = if cnt = 0
                          then chEvtBroadcastI (pss2 | inserted_event_p)
             } else {
               extvar "cnt" = POLLING_INTERVAL
             }
           else if ~blkIsInserted(bbdp) then {
```

```
            extvar "cnt" = POLLING_INTERVAL
            val () = chEvtBroadcastI (pss2 | removed_event_p)
          }
  prval () = pss := pss2
  val () = chVTSetI (pss | tmr_p, MS2ST (POLLING_DELAY), tmrfunc, bbdp)
  val () = chSysUnlockFromISR (pss | )
}

extern fun tmr_init (!chss(chss_thread) | ptr): void = "mac#"
implement tmr_init (pss | p) = {
  val bbdp = $UN.cast{cPtr0(BaseBlockDevice)}(p)

  val () = chEvtObjectInit (pss | inserted_event_p)
  val () = chEvtObjectInit (pss | removed_event_p)
  val () = chSysLock (pss | )
  extvar "cnt" = POLLING_INTERVAL
  val () = chVTSetI (pss | tmr_p, MS2ST (POLLING_DELAY), tmrfunc, bbdp)
  val () = chSysUnlock (pss | )
}
```

全体的に上記の見た目は元になった C 言語コードと変わらない。主な違いは全ての関数が pss という抽象観の命題を引数として取るようになっていることだ。この命題 pss がシステム状態を表現しており、間違ったシステム API 呼び出しをコンパイル時に防止してくれる。例えば、以下のようにコードを修正してロックせずに chVTSetI 呼び出しをしようとすると、コンパイルエラーになる。すばらしい。

```
extern fun tmr_init (!chss(chss_thread) | ptr): void = "mac#"
implement tmr_init (pss | p) = {
  val bbdp = $UN.cast{cPtr0(BaseBlockDevice)}(p)

  extvar "cnt" = POLLING_INTERVAL
  val () = chVTSetI (pss | tmr_p, MS2ST (POLLING_DELAY), tmrfunc, bbdp)
}
```

ちなみに今回の手法は ChibiOS/RT よりもっと大きな OS である NetBSD kernel [*11] における lwp 構造体の利用に良く似ている。NetBSD kernel 中の関数の多くはその第一引数に lwp 構造体を取る。lwp 構造体は light-weight process (LWP) と呼ばれる kenrel 内プロセスの状態を管理している。NetBSD kernel のシステム API の中には以下のように LWP の状態を実行時に検査するものがある。

```
void
lwp_exit(struct lwp *l)
{
        struct proc *p = l->l_proc;
        struct lwp *l2;
```

[*11] http://netbsd.org/

```
        bool current;

        current = (l == curlwp);

        KASSERT(current || (l->l_stat == LSIDL && l->l_target_cpu == NULL));
```

もちろん上記の C 言語コードの KASSERT 関数は実行時の検査だ。一方今回 ChibiOS/RT に導入した ATS の抽象観を使ったシステム状態の表現はコンパイル時に検査できる。NetBSD kernel 中の LWP もいつの日かコンパイル時に静的に検査できる日が来るかもしれない。

世界は繋がっている。ひょっとしたら小さな昆虫の観察も将来の宇宙開発と繋がる可能性もゼロだとは言えないのかもしれない。

2.5 変わらないと信じること

「この宿題はよく書けてるじゃないカ!」
めずらしく宿題をまじめにやってきた。担任に褒められるのはなぜだか少し恥しい。

思い直してみれば、変わないことは現実に存在するかもしれない。生物はいつか死んでしまう。でも飼っていた猫が僕にとてもなついていたことは今でも覚えている。今日の時点での統一理論を信じて物理現象を説明することはできる。明日違う統一理論を信じることも自由だ。数学の命題の矛盾は誤った公理から導かれることも多い。しかし公理に矛盾をかかえていても命題を表現することはでき、それは物事を説明するには十分機能する。完全でない数学でも有用なのだ。そして僕等の心の中に想起 (生成) されたそれらの記憶は、いつでも忘却 (消費) することができる。忘却されて存在が消えてしまっても、想起から忘却までの間に記憶が存在していたという事実は変わらない。

変わらないことは存在する。ヒトやモノを観測して、その特性を信じる人間の心の中に。

2.6 ライセンス

本記事で引用した ChibiOS/RT のコードは以下のライセンスで配布されています。

```
ChibiOS/RT - Copyright (C) 2006-2013 Giovanni Di Sirio

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

また、本記事で引用した NetBSD のコードは以下のライセンスで配布されています。

```
Copyright (c) 2001, 2006, 2007, 2008, 2009 The NetBSD Foundation, Inc.
All rights reserved.
```

This code is derived from software contributed to The NetBSD Foundation
by Nathan J. Williams, and Andrew Doran.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE NETBSD FOUNDATION, INC. AND CONTRIBUTORS
``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FOUNDATION OR CONTRIBUTORS
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.

2.7 参考文献

- ChibiOS/RT [*12]
- ATS programing on ChibiOS/RT [*13]
- The ATS Programming Language [*14]
- JATS-UG - Japan ATS User Group [*15]
- ATS プログラミング入門[*16]
- ATS プログラミングチュートリアル[*17]

[*12] http://www.chibios.org/
[*13] https://github.com/fpiot/chibios-ats-2
[*14] http://www.ats-lang.org/
[*15] http://jats-ug.metasepi.org/
[*16] http://jats-ug.metasepi.org/doc/ATS2/INT2PROGINATS/index.html
[*17] http://jats-ug.metasepi.org/doc/ATS2/ATS2TUTORIAL/index.html

第3章

IST(Internal Set Theory)入門(前編)

— @dif_engine

目標と概要

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

3.1 プロローグ

夏が苦手だ．夏の草木や空気——そしてなによりあの太陽——それらが，まるで横溢するその生命力でわたしを圧殺しようとしているかのように感じてしまうのだ．まあこんなのは，うっすらとした霧につつまれて夏でもひんやりとしたこの紅魔館の，さらに地下の図書館にこもっていたいだけのわたしの言い訳なのかもしれないが．

いつものように遅く起きたわたしは，居室代わりにしている紅魔館の図書館の扉を開けた．漂ってくる紅茶の香り——この部屋に先客だなんて珍しい．
すました顔で茶を喫していたのはアリスだった．アリス・マーガトロイド．「人形遣い」の能力を持つ魔法使いだ．彼女の傍らには美しい人形があった．図書館のテーブルのその上にミニチュアテーブルが置かれており，その上に可愛らしいティーセットが並んでいた．このミニチュアティーセットからも湯気がうっすらと立ち上っている．ミニチュアテーブルの傍らの椅子に座る，ちんまりとした陶製の顔をもつその人形は服から顔にいたるまでアリスに似せて造られていた．
それにしても，とわたしは思う——この人形は，なんとアリスに似ていることだろう——と．それとも話は全く逆で，アリスがあの人形に似ているのかしら——などと埒もないことを考えてしまう．アリスがカップと皿をテーブルに戻すと同時に，人形が動き出す．人形は，その創り主を真似たような優雅な所作でティーカップと皿をとりあげ，うっとりとした表情で紅茶の芳香を味わっていた．
じっとその様子を眺めていると人形がついと顎を上げ，眼が合ってしまった．光彩をその裡にけぶらせた宝石のような瞳は夢を見るようにはかなげで，わたしはたちまち魅入られてしまう．
「——挨拶もなしに人の顔をじろじろ見るなんて，礼儀を知らないのかしら」
声の主はアリスだった．
「あらごめんなさい——」
そう言いつつわたしは戸惑う．いまわたしは彼女の人形の顔を見ていたはずなのでは？ いやいや，それも気になるところだが紅魔館の食客とはいえわたしはこの部屋の主なのだ，連絡もなく訪れて

我が物顔でお茶を飲んでいる彼女のほうがよほど礼儀に反しているということにならないだろうか？
そんな疑問も，彼女の白皙の美貌に軽く跳ね返されてしまいそうで思わずわたしは黙ってしまったのだが——

まさにこの瞬間，わたしはある異変に気づいた——ミニチュアテーブルの周りの人形がいつの間にか二体に増えていたのだ．新しい人形の顔はこちらからは陰になって見えないが，その衣装には見覚えがあった——わたしの服だ．ならば，この新しい人形の顔がどんな様子かは予想がつく——きっとわたしそっくりに違いない．
アリスの後ろに回りこんで新しい人形の顔を覗き込みたいという気持ちを抑えながらわたしは考えた；一体いつアリスは新しい人形——おそらくわたしそっくりの顔をした——を取り出したのだろう？
いっそアリスに直接聞いて見ればよかったのかもしれない．だが——なんと聞けばよかったのだろうか．「あら，素敵なお人形さんね，さっきまで一体しかなかったお人形が二体に増えてるわね．それともわたしの見間違いかしら？ 片方はアリス，あなたそっくりだけどもう一方は，後ろ姿からするとわたしに似てなくもないわね」とか？
だがあの場面ではそんな質問を思いつきもしなかったし，アリスが出し抜けに
「魔理沙が来るの——いまから，ここに」
と宣言したことでそんな疑問も消し飛んでしまったのだった．

3.2 解析学からの無限小概念の追放

魔理沙は今回も空堀づたいに図書館に入ってきた．もうお昼も近いというのに「おはよう」などと，わたしたちのどちらにともなく挨拶してからこちらに向き直ると早速要件を切り出した——．
「ねえパチュリー，IST っての教えてよ」
「IST って言っても色々あるわね．ぱっと思いつくだけでも Intuitionistic Set Theory（直観主義的集合論）もあるし，Internal Set Theory （内的集合論）もあるし，Irish Standard Time （アイルランド標準時）なんかもあるわ」
「あ，それそれ」
「《それ》じゃわからないでしょ」
「Internal Set Theory のほう」
「ねえ魔理沙，この IST が何をするものなのかは知ってるの？」
「知ってるよ，超準解析でしょ？」
「ええその通りよ．では超準解析は何をするものかしら？」
「超準解析は微積分の議論をすごく簡単にしてくれるんでしょ」
乱暴な要約だが，まったく間違っているとも言えない．それにしてもどうして魔理沙が超準解析に興味を持っているのだろうか．
「なるほどね，では魔理沙の言うとおり微積分の議論が超準解析によって簡単になるとしたら，そもそもなんで微積分の議論は面倒になってしまったのだと思う？」
「んん——なんでだろう？」
ふとアリスのほうを見ると，彼女はただ静かにお茶を飲んでいた．魔理沙とわたしの会話に参加するつもりはなさそうだ．彼女の視線の先にあるミニチュアテーブルには，さらに一体の人形——魔理沙の姿の人形——が出現している．魔理沙の人形とわたしの人形はなにか話し合っている素振りをみせて小刻みに動いていた．

「微積分を創始したのはニュートンやライプニッツね．その後，微積分は大いに発展するのだけどジョージ・バークリーはその当時の微積分の理論の基礎にあった無限大や無限小についての非論理性を攻撃したのよ」
「非論理性――？」
「バークリーによる批判を説明する準備として，一変数関数 f の微分の復習をしましょう．関数 f の点 x_0 における微分値 $f'(x_0)$ は次のように定義されたのだったわね」:

$$f'(x_0) := \lim_{h\to 0} \frac{f(x_0+h)-f(x_0)}{h} \tag{3.1}$$

「うん，そうだね」
「ここに出てくる《極限を取る操作》$\lim_{h\to 0}$ の部分は，その右隣にある分数式を《変量 h の式》とみて，限りなく 0 に近づけていくことを意味しているんだよね」
「そうだね，h をどんどん小さくする」
「ところで，この極限に関する変量 h を《限りなく 0 に近づける》ならば，そんな面倒なことをせず最初から $h=0$ にしたらどうなるかしら？」
「それはおかしいよ，そんなんにしたら式 (3.1) の分母が 0 になっちゃう」
「その通り，だから h は 0 では絶対にいけない．かといって，h が段々小さくなる途中で有限の量――例えば 1 とか $-2/3$ のような――を取るのもなんか無駄だと思わない？」
「無駄，ってのはどういうこと？」
「つまり，いま極限をとりたい分数式を

$$\varphi(h) := \frac{f(x_0+h)-f(x_0)}{h} \tag{3.2}$$

と定義すると，欲しいのは

$$\lim_{h\to 0} \varphi(h) \tag{3.3}$$

であって，

$$\varphi(1), \varphi\left(\frac{1}{10}\right), \varphi\left(\frac{1}{100}\right), \varphi\left(\frac{1}{1000}\right), \varphi\left(\frac{1}{10000}\right)$$

といった量には興味がないのよ」
「まあ，極限ってのはそういうことだよね．h が $\dfrac{1}{10000}$ なんかよりずーっと小さくなっていった究極の値が欲しいんだから」
「さて，ある実数 $\varepsilon(\neq 0)$ を考えたとき，わたしたちは $|\varepsilon|$ より絶対値が小さくてしかも 0 でない実数を考えることができるわね」
「そうだね，実際 $|\varepsilon| > \left|\frac{1}{10}\varepsilon\right| > 0$ だから，$\frac{1}{10}\varepsilon$ はそういう実数――つまり《絶対値が $|\varepsilon|$ より小さくてしかも 0 でない実数》――の例だよね」
「そうね．これをもっと続けても

$$|\varepsilon| > \left|\frac{1}{10}\varepsilon\right| > \left|\frac{1}{100}\varepsilon\right| > \left|\frac{1}{1000}\varepsilon\right| > \ldots > \left|\frac{1}{10^n}\varepsilon\right| > 0 \qquad (n \geq 4) \tag{3.4}$$

となって，さらに小さな量をどんどん考えられるわね」
「うん」
「こうしてどんどん小さな数を作っていったその先に究極的に小さな実数――ただし 0 ではないもの――があるとは考えられないかしら？」

「上の式の $n \to \infty$ の極限を取るということ？」
「いいえ，そんなにふうにきちんと考えるのではなくて．ずうっと小さくなっていったその先に，ついには《究極の小ささ》だけが影のように残ってる——そんな感じ」
「うーん，なんか話がふわふわしている感じがする」
そのとき，しばらく黙っていたアリスが口を開いた．
「——無限小の量について，昔の偉人が実際に何を書き残しているか確かめてみるのもいい．たとえばオイラーはその 1748 年の著作『無限解析序説』の 7 章冒頭において次のように簡潔に無限小量を導入している：

> そこで ω は無限に小さい数，すなわち，どれほどでも小さくてしかも 0 とは異なる分数としよう．

この ω という文字はギリシャ語の最後の文字ね——究極の量を表すのにふさわしい」
「ありがとう，アリス．さて，もし仮にそのような量を考えられるのならば，微分はもはや極限を経由することなく

$$f'(x_0) = \frac{f(x_0+\omega) - f(x_0)}{\omega} \tag{3.5}$$

と《簡単に》表せることになるのよ」
「なるほど，ω は究極的に小さい量だけれど 0 ではないから分母に来てもいいわけだね」
「こんな風にして《無限小量》を用いれば，微分の定義は見かけ上簡単になるし，実際，多項式なんかを扱っているときには無限小量は計算上とても便利よ．例えば $f(x) := x^3$ でやってみると

$$\begin{aligned} f'(x_0) &= \frac{f(x_0+\omega) - f(x_0)}{\omega} = \frac{(x_0+\omega)^3 - x_0^3}{\omega} \\ &= \frac{x_0^3 + 3x_0^2\omega + 3x_0\omega^2 + \omega^3 \ - \ x_0^3}{\omega} \\ &= \frac{3x_0^2\omega + 3x_0\omega^2 + \omega^3}{\omega} \end{aligned}$$

ω は 0 ではないから除算が実行できて

$$= 3x_0^2 + 3x_0\omega + \omega^2$$

ω は限りなく 0 に近いので無視することが出来て

$$= 3x_0^2 \tag{3.6}$$

のようにして計算できるわ」
「えー，それおかしいよ．だって《ω は 0 ではないから除算が実行できて》と言った直後に《ω は限りなく 0 に近いので無視することが出来て》とか言ってるじゃん」
「そうね，いまの議論はどこかこじつけっぽいところが含まれているし，バークリーもそのような非合理性をその著書『解析者』において糾弾したのよ」
「うーん，無限小の議論に怪しいところがあったとしても，そこから”糾弾”ってところまでぶちきれるかな？」
「バークリーの意図は『解析者』のサブタイトル『不信心なる数学者へ向けられた談話．近代解析の目標，原理，そして推論が宗教的神秘と信仰の特徴よりも際立った着想または疑いようのない推論であるかどうかが検証されている．』からよりよく汲み取れると思うわ」

「ええっ，なんかバークリーって人やばそう……」
「どうやら，バークリーは人々が合理的な思考に溺れて神から離反するようなことが起こってはならない，そのような思考を広めてはならないという信念の持ち主だったみたいね．他にも《物質》を否定していたことでも知られているわ．つまりバークリーは，人々が《物質》を信じるようになればその分だけ信仰から遠くなる，だから否定しなければならないと考えたのね」
「ばかみたい．そこまで来ると，バークリーってただの変な人じゃん」
「さあ，どうでしょうね．昔の人間の評価を正しく行うというのは——その評価規準を恣意的に選んでしまう危うさに気づいてしまうと——なかなか難しいものよ」
「——バークリーの異常なほどの物質憎悪から，彼が便秘に悩まされていたのではないかと推測する人もいる」アリスが澄ました顔でそう付け加えると，魔理沙はけたたましく笑った．
「バークリー先輩便秘説ｗｗｗ」
神聖な図書館で便秘談義とは，なんと嘆かわしいガールズトークだろう．
気を取り直してわたしは続ける——．
「バークリーの今の批判も重要だけど，無限小量の実在性それ自体についても疑念がのこるわね」
「そうだね，無限小量 ω が 0 でないならさっきの議論と同じようにして

$$|\omega| > \left|\frac{1}{10}\omega\right| > 0$$

だから，ω は——自分よりさらに小さな数 $\frac{1}{10}\omega$ がある以上——《どれほどでも小さい》とは言えないよね」
「——無限小量や無限大量については，ライプニッツなんかも《架空の量》だと考えていたらしい」アリスが付け加えた．
「架空だけど便利だったから納得行ってないけど使い続けたってことかなぁ．いずれにせよ昔の微積分は《無限小量》や《無限大量》についての基礎の議論があやふやだったんだね」
「その後，微積分の基礎について関心が払われるようになり，コーシー; 彼自身は一貫して無限小量に基づいた議論をしていたのだけど——が残した《無限小量についての判定基準》をうまく用いて無限小量への言及を回避するというような努力がなされるようになったわ」
「——そのような解析学の再建に向けた努力が積み重ねられた結果，たとえば未だに広く読まれている解析学の教科書 E. T. Whittaker and G. N. Watson の “A Course of Modern Analysis”(初版 1902 年) では数列の収束について次のように定義が与えられるようになった：

Let $z_1, z_2, z_3, \ldots$ be an unending sequence of numbers, real or complex. Then, if a number l exists such that, corresponding to every positive number ε, no matter how small, a number n_0 can be found, such that

$$|z_n - l| < \varepsilon$$

for all values of n greater than n_0, *the sequence* (z_n) *is said to tend to the limit* l *as* n *tends to infinity*.

これは今日の大学の微積分の授業で扱うのと同じ形式．こんな風にして，無限小量に頼らない解析学の議論は普及していったと考えられる」アリスが付け加えた．
「なるほど，解析学の議論から無限小量は《追放》されたんだね」
「——念のために言えば，微積分の基礎の不確かさがすべて《無限小量》を用いた恣意的な論法と

関係していたわけではない．たとえば二項係数 $\binom{r}{k}$ を実数 r，整数 k に対して

$$\binom{r}{k} = \frac{r(r-1)\dots(r-k+1)}{k!}$$

として定義したときに，$|x|<1$ に対して

$$(1+x)^r = \sum_{k=0}^{\infty} \binom{r}{k} x^k$$

となるという《二項級数定理》というものがある．これを任意の実数 r に対してきちんとした証明を与えたのはアーベルが最初．アーベルは 1802 年生まれだから 1783 年に死んだオイラーよりちょっと後の人になる」

こう言ってお茶を一口飲むと，アリスはまた続けた．

「——指数 r が正の整数の場合にはこの《二項級数定理》は普通の二項定理に帰着する．ニュートンは 1664 年から 1665 年にかけて r に有理数——たとえば $\frac{1}{2}$ を代入することにより指数 r が有理数の場合の二項級数定理を《発見》した．その《発見》から，16 歳のアーベルが証明を行うまでの 150 年間は合理的な根拠なしに使われていたということになる」

「念のためにアリスの説明に補足しておくと，ニュートンが指数 r に有理数を代入してみたときのような思考自体はとても重要なものなのよ」

「発見的考察，っていうんだよね」魔理沙が言った．

「無限小量の《追放》は解析学の厳密化という大事件の中の重要なエピソードだけど，解析学の厳密化が無限小量の《追放》だけでなされたわけではないことには注意しておいてほしいわ」わたしは付け加えた．

3.3 述語論理

「さて，いまアリスが引用した Whittaker and Watson の引用箇所は，述語論理を用いれば次のように書き直せるわね」:

実数列または複素数列 $(z_n)_{n=1}^{\infty}$ が与えられているとする．次の条件

$$\exists l \ \forall \varepsilon > 0 \ \exists n_\varepsilon \ \forall n > n_\varepsilon \quad |z_n - l| < \varepsilon$$

が成立するとき《数列 $(z_n)_{n=1}^{\infty}$ は n が無限大に近づくと極限 l に近づく》という．

「どれどれ……うーんと，まあ，なんとなくそうなってるっぽいね」

3.3.1 述語論理で書かれた数学的な命題を読むこと

「ちょっと自信なさそうね……？　いま書いた論理式を読んで元の文をどれくらい復元できるか，やってみて」

「えーと，《存在して l が全ての $\varepsilon>0$ で》——あーなんかごちゃごちゃしてきてわかんなくなった」

「——述語論理の理解がこれほどあやふやだと超準解析を学ぶのは難しい」アリスが何食わぬ顔で厳しいことを言った．アリスの言ったことはまったく正しいのだが，可愛い魔理沙のためだからもう少し頑張ってみよう．

「そうね，IST の公理やその使用方法を理解するためにも，もう少し述語論理に慣れてほしいかな」

「なんか論理学の本とかって難しくて」

実のところ，述語論理は二段階で理解をする必要がある；最初に《初心者用》の理解をしておき，その上で《中級者用》の理解をする，という順で．《初心者向け》の述語論理の本で一番重要な事は，日常扱うような文章をどうやって記号に置き換えていくかという経験を積むことだ．例えば，表面的には少し似ているが全然違った状況を表す二つの文

(a) どんな人でも誰かを愛している．
(b) どんな人にも愛される誰かがいる．

の違いを，記号化を通して

(a') $\forall x\ \exists y \quad \mathrm{Loves}(x,y)$.
(b') $\exists y\ \forall x \quad \mathrm{Loves}(x,y)$.

の違いである，というような事として把握できるようになることが《初心者向け》の述語論理の重要な目標だ．この段階を経ていないと，《中級者用》の教科書を読むことは——原理的には不可能ではないものの——非常に困難だろう．述語論理について書いてある教科書の多くは，その意味では《本当の本当に初心者》に読みこなすのは難しいのだ．いま魔理沙が引っかかってるのはこの辺だ．そして，IST をやるためにはもう一段階，《述語論理の形式化》にある程度慣れていることが必要だろう．こちらの話題は，述語論理について書いてある教科書の冒頭付近に書いてある程度の簡単な事を知っていれば十分だ．そんなことを考えながらわたしは方針を立ててみた．
「そうね，《述語論理によって数学的な命題を読み書きする》ことが本当は望ましいのだけど速習コースということで，《述語論理で書かれた数学的な命題を読むこと》だけに絞って説明してみましょう．こんな手順をとれば述語論理で書かれた命題を理解できるはずよ」:

1. 論理記号にルビを振る．
   - $\forall$ → for all
   - $\exists$ → exists
   - $\neg$ → not
   - $\wedge$ → and
   - $\vee$ → or
   - $\Longrightarrow$ → then または implies
   - $\Longleftrightarrow$ → if and only if または is equivalent to
2. 論理記号をルビに置き換えてゆき，英文らしきものをつくる．この際，必要に応じて括弧やコンマや such that を補う．
3. 日本語に訳す

「うーん，なんか漢文なんかを読むときみたいな感じ？」
「似てるかもね．では実際にやってみましょう．元の述語論理の命題は

$$\exists l\ \forall \varepsilon > 0\ \exists n_\varepsilon \quad \forall n > n_\varepsilon \quad |z_n - l| < \varepsilon.$$

で，これにいま説明した流儀でルビを振ると

$$\overset{\text{exists}}{\exists}\ l \quad \overset{\text{for any}}{\forall}\ \varepsilon > 0 \quad \overset{\text{exists}}{\exists}\ n_\varepsilon \quad \overset{\text{for any}}{\forall}\ n > n_\varepsilon \quad |z_n - l| < \varepsilon.$$

論理記号をルビに置き換えいってから括弧を補うと

$$\mathrm{exists}\, l\ \ (\mathrm{for\ any}\, \varepsilon > 0\ \ (\mathrm{exists}\, n_\varepsilon\ \ (\mathrm{for\ any}\, n > n_\varepsilon \quad |z_n - l| < \varepsilon))).$$

いちばん内側の括弧から順に，機械的に日本語に置き換えていくと

→ exists l (for any $\varepsilon > 0$ (exists n_ε (いかなる $n > n_\varepsilon$ に対しても $|z_n - l| < \varepsilon$))).
→ exists l (for any $\varepsilon > 0$ (ある n_ε が存在して(いかなる $n > n_\varepsilon$ に対しても $|z_n - l| < \varepsilon$))).
→ exists l
(いかなる $\varepsilon > 0$ に対しても(ある n_ε が存在して(いかなる $n > n_\varepsilon$ に対しても $|z_n - l| < \varepsilon$))).
→ある l が存在して
(いかなる $\varepsilon > 0$ に対しても(ある n_ε が存在して(いかなる $n > n_\varepsilon$ に対しても $|z_n - l| < \varepsilon$))).

のようになるわね」
「《いかなる $n > n_\varepsilon$ に対しても》のあたりは，n が後ろから修飾されていてちょっと日本語の語順からすると収まりがわるいかな．ちゃんとした日本語の語順にすると《n_ε より大きい，いかなる n に対しても》の方がいいかもしれないな」
「ええ，確かにいまの機械的な操作で作った日本語の文は名文名調子とはとてもいかないけれど，意味を拾うには十分役立つわ」
「まあ，一応はね……」
「慣れれば，こういうプロセスをすべて頭のなかでやってしまうこともできるようになるわよ」

3.3.2 述語論理を用いた形式的推論

「さて，これで《述語論理で書かれた数学的な命題を読むこと》はできたことにしましょう」
「述語論理ってなんかいまみたいな面倒な形で文章を書くことが目的なの？」
「いいえ，この形式で書くことは——面倒かどうかは慣れの問題ね——単なる手段よ．そして目的は，形式的な推論や思考の分析を行うことよ」
「形式的な推論？」
「たとえばド・モルガンの法則

$$\begin{aligned} \neg(A \wedge B) &\rightleftharpoons (\neg A) \vee (\neg B). \\ \neg(A \vee B) &\rightleftharpoons (\neg A) \wedge (\neg B). \end{aligned}$$

なんかを利用して，否定命題の分析を行えるわ」
「ええと，最初のやつは，《(A かつ B) ではない》ことと《(A ではない) または (B ではない)》ことが一緒ということだね」
「実際に A や B がどんな内容であるかを問わずにこのような法則が成り立ってるわ．いまの法則を

- ((A かつ B) ではない) ことと ((A ではない) または (B ではない)) ことは同値である
- ((A または B) ではない) ことと ((A ではない) かつ (B ではない)) ことは同値である

という感じに言葉で書き下すと，意味を取るのは結構大変よ」
「確かに，記号で書いてあると機械的に計算できそうだね」
「そう．論理的思考という負荷の高い操作を，記号操作という代数的な操作で置き換えていくことでいくらか楽にしようということね」
「なるほど」
「ド・モルガンの法則には，量化子が関わるものもあるわ」:

$$\begin{aligned} \neg(\forall x A(x)) &\rightleftharpoons \exists x (\neg A(x)). \\ \neg(\exists x A(x)) &\rightleftharpoons \forall x (\neg A(x)). \end{aligned}$$

「最初のやつは，《(いかなる x についても $A(x)$ である) ではない》ことと《ある x が存在して ($A(x)$ ではない)》ことが一緒だということだね．なるほど，反例があるってことが《いかなる……》の否定と同じだってことか」

「ド・モルガンの法則を使えば，《外側についた否定を内側に送り込む》ことができるのよ．では試しに《数列 $(z_n)_{n=1}^{\infty}$ は n が無限大に近づくと極限 l に収束する》の否定文でやってみましょうか？」
「うん，ちょっとやってみるね．

$$\begin{aligned} & \neg\ \forall\varepsilon>0\ \exists n_\varepsilon\ \forall n>n_\varepsilon \quad |z_n-l|<\varepsilon. \\ \rightleftharpoons\ & \exists\varepsilon>0\ \neg\ \exists n_\varepsilon\ \forall n>n_\varepsilon \quad |z_n-l|<\varepsilon. \\ \rightleftharpoons\ & \exists\varepsilon>0\ \forall n_\varepsilon\ \neg\ \forall n>n_\varepsilon \quad |z_n-l|<\varepsilon. \\ \rightleftharpoons\ & \exists\varepsilon>0\ \forall n_\varepsilon\ \exists n>n_\varepsilon \quad \neg(|z_n-l|<\varepsilon). \end{aligned}$$

さらに否定を内側に送り込むと不等号が逆になって

$$\rightleftharpoons\ \exists\varepsilon>0\ \forall n_\varepsilon\ \exists n>n_\varepsilon \quad |z_n-l|\geq\varepsilon.$$

となるね．合ってる？」
「うん，形式は合ってるけど意味を取りやすくするためには $\forall$ にかかってる変数からは添字を外した方がいいかもね」：

$$\exists\varepsilon>0\ \forall n\ \exists m>n \quad |z_m-l|\geq\varepsilon.$$

「どれどれ，これを読み下すと：

ある $\varepsilon>0$ が存在して (いかなる n に対しても (ある $m(>n)$ が存在して $|z_m-l|\geq\varepsilon$ となる))

となるな」
「つまりこれはどういうことだろう？」
「まず，なんか正の数 ε があるって言ってる．で，なんでもいいから n を選ぶと，それより大きな m があって，z_m は l の ε 近傍からはみ出すって言ってるんだよね」
「《数列 $(z_n)_{n=1}^{\infty}$ が l に収束しない》ということから今の条件を引き出すまでの過程では，論理式の記号操作だけで行けたでしょ．これってすごいと思わない？」
「確かに，記号だけ操作して《論理的に考える》ってことをしてないのにこんなのが出来るのはすごいかも」
「さて，こういうことができるための基盤ってなんでしょうね？」
「基盤？」
「——論理式とは何だと定義されるのか，また論理式に許される操作はどのように規定されているのか」アリスがぼそっと呟いた．
ふとミニテーブルを見やると，魔理沙の人形とわたしの人形は丁度わたしたちがそうしているように隣り合って座っており，ミニテーブルに広げたノートを囲んでしきりにこくこくと頷き合ったりしていた．アリスの人形はというと，これまたアリス本人とそっくりな仕草で静かにお茶を飲んでいるのだった．

3.3.3 述語論理の形式化

「これまでのところ，《論理式とは何か》というようなことはきちんと定義しないで議論してきたけれど，いまからこれを形式化することについて話すわ」
「うーん，なんで形式化したいんだろう？」
「たとえば《論理式全体の集合》のようなものをきちんと定めておくことで《論理式の操作》の意味が明確になるのよ」
「——形式化のもう一つの恩恵は機械化．論理を十分に形式化すれば，論理式の操作をするプログラムを書けるようになる」アリスが言った．
「そうね，すべての数学がアルゴリズム的である必要はないけど，何かのアルゴリズムを書けるほ

どに操作を明確にすることは，その分野についての理解を深めてくれるわ」
「形式化ってなんか難しそうなイメージがあるんだよね．《形式化するとより深く理解できる》って視点はピンとこないな〜」
「何かを形式化してゆく過程で，なんとなく理解していたり，勘や経験や常識を働かせて処理していた部分をより明確にしなければならなくなるのよ——そういうことはいつでも簡単なわけじゃないけど．でも，うまくいくと，いままでは《なんとなくやっていた》ことをより明確に説明できるようになるわけ．そうやって理解が深まるのよ」
「——述語論理はわたしたちの論理的思考の形式化だと言える．論理からさらに視野を広げて，感情や意志までも形式化すればそれをアルゴリズムに書き下し，プログラムとして実行できるようになる．わたしの人形たちは，基本的にはそんな考えにもとづいて設計され，動いている」アリスがこともなげに言った．
「じゃあ，その子たちには意志や感情があるということ？」
「——わたしが形式化した範囲での意志と感情だけどね．わたしの理解の及ばない部分については形式化出来ていない」
アリスの人形の高い自律性については紅魔館でも話題になることがあったが，アリス自身の口からその秘密が語られるのを聞くのは初めてだった．
「それにしても，意志や感情まで形式化できるなんてすごいなぁ」魔理沙が素直に賛嘆した．
人間の意志や感情を形式化するというのはかなり奇妙な話に思える．そもそも，そうして形式化したものを動かしたものは《本物の意志》や《本物の感情》なのだろうか？　しかし人間の心の機能のうち，《論理》に関わるものだけが形式化を許し，《意志》や《感情》は形式化を拒絶するとしたらそれはそれでまた奇妙だという気がした．
「ねえ，わたしも興味あるわ．一体どうやって意志や感情を形式化したの？」
「——誰にでも秘密はある」アリスが即答した．
「まあ，そりゃそうかー」
魔術は膨大な試行錯誤の賜物だ．大成しようと思ったら通常の人間に与えられた寿命ではとても時間が足らない．そんなわけで，魔術を極めようとするものはまず《長命の術》や《不死の術》を目指すものが多いのだ．そうしてまで得た貴重な知識や経験は，軽々しく他人に伝授できるものではない．ここは素直に諦めよう．
「では話を述語論理の形式化に戻しましょう．述語論理は，わたしたちの論理的な思考を扱うものだから，その形式化の対象は《わたしたちの言語の論理的な部分》ということになるわね．《形式化》なんていうと難しそうだけど，要するに《意味》を忘れてしまっても物事が表面上うまくいくぐらいに記号化するということなのよ」
そう言ってからわたしは次のようにノートに書きだした：

定義 (一階述語論理の言語)
一階述語論理の言語 $\mathscr{L}$ は次のものからなる：

(a) 集合 Var．この集合の要素を《変数》と呼ぶ．
(b) 集合 Const．この集合の要素を《定数》と呼ぶ．
(c) 集合 $\mathsf{Pred} = \bigcup_{n=1}^{\infty} \mathsf{Pred}_n$．この集合の要素を《述語》と呼ぶ．
(d) 《論理記号》と呼ばれる特別な記号 $\boxed{\forall}$, $\boxed{\exists}$, $\boxed{\neg}$, $\boxed{\wedge}$, $\boxed{\vee}$, $\boxed{\Rightarrow}$, $\boxed{\Leftrightarrow}$, $\boxed{=}$.

上記の (a)(b)(c)(d) の要素には交わりがないものとする．

「ここでは，まず言語のアルファベットに相当するものを定義したことになるわ」
「論理記号が四角で囲まれてるのはなんでだろう？」

「わたしたちは論理記号を自然言語の一部として使うこともあるわ．対象となる言語とわたしたちの言語で同じ記号を使っていると混乱するわよね．だから，わたしたち自身の言語ではなく，それを記号化して扱っていることを強調して四角で囲っておいたの．さて，言語 $\mathscr{L}$ の《項》と《論理式》を定義するわ」

定義 (言語 $\mathscr{L}$ の項と論理式)

一階述語論理の言語 $\mathscr{L}$ の項の集合 $\mathsf{Term}_{\mathscr{L}}$ を次のように定義する：

$$\mathsf{Term}_{\mathscr{L}} := \mathsf{Var} \cup \mathsf{Const}.$$

一階述語論理の言語 $\mathscr{L}$ の論理式の集合 $\mathsf{Form}_{\mathscr{L}}$ を次のように定義する：

$$\mathsf{Form}_{\mathscr{L}} := \bigcup_{k\geq 0} \mathsf{Form}_{\mathscr{L}}^{(k)},$$

ただし

$$\begin{aligned}
\mathsf{Form}_{\mathscr{L}}^{(0)} &:= \left\{ \langle \boxed{=}, t, u\rangle \;\middle|\; t,u \in \mathsf{Term}_{\mathscr{L}} \right\} \\
&\qquad \cup \bigcup_{n\geq 1} \left\{ \langle p, u_1, \ldots, u_n\rangle \;\middle|\; p \in \mathsf{Pred}_n, u_1, \ldots, u_n \in \mathsf{Term}_{\mathscr{L}} \right\}, \\
\mathsf{Form}_{\mathscr{L}}^{(k+1)} &= \mathsf{Form}_{\mathscr{L}}^{(k)} \cup \left\{ \langle \boxed{\forall}, x, A\rangle \;\middle|\; x \in \mathsf{Var}, A \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\exists}, x, A\rangle \;\middle|\; x \in \mathsf{Var}, A \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\neg}, A\rangle \;\middle|\; A \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\wedge}, A, B\rangle \;\middle|\; A,B \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\vee}, A, B\rangle \;\middle|\; A,B \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\Rightarrow}, A, B\rangle \;\middle|\; A,B \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \\
&\qquad \cup \left\{ \langle \boxed{\Leftrightarrow}, A, B\rangle \;\middle|\; A,B \in \mathsf{Form}_{\mathscr{L}}^{(k)} \right\} \qquad (k \geq 0).
\end{aligned}$$

「ここに出てきた $\langle \boxed{=}, t, u\rangle$ は普通の中置形式で書けば $t \boxed{=} u$ となるわ」

「操作の記号が前に置いてあると，なんか Lisp っぽいね」

「——論理記号を前置して論理式を扱うことを最初に思いついたのはポーランド人のウカシェヴィチ．Lisp は彼のアイデアを利用しているので，この形式化が Lisp に似ているというのは正しい——話の順序はねじれてるけど」アリスがぼそりと呟いた．

「ここでは論理式の集合 $\mathsf{Form}_{\mathscr{L}}$ を帰納的に構成したけれど，結局のところ次のような性質が成り立つことが重要ね」：

- 任意の $t,u \in \mathsf{Term}_{\mathscr{L}}$ に対して　$\langle \boxed{=}, t, u\rangle \in \mathsf{Form}_{\mathscr{L}}$.
- 任意の $P \in \mathsf{Pred}_n$ と任意の $u_1, ..., u_n \in \mathsf{Term}_{\mathscr{L}}$ に対して　$\langle p, u_1, \ldots, u_n\rangle \in \mathsf{Form}_{\mathscr{L}}$.
- 任意の $x \in \mathsf{Var}$ と任意の $A \in \mathsf{Form}_{\mathscr{L}}$ に対して　$\langle \boxed{\forall}, x, A\rangle \in \mathsf{Form}_{\mathscr{L}}$.
- 任意の $x \in \mathsf{Var}$ と任意の $A \in \mathsf{Form}_{\mathscr{L}}$ に対して　$\langle \boxed{\exists}, x, A\rangle \in \mathsf{Form}_{\mathscr{L}}$.
- 任意の $A \in \mathsf{Form}_{\mathscr{L}}$ に対して　$\langle \boxed{\neg}, A\rangle \in \mathsf{Form}_{\mathscr{L}}$.
- 任意の $A,B \in \mathsf{Form}_{\mathscr{L}}$ に対して　$\langle *, A, B\rangle \in \mathsf{Form}_{\mathscr{L}}$　$(* = \boxed{\wedge}, \boxed{\vee}, \boxed{\Rightarrow}, \boxed{\Leftrightarrow})$.

「Haskell とかに慣れてると，こういう再帰的な構成は理解しやすいよね」

「さて，述語論理については《自由変数》までやって，おしまいにしましょう．論理式 A に対して，《自由変数》の集合 $\mathrm{FV}(A) \subseteq \mathsf{Var}$ を次のように定義するわ」：

$$\begin{aligned}
&\mathrm{FV}\colon \mathsf{Form}_{\mathscr{L}} \longrightarrow \mathscr{P}(\mathsf{Var}) \\
&\mathrm{FV}(A) \\
&\quad = \begin{cases}
\left(A \in \mathsf{Form}_{\mathscr{L}}^{(0)}\right) & : \begin{cases} (A = \langle \boxed{=}, t, u\rangle) & : \mathrm{termFV}(t) \cup \mathrm{termFV}(u) \\ (A = \langle\, p, u_1, \ldots, u_n\rangle) & : \mathrm{termFV}(u_1) \cup \cdots \cup \mathrm{termFV}(u_n) \end{cases} \\
(A = \langle \boxed{\forall}, x, B\rangle) & : \mathrm{FV}(B) - \{x\} \\
(A = \langle \boxed{\exists}, x, B\rangle) & : \mathrm{FV}(B) - \{x\} \\
(A = \langle \boxed{\neg}, B\rangle) & : \mathrm{FV}(B) \\
(A = \langle \boxed{\wedge}, B, C\rangle) & : \mathrm{FV}(B) \cup \mathrm{FV}(C) \\
(A = \langle \boxed{\vee}, B, C\rangle) & : \mathrm{FV}(B) \cup \mathrm{FV}(C) \\
(A = \langle \boxed{\Rightarrow}, B, C\rangle) & : \mathrm{FV}(B) \cup \mathrm{FV}(C) \\
(A = \langle \boxed{\Leftrightarrow}, B, C\rangle) & : \mathrm{FV}(B) \cup \mathrm{FV}(C)
\end{cases} .
\end{aligned}$$

ただし

$$\begin{aligned}
&\mathrm{termFV}\colon \mathsf{Term}_{\mathscr{L}} \longrightarrow \mathscr{P}(\mathsf{Var}) \\
&\mathrm{termFV}(s) \\
&\quad = \begin{cases} (s \in \mathsf{Var}) & : \{s\} \\ (s \in \mathsf{Const}) & : \varnothing \end{cases} .
\end{aligned}$$

「この関数の定義の記法は見慣れないけど，これは Haskell のガード式みたいに読めばいいのかな」
「ええ，そんな風に読んでくれると助かるわ．構造帰納法にもとづいて関数を定義するとき，こういう記法は便利ね」
「こうやって論理式を形式的に定義したことで何が明確になったんだろう？」
「まず，論理式の集合 $\mathsf{Form}_{\mathscr{L}}$ が明確に定義されたことで，何が論理式であるかがはっきり定まったわ．つまり，論理式というものが項からどんな風に組み立てられているのか、という形で構造帰納法によって明確に定義されるようになったわ」
「なるほど」
「さて，こういう定義をすると論理式の木構造は明確になるけれど，この通りに書くのは煩雑なので略記することにしましょう．略式記法では，今まで《$\boxed{\forall}$》と書いていたのを《$\forall$》に戻すことにするわ (表 3.1). 」

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

表 3.1: 論理式の定義通りの書式と略式記法

「せっかく論理式をカッチリと定義したのに略記に戻しちゃうのはなんか損した感じがするな……？」

「必要に応じてこの形式に戻って議論できることさえ覚えておけばいいのよ」
「なるほどね、普段は略式記法を使って、論理式についての議論が混乱しそうなときだけカッチリした形式で議論すればいいんだね」
「略式記法での論理式の自由変数の扱いについて補足しておくわ．述語 $p \in \mathsf{Pred}_n$ と変数 $x_1,...,x_n \in \mathsf{Var}$ に対して，

$$\mathrm{FV}\,(p(x_1,...,x_n)) = \{x_1,...,x_n\}$$

となることはいいかしら」
「えーと，定義をたどるとそうなるね」
「一般の論理式は，$t = u$ の形の論理式や $p(x_1,...,x_n)$ の形の論理式を量化子 $(\forall,\exists)$ や論理結合記号 $(\neg,\wedge,\vee,\Longrightarrow)$ で結んで作られたのだったわね．さて，そのようにして得られる一般の論理式 A に対して，$\mathrm{FV}(A) = \{x_1,...,x_m\}$ であることを

$$A(x_1,...,x_m)$$

と書くことで表すことがあるわ．それから，多くの自由変数を持つ論理式

$$B(x,y,t_1,...,t_n)$$

があったとき，一部の自由変数だけ書いて

$$B(x,y)$$

のように書くこともあるわ」
「そんなにいつも省略していいの？」
「——IST の公理図式を扱うときには，論理式の自由変数をすべて表示しておきたい場面もある」アリスが言った．
「必要ならば議論を正確に行えるということが大事ね——実際，すべての自由変数を常に表示しておくというようにすれば煩雑だけど間違いを犯す危険は減るわ」
「なるほどね」
「それから，これは定義から明らかだけど論理式を量化子で修飾すると自由変数は減るわね」
「えーと……？」
「たとえば，$\mathrm{FV}(A) = \{x,y_1,...,y_m\}$ のとき

$$\begin{aligned}\mathrm{FV}\big(\forall x\ A(x,y_1,...,y_m)\big) &= \mathrm{FV}\big(\langle\boxed{\forall},x,A(x,y_1,...,y_m)\rangle\big)\\ &= \mathrm{FV}\big(A(x,y_1,...,y_m)\big) - \{x\}\\ &= \{y_1,...,y_m\}.\end{aligned}$$

となるわね」
「うん，そうなるね」
「こうして，量化子によって《減らされた》変数を《束縛変数》と呼ぶわ」
「束縛変数も自由変数と同じようにして定義できそうだね」
「ええ，自分でやってみるといいわ」
「——ちゃんとした論理学の本には必ず載ってる．迷ったら見るといい」アリスが呟いた．
「さて，IST の話をする準備は整ってきたけれど，その前に公理的集合論 ZFC についてざっと眺めておきましょう」
「IST ってのに早く行かない？ もう随分話してる気がするんだけど」
「——ZFC の理解があやふやだと IST の理解まであやふやになる」アリスが言った．
「IST は ZFC に公理図式を幾つか追加したものだわ．そういう意味では，ZFC をまったく知らない

という人が IST を正しく使えるものかといえばちょっと疑問ね」
「なるほど，そんなもんか．でもちょっと休憩入れてからにしようよ」そう言って魔理沙がスカートのポケットから小さなクッキー缶を取り出した．それからわたしたちはささやかな茶会を楽しんだ．

3.4 公理的集合論 ZFC

「さて，では IST の準備として公理的集合論 ZFC の話をしましょう．さっきも言ったように，ZFC の公理はすべて IST の公理でもあるので一通り公理を眺めておくのは重要だと思うわ」

3.4.1 ZFC の言語

「まずは ZFC の言語をはっきりさせましょう．ZFC の言語 $\mathscr{L}_{\mathrm{ZFC}}$ を次のように定義するわ」:

- $\mathsf{Const} = \{\boxed{\varnothing}\}$.
- $\mathsf{Pred}_2 = \{\boxed{\in}\}$.

「言語 $\mathscr{L}_{\mathrm{ZFC}}$ の定数記号はただ一つ $\boxed{\varnothing}$ で，述語記号は arity 2 の記号 $\boxed{\in}$ だけだということだね」
「ここで，$\boxed{\varnothing}$,$\boxed{\in}$ のように箱で囲った意図はわかるかしら」
「たとえば $\boxed{\varnothing}$ について言えば，《わたしたちの世界の $\varnothing$》じゃなくて，《対象言語の定数記号である $\varnothing$》を指したかった，から？」
「うん，そのとおり．でもこんなふうに書くのは煩雑だから，以下では誤解が起こらないかぎり $\boxed{\varnothing}$,$\boxed{\in}$,$\boxed{=}$ を $\varnothing$,$\in$,$=$ とそれぞれ略すことにするわ．
それにあわせて，$\langle\boxed{\in}, x, y\rangle$ は《$x \in y$》と，$\langle\boxed{=}, x, y\rangle$ は《$x = y$》と，どちらにも中置記法を用いることにするわ」
「メタ言語での $\varnothing$,$\in$,$=$ と迷いそうなら自分で四角の囲みをつけて区別してみるといいかもね」
「それはいい考えだと思うわ」

3.4.2 ZFC の公理（と公理図式）

「さて，これから ZFC の公理を紹介していくわ．最初は等号公理ね」

等号公理 1

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

「言葉で言えば《いかなる集合もそれ自身に等しい》ということになるわね」

等号公理 2

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

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

「この《等号公理 1》《等号公理 2》は，等号に関する論理公理の方に入れることが多いけど，今回は論理公理の扱いを省略したので集合論の公理に入れておくことにしたの」
「——《等号公理 2》は実際には《公理》ではなく《公理図式》になっている」
確かにアリスの言うとおりなので，公理図式の説明をしておこう．
「いまの《等号公理 2》は，$\mathscr{L}_{\mathrm{ZFC}}$ の論理式 A をパラメータとして持っていたわね．1 個以上の自由変数を持つ $\mathscr{L}_{\mathrm{ZFC}}$ の論理式は——その構成過程から明らかなように——無限個あるので，結局無限

個の公理を与えたことになるわ．このように論理式パラメータを持つような公理を，単一の《公理》と区別して《公理図式》と呼ぶことがあるわ」
「図式ってのは英語で言うと diagram?」
「いいえ， schema よ」
「——《代入》の説明も省かれている」アリスが呟いた．
「アリスの言うとおり，今回は論理式への変数の代入も扱わなかったわ」
「代入ってのは $A(x,t_1,...,t_n)$ から $A(y,t_1,...,t_n)$ をつくるところ？」
「ええ，そのとおり．代入をきちんと定義するというのはそれなりに面倒だけど，プログラミング言語で形式言語の処理系を作るわけでもなければそれほど神経質にならなくてもいいと思うわ」
「逆に言うと，プログラミング言語で論理式の処理をしたい人は気にしたほうがいいわけか」
「——代入の正確な定義が気になるなら，たとえば巻末参考文献に挙げた Gallier の本を見るといい」アリスがメタ発言をした．
「さて次は外延性公理ね」

外延性公理

$$\forall x\ \forall y\ \Big(\big(\forall t\ (t\in x \iff t\in y)\big) \implies x=y\Big).$$

「この公理は，等号 $\boxed{=}$ と二項関係 $\boxed{\in}$ の関係を述べたものになっているわ」
「意味を読み取ろうとすれば，２つの集合 x,y があったとき，どんな t についても

$$t\in x \iff t\in y$$

ならばそこから $x=y$ が出てくるわけか」
「——さっきの等号公理をうまく組み合わせて使うと外延性公理の《$\implies$》の前後をひっくり返したもの

$$\forall x\ \forall y\ \Big(x=y \implies \big(\forall t\ (t\in x \iff t\in y)\big)\Big).$$

を示せる」アリスが言った．
「さて，次は空集合公理ね」

空集合公理

$$\forall x\ \ x\notin \varnothing.$$

「この公理は言語 $\mathscr{L}_{\mathrm{ZFC}}$ の定数記号 $\boxed{\varnothing}$ の性質を規定しているわ」
「ここの $x\notin\varnothing$ は $\neg(x\in\varnothing)$ の略記だね」
「そうね．ところで，$\varnothing$ の性質を持つ集合がただひとつ存在することは言えるかしら？」
「《存在すること》はもう公理で保証されているから，《存在するとしたら一つしかない》の部分を言えばいいんだよね．でもどうやるんだろう？」
「魔理沙の言った方針で，一意性の部分の証明はたとえばこんなふうに出来るわ．まず $\mathrm{NULL}(x)$ という述語を

$$\mathrm{NULL}(x) \equiv \forall t\ \ (t\notin x).$$

として定義してやる．すると

$$\mathrm{NULL}(x)\wedge\mathrm{NULL}(y) \implies x=y.$$

が外延性公理を使って示せるわ」
「なるほど．空集合公理はこの述語で

$$\mathrm{NULL}(\varnothing).$$

と言い換えられるね」
「——今回の議論では，推論規則を明確に導入しなかったので《証明とは何か》ということが若干曖昧かもしれない」アリスがコメントした．
「確かにそうね．さて，ではついでによく使われる省略記法を導入するわ」:

$$x \subseteq y \equiv \forall t \ \ (t \in x \implies t \in y).$$

「これは集合の包含関係の記法だね」
「外延性公理を使えば次も示せるわ」:

$$x = y \iff (x \subseteq y) \wedge (y \subseteq x).$$

「量化子に関する省略記法もここで紹介しておくわ」:

$$\begin{array}{lcl} \forall x \in z \ Q(x,z) & \equiv & \forall x (x \in z \implies Q(x,z)). \\ \exists x \in z \ Q(x,z) & \equiv & \exists x (x \in z \wedge Q(x,z)). \end{array}$$

「《$\forall x \in z \ Q(x,z)$》を英語式に読み下すと《for all x in z, $Q(x,z)$ holds》って感じかな．そして《$\forall x (x \in z \implies Q(x,z))$》を英語式に読み下すと《for all x, ($x \in z$ implies $Q(x,z)$) 》って感じか．確かに《同じこと》だな」
「述語論理は西洋で生まれたものだから，どうしても印欧語に置き換えて読むほうが意味を取りやすいわね」
「なるほど，そんなもんか」
「さて，次は分出公理図式ね」
「公理図式，だから無限個の公理だね」

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

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

「うわ，なんか複雑すぎて意味がわからない」
「こういう論理式から意味をとるときのコツは，まず前半の $\forall u_1 ... \forall u_n$ を無視してみることね」
「——あるいは $n = 0$ の場合，つまり論理式 A の自由変数が t だけの場合で考えてみるのもいい」アリスが言った．
「そうね，では $n = 0$ の場合を書き下してみましょう」:

$$\forall x \ \exists y \ \forall t \ \ \big(t \in y \iff t \in x \wedge A(t) \big)$$

「これだったらなんとかなるかな．《いかなる x に対しても (ある y が存在して (いかなる t に対しても ($t \in y$ であることと ($t \in x$ かつ $A(t)$) であることは同値)))》だね」
「つまり，この集合 y は，集合 x の要素 t のうち $A(t)$ となってるものだけを集めるようにして出来ていることになるわ」
「x から欲しいところだけ抽出する感じだね．なるほど，だから《分出》なのか．ちなみにこの公理図式の名前を英語で言うと……？」
「axiom schema of separation よ——ほかにも幾つか異名はあるけどね」

「いまは $n=0$ の場合をかんがえてみたけど，他の場合も $u_1,...,u_n$ が入るだけで《基本的には同じ》と考えればいいわ」
「なるほどね，前のほうに全称量化子がたくさん並んでいても《最初無視して，あとでちょっと考える》ようにすればいいのか！」
「分出公理図式で保証を存在された集合 y を

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

と表記するわ」
「なるほど，公理で書いてると難しいけどこの書き方ならわかる感じだ」
「さて，次は非順序対の公理よ」

非順序対の公理

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

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

$$\{x,y\}$$

と書いて，《集合 x,y から作られる非順序対》と呼ぶわ」
「非順序対は，二つしか要素を持たない集合だね」
「特別な場合として，$\{x,x\}$——つまり x と y が同じ場合——のことは

$$\{x\}$$

と書くことに決めておくわ」
「なるほど，非順序対の公理で x と y に同じ x_0 を代入すると

$$\exists z_0\ \forall t\ \ (t \in z_0 \iff t = x_0)$$

となるから，確かにこの $z_0(=\{x_0\})$ は一つだけ要素を持つ集合になるね」
「この非順序対を使って，集合 x,y の《順序対》$\langle x,y\rangle$ が次のように定義されるわ」：

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

「けっこう複雑だな」
「要するに重要なのは次の性質が成り立つことよ——証明はここではしないけど」：

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

「なるほど，簡単な道具を組み合わせてだんだん便利なものを作ってく感じだね」
「順序対が定義されたので，《関係》や《関数》を集合論で扱うための言葉もそろいつつあるわ．とりあえずここでは三つの述語を用意しておきましょう」：

$$\begin{aligned}
&\mathrm{RELATION}(f) &&\equiv\quad \forall t \in f\ \exists u\ \exists v\ \ (t = \langle u,v\rangle).\\
&\mathrm{SINGLEVALUED}(f) &&\equiv\quad \forall x\ \forall y_1\ \forall y_2\ \ \big(\langle x,y_1\rangle \in f \wedge \langle x,y_2\rangle \in f \implies y_1 = y_2\big).\\
&\mathrm{FUNC}(f) &&\equiv\quad \mathrm{RELATION}(f) \wedge \mathrm{SINGLEVALUED}(f).
\end{aligned}$$

「RELATION(f) が言ってるのは集合 f が順序対の集まりになってるということだね．そして，SINGLEVALUED(f) が言っているのは集合 x に対して，$\langle x,y\rangle \in f$ となってるような集合 y があったとしたらそういう y は一個しかないということだね」

「そのとおり．RELATION(f) は《f が関係であること》を意図して定義されているわ．そして FUNC(f) は SINGLEVALUED(f) と組み合わせて，《f が関数であること》を意図して定義されているの．さて，では和集合公理の説明をするわ」

和集合公理

$$\forall x\ \exists y\ \forall t\ \big(t \in y \iff \exists s \in x\ \ (t \in s)\big).$$

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

$$\bigcup x$$

と書いて《x の和集合》と呼ぶわ．ただし，集合 x が《添字付け》されていて

$$x = \{a_i \,|\, i \in I\}$$

のようになっている場合は $\bigcup\{a_i \,|\, i \in I\}$ のことを

$$\bigcup_{i \in I} a_i$$

と書いたりするわね」
「こういう書き方は上の方でも使ってるよね」
「それから，$x = \{t,u\}$ のときは $\bigcup\{t,u\}$ を

$$t \cup u$$

と書く決まりね．さらに，分出公理図式によって $t \cup u$ から t と u の共通部分 $t \cap u$ が

$$t \cap u := \{s \in t \cup u \,|\ \ s \in t \wedge s \in u\}$$

と定義されるわ．同じようにして，和集合公理と分出公理図式を組み合わせて集合 x から $\bigcap x$ を定義できるわ」
「こんな感じに定義すればいいかな」:

$$\bigcap x := \Big\{t \in \bigcup x \,\Big|\, \forall s \in x\ \ (t \in s)\Big\}.$$

」
「そうね．こうして，だんだん集合の演算がそろってきたわね．さて，では次に無限公理を紹介するわ」

無限公理

$$\exists x\ \ \big(\varnothing \in x\ \ \wedge\ \ \forall t \in x\,(t \cup \{t\} \in x)\big).$$

「これは何が言いたいんだろう？」
「ある集合 x があって，まず $\varnothing \in x$ となってるわね」
「うん」
「$\varnothing \in x$ だから後半の条件から $\varnothing \cup \{\varnothing\} = \{\varnothing\} \in x$ となる」
「たしかに」
「$\{\varnothing\} \in x$ だから後半の条件から $\{\varnothing\} \cup \{\{\varnothing\}\} \in x$ となる」

「もうすでにめんどくさくなってきた」
「$\varnothing$ からはじめて，この過程をずっと続けることができるわ」
「なるほど，だから無限公理によって存在を保証された集合 x には無限個の要素が入ってるわけか」
「——実際にこの操作でつねに新しい集合ができることは基礎の公理からわかる」アリスが言った．
「ではここで基礎の公理を紹介しておきましょう」

基礎の公理

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

「集合 t から $t \cup \{t\}$ を作ったとき，もとの t と一致してしまったとしましょう：

$$t = t \cup \{t\}.$$

するとここから $t \in t$ が導かれるわね」
「なんか $t \in t$ だと $t \in t \in t \in t \in t \in t \in t \in \ldots$ ってなってオレンジのポータルとブルーのポータルを向かい合わせにしたみたいな感じだよね」
魔理沙はときどきこういうわからないことを言う．
「この t をもとに，新しい集合 $\{t\}$ を考えると，当然 $\{t\} \neq \varnothing$ となるわね」
「t が入ってるもんね」
「よって基礎の公理から次が言えるわ」：

$$\exists y \in \{t\}\ \ \forall t' \in \{t\}\ \ (t' \notin y).$$

「この y も t' も t にしかとりようがないね」
「ええそうね．だから結局次が得られる」：

$$t \notin t.$$

「あれ，さっきは $t \in t$ だったのにその否定が出てきた．ってことは矛盾！」
「というわけで，どんな集合 t についても，t と $t \cup \{t\}$ は一致しないわね」
「——いまは一段階で t と $t \cup \{t\}$ が一致することはない，ということを示したけど，t から $t \cup \{t\}$ を作る処理を何段階か繰り返したらもとの t に戻る可能性は残ってることになる」アリスがコメントした．
「その可能性も，基礎の公理を使って潰せるわよ．あとでやってみるといいわ．さて，次に冪集合公理を紹介するわ」

冪集合公理

$$\forall x\ \ \exists y\ \ \forall t\ \ \ (t \in y \iff t \subseteq x).$$

「この公理によって集合 x に対して存在することが保証される y を《x の冪集合》と呼んで，$\mathscr{P}(x)$ と書くわ．さて，次は置換公理図式を紹介するわね」

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

$$\forall t_1 \ldots \forall t_n \ \forall a \Big(\big(\forall x \in a \ \exists! y \ A(x,y,a,t_1,\ldots,t_n) \big) \\ \Longrightarrow \exists b \ \forall y \ \ (y \in b \iff \exists x \in a \ A(x,y,a,t_1,\ldots,t_n)) \Big).$$

「ここに出てくる《∃!》ってのは？」
「ごめんなさい，説明してなかったわね．これは《ただひとつ存在する》ことを表すために使われる記号で，次のように定義されるわ」:

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

「なるほど，《存在すること》と《存在するとしたら一致すること》を言ってるんだね」
「そうね．両方合わせると《ただひとつ存在する》ことになるわね」
「ところで，この置換公理図式って，なんかさっきの分出公理図式に雰囲気が似てるね」
「実際，分出公理図式を一般化したものがこの置換公理図式だとも言えるわ．とても強力な公理だけど，普通に数学をやっているときは分出公理図式で間に合うことが多いんだけどね」
「なるほど」
「置換公理図式で存在を保証された集合 b を

$$\big\{ y \,\big|\, \exists x \in a \ A(x,y,a,t_1,\ldots,t_n) \big\}$$

と表記するわ」
「この書き方だとわかりやすいね」
「さて，ZFC 集合論の公理——それと公理図式——の最後は選択公理だけど，ここでは選択公理を関数に関する公理として述べたいから関数の用語を少し補うわね．まず，集合 f が《関数であること》は上で定義したように FUNC(f) として定義されるのだったわね．関数や関係を扱うときに，定義域や値域は重要なので，それらを取り出す操作にふさわしい名前をつけておきましょう」:

任意の集合 R に対して，分出公理図式を用いて次のように定義する：

$$\mathrm{dom}(R) := \Big\{ x \in \bigcup\bigcup R \,\Big|\, \exists y \ \big(\langle x,y \rangle \in R \big) \Big\}.$$
$$\mathrm{ran}(R) := \Big\{ y \in \bigcup\bigcup R \,\Big|\, \exists x \ \big(\langle x,y \rangle \in R \big) \Big\}.$$

「この R のところに RELATION(f) であるような f を入れると定義域と値域になるの？」
「やってみるといいわ」
「うーん……集合 R に対して和集合をとる操作を二回やってるし，なんか複雑だな」
「$\bigcup\bigcup R$ が出てくる理由は順序対の定義が原因ね．集合 f が関係であることは，f が順序対の集まりであることとして RELATION(f) で定義されていたでしょ．そこで，いま $\langle x,y \rangle \in f$ としてやると，$\langle x,y \rangle = \{\{x\},\{x,y\}\}$ だったから

$$\{x,y\} \in \bigcup f$$

となるわね」
「なるほど，だから

$$x, y \in \bigcup\bigcup f$$

となるわけか」
「次に，関数についてよく使われる用語を定義しておくわ」:

$$f\colon A \to B \quad \equiv \quad \mathrm{FUNC}(f) \wedge (\mathrm{dom}(f) = A) \wedge (\mathrm{ran}(f) \subseteq B).$$
$$f\colon A \xrightarrow{\text{onto}} B \quad \equiv \quad (f\colon A \to B) \wedge (\mathrm{ran}(f) = B).$$
$$f\colon A \xrightarrow{\text{1-1}} B \quad \equiv \quad (f\colon A \to B) \wedge \big(\forall y \in \mathrm{ran}(f)\ \ \exists! x\ \ \langle x,y\rangle \in f\big).$$

「なるほどね」
「最後に，集合 f について $\mathrm{FUNC}(f)$ が成り立っているとき，任意の $x \in \mathrm{dom}(f)$ に対して

$$\langle x,y\rangle \in f$$

となる $y \in \mathrm{ran}(f)$ はただ一つ存在するのだったわね．その y を $f(y)$ と書き，《関数 f の x における値》と呼ぶことにしましょう」
「ここまでくると《ふつうに使う集合の言葉》がかなりそろった感じだね」
「さて，関数についての言葉が用意されたので，選択公理を関数の言葉で次のように紹介できるわ」：

選択公理

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

「この公理で存在を保証されている関数 f は集合 a に対する**選択関数 (choice function)** と呼ばれる事があるわ」
「関数 f は、集合 a のそれぞれの要素 x から一個の要素を取ってくる感じだね」

3.4.3 ZFC の自然数と形式的有限性

「さて，以上で ZFC の公理と公理図式をすべて紹介したことになるけど，最後に ZFC における自然数や形式的な有限性について話しておくわ——ZFC において自然数や《有限集合》がどのように定義されているかを理解していないと IST の議論についていけなくなるでしょうから」
「そんなもんなのか」
「さっきアリスが言ったように，ZFC 集合論では集合だけを扱うから，自然数も集合として定義する必要があるわ」
「——つまり《自然数》を《集合》にエンコードする」アリスが呟いた．
「《0 という自然数》を集合にエンコードしたもの $\boxed{0}$ は次のように定義されるわ：

$$\boxed{0} := \varnothing.$$

一般に，《n という自然数》を集合にエンコードしたものを x とするとき，自然数 $n+1$ を集合にエンコードしたもの x' を次のように定義しましょう」：

$$x' := x \cup \{x\}.$$

「この形は無限公理のところで出てきたね」
「集合 x に $x \cup \{x\}$ を対応させる手続きは，いまの文脈では後続数 (successor) に対応するから

$$\mathrm{SUCC}(x) \equiv x \cup \{x\}$$

としましょう．すると，無限公理は次のように書き直せるわ」：

$$\exists X_0\ \ \big(\boxed{0} \in x\ \ \wedge\ \ \forall t \in X_0\ (\mathrm{SUCC}(t) \in X_0)\big).$$

「なるほど，じゃあこの集合 X_0 が，エンコードされた自然数全体の集合なんだね」
「いいえ，そんなに単純じゃないわ．この集合 X_0 は，確かにエンコードされた自然数をすべて含

んでいる——でも《それ以外》の要素も持っているかもしれないのよ」
「それ以外？」
「たとえば，これは適当な思いつきだけど $\{\{\{\varnothing\}\}\}$ とか」
「なるほど，確かにそれは SUCC を使って $\varnothing$ から作れそうじゃないね．えーと，じゃあ分出公理図式を使ってこうしたらどうかな：

$$Y := \{x \in X_0 \mid x \text{ は} \varnothing \text{に有限回 SUCC を適用したもの}\}.$$

こうしたらこの集合 Y は《自然数全体の集合》にならない？」
「——《x は $\varnothing$ に有限回 SUCC を適用したもの》という部分が言語 $\mathscr{L}_{\mathrm{ZFC}}$ で表現できていなければ分出公理図式は使えない」アリスがコメントした．
「いまアリスが言った通りよ．言語 $\mathscr{L}_{\mathrm{ZFC}}$ の論理式で述べられる条件しか使えないからこれは正しい定義ではないわ」
「うーん……そうか．じゃあ，ZFC ではどうやって自然数の集合を定義するの？」
「この話を丁寧にすると長くなるから，概略だけね．あとで自分で勉強して欲しいのだけど《順序数》の理論というのがあるのよ．無限公理を使うと，《極限順序数》というものが存在することが言えるの．そして，自然数の集合 $\mathbb{N}$ は《最小の極限順序数》として定義されるわ」
「なるほど，最小ってことならエンコードされた自然数だけがぴったり入った集合なんだね」
「集合 $\mathbb{N}$ が《最小の極限順序数》として定義されていているという以上の情報は得られないわ．わたしたちは，集合 $\mathbb{N}$ と《わたちたちの自然数》がエンコーディングを通して一致すると信じて生活している．でも，もし《信じる》というのが気に入らないならば《証明》しなければならないわね．でも一体何を証明すればいいのかしら」
「なんか難しくて面倒な話だね．」
「——集合 $\mathbb{N}$ が《わたしたちの自然数》をエンコードしたものだけで埋まってると思って暮らしてもいい．でも集合 $\mathbb{N}$ がバンダースナッチのように危険で油断のならないやつだという可能性は消えない」アリスが呟いた．
《バンダースナッチ》はルイス・キャロルの『鏡の国のアリス』の中に出てくる詩で言及される架空の生物だ．正体は不明だがどうやら恐ろしい怪物という設定らしい．
「それはいくらなんでも脅しすぎじゃないかしら．さて，自然数 1,2,3,4,5,6,7 を集合にエンコードしたものを順に $\boxed{1}$,$\boxed{2}$,$\boxed{3}$,$\boxed{4}$,$\boxed{5}$,$\boxed{6}$,$\boxed{7}$ とすると，

$$\begin{aligned}
\boxed{1} &= \boxed{0} \cup \{\boxed{0}\},\\
\boxed{2} &= \boxed{1} \cup \{\boxed{1}\},\\
\boxed{3} &= \boxed{2} \cup \{\boxed{2}\},\\
\boxed{4} &= \boxed{3} \cup \{\boxed{3}\},\\
\boxed{5} &= \boxed{4} \cup \{\boxed{4}\},\\
\boxed{6} &= \boxed{5} \cup \{\boxed{5}\},\\
\boxed{7} &= \boxed{6} \cup \{\boxed{6}\}
\end{aligned}$$

となるわね」
「うん」
「あとに出てくる数は一個前の数を引用しているから，これを使えば展開できるわね」
「なるほど，じゃあ $\boxed{7}$ でやってみるか：

$$\begin{aligned}
\boxed{7} &= \boxed{6}\cup\{\boxed{6}\}\\
&= (\boxed{5}\cup\{\boxed{5}\})\cup\{\boxed{6}\} = \boxed{5}\cup\{\boxed{5},\boxed{6}\}\\
&= (\boxed{4}\cup\{\boxed{4}\})\cup\{\boxed{5},\boxed{6}\} = \boxed{4}\cup\{\boxed{4},\boxed{5},\boxed{6}\}\\
&= (\boxed{3}\cup\{\boxed{3}\})\cup\{\boxed{4},\boxed{5},\boxed{6}\} = \boxed{3}\cup\{\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}\\
&= (\boxed{2}\cup\{\boxed{2}\})\cup\{\boxed{3},\boxed{4},\boxed{5},\boxed{6}\} = \boxed{2}\cup\{\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}\\
&= (\boxed{1}\cup\{\boxed{1}\})\cup\{\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\} = \boxed{1}\cup\{\boxed{1},\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}\\
&= (\boxed{0}\cup\{\boxed{0}\})\cup\{\boxed{1},\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\} = \boxed{0}\cup\{\boxed{0},\boxed{1},\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}
\end{aligned}$$

$\boxed{0}$ は $\varnothing$ だから

$$= \{\boxed{0},\boxed{1},\boxed{2},\boxed{3},\boxed{4},\boxed{5},\boxed{6}\}.$$

ん？ $\boxed{7}$ という集合が 7 個の要素を持つのか！」

「というわけで，$\mathbb{N}$ の要素を使って《集合 x の有限性》$\mathrm{Fin}(x)$ の定義を次のように与えられるわ」:

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

「《わたしたちの有限性》を，集合 $\mathbb{N}$ を通して形式化したんだね」

「集合の有限性は IST で重要な役割を果たすわ．そして IST における有限集合のなかには大変奇妙なものもあるの．集合 $\mathbb{N}$ や《集合の有限性》の ZFC における定義の理解があやふやだと IST の理解もあやふやになってしまうわ」

「そうなのかー．でも《自然数の集合》だとか《有限集合》がそんなに難しく定義されるものだなんて全く知らなかったな」

「少し難しさを強調しすぎたかもしれないわね．《自然数の集合》だとか《有限集合》には少し油断のならないところがあるけど，形式に沿って議論するように気をつけていれば殆どの場合問題にはならないわよ」

「なるほど，そんなもんか」

「というわけで，これからは《わたしたちの自然数》の集合論へのエンコーディングなんかは必要なときを除いて意識しないことにして，$\boxed{0},\boxed{1},\boxed{2},\ldots$ などと書く代わりに $0,1,2,\ldots$ などと書くようにしましょう」

「わかった」

「ZFC 集合論の話の締めくくりとして ZFC における数学的帰納法を述べておくわ」:

$$\forall K\ \left(\Big(K\subseteq\mathbb{N}\wedge 0\in K\wedge\forall n\in\mathbb{N}\ \big(n\in K\implies \mathrm{SUCC}(n)\in K\big)\Big)\implies K=\mathbb{N}\right)$$

「これはどうやって証明するの？」

「公理的集合論の《順序数》のあたりを勉強すれば証明できるわ」

「なるほど，じゃあこれは持ち帰り問題にしとくよ」

「さて，これでようやく IST に入れるのだけど，《前編》はここで終わりよ」

「ええ，そんな」

「ネルソンの IST 論文を読む上での一番引っかかりそうなところはこれで説明したわ」

（前編終了，後編に続く）

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.

3.4.4 参考文献

[1] 結城 浩「数学ガール」シリーズ ソフトバンク・クリエイティブ
本記事の直接のネタ元というわけではありませんが，主人公「僕」が周囲の人間を巻き込みあるいは巻き込まれながら，対話を交えた試行錯誤のなかで数学を学んでいくというスタイルには多大な影響を受けています.

[2] E. Nelson, “Internal set theory: A new approach to nonstandard analysis”, Bull. Amer. Math. Soc. **83**(6), pp.11651198.
IST の創始者ネルソンによる論文です．非常に密度が高く，読者に要求している前提知識の質も比較的高いと思います.

[3] A. Robert, “Analyse non standard”, Presses polytechniques romandes (1985).
この本は，ネルソンの IST を解説した本になっています．この本の英訳 “Nonstandard Analysis” が Dover から出ていました（2016 年現在絶版）.

[4] F. Diener & M. Diener (Eds.), “Nonstandard Analysis in Practice”, Springer (1991).
ネルソンの IST の様々な応用が述べられています．外集合の使用に積極的なのも特徴です.

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

[6] J. H. Gallier, “Logic for Computer Science (2nd ed.)”, Dover (2015).
述語論理の箇所を書くときに上記の二冊の書物を参考にしました.

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

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

[9] 広瀬健 『数学・基礎の基礎』 海鳴社 (1996).

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

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

[12] I. Hacking, “Why Does Language Matter to Philosophy?”, Cambridge (1975)
(日本語訳： イアン・ハッキング 『言語はなぜ哲学の問題になるのか』 勁草書房 (1989)).
ジョージ・バークリーの話を書くときに参考にしました.

[13] L. Euler, 『無限解析序説』（原題：“Introductio in Analysin Infinitorum”）, (1748)
(日本語訳： 高瀬正仁訳『オイラーの無限解析』， 海鳴社 (2001))
本文中で引用した訳文は高瀬訳からお借りしました.

改訂情報

重要なものから順に

- p.53 公理の形が

$$\forall a\ \exists f\colon a \to \bigcup a\quad \forall x \in a\ \left(x \neq \varnothing \implies f(x) \in x\right).$$

だったのを

$$\forall a\ \left(\varnothing \notin a\ \implies\ \exists f\colon a \to \bigcup a\ \left(\forall x \in a\ \left(f(x) \in x\right)\right)\right).$$

に修正しました.

- p.51 ”FV$(A) = \{x, y, t_1, ..., t_n\}$”を”FV$(A) = \{x, y, a, t_1, ..., t_n\}$”に修正しました．また，これに合わせて二箇所の”$A(x, y, t_1, ..., t_n)$”を”$A(x, y, a, t_1, ..., t_n)$”に修正しました.

- p.45 "$\forall A(x, y_1, ..., y_m)$"を"$\forall x\ A(x, y_1, ..., y_m)$"に修正しました.
- p.55 ZFC における数学的帰納法の箇所で"$n \in K \implies \mathrm{SUCC}(x) \in K$"となっていた箇所を"$n \in K \implies \mathrm{SUCC}(n) \in K$"と修正しました.
- p.43 $\mathsf{Form}_{\mathscr{L}}$ の定義が二回書かれていたのを訂正しました. また, $\langle \boxed{\Leftrightarrow}, A, B \rangle$ の形の論理式の定義の欠落を補いました. この修正に合わせて, これに続く幾つかの箇所で修正を行いました.
- p.49 公理の名称が「対の公理」となっていたのを「非順序対の公理」に訂正しました.
- —— 数カ所で会話の流れや語調を整えました.

会員名簿じゃないカ?

@ark_golgo （1 章）

博士課程は 4 年目に突入してしまいましたが、「せめて AlphaGo のことを皆に伝えてからでないと、死んでも死にきれない!」ということで今回執筆させて頂きました。

@master_q (2 章)

大好きだったエロゲの続編が C90 に間に合わないと知って、生きる気力が減少気味です。

@dif_engine (3 章)

IST の原論文を読むのに必要になりそうな ZFC の基礎事項を書いていたら時間切れになってしまい残念です.（後編にご期待ください）


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


「簡約（かんやく）！？ ラムダ カ ムスメ 9」

2016 年 8 月 14 日 コミックマーケット 90 初版発行

原作：安部真弘「侵略！ イカ娘」より

発行：参照透明な海を守る会

http://www.paraiso-lang.org/ikmsm/

ikmsm-authors@googlegroups.com

著者：@ark_golgo, @master_q, @dif_engine

表紙：@dif_engine

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


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

参照透明な海を守る会