競技プログラミングの問題を自動的に解きたい

Kimiyuki Onaka

2021 年 9 月 27 日

0　概要

この PDF では「競技プログラミングの問題を自動的に解けるか？」という問いについて考える。あるいは「競技プログラミングのソルバを作ることはできるか？」と言い換えてもよいだろう。

1 章ではこのまだ曖昧な問いをより具体的な問いへと整理する。競技プログラミングの問題を自動的に解くソルバにも様々な種類のものがあることを説明し、今回我々が作成を目指すソルバがそれらの中のどの種類のものかを明確にする。我々が作成を目指すのは「形式的に表現された問題を受け取り、それに対する解答を出力する」という種類のものであり、その中でも特に「形式的に表現された問題」として「愚直解のソースコード」を用いるものである。そのようなソルバはつまり「漸近的計算量のレベルの最適化を伴うトランスパイラ」である。よって「競技プログラミングのソルバを作ることはできるか？」という問いは「漸近的計算量のレベルの最適化を伴うトランスパイラは作れるか？」という具体的な問いとなる。また、より実用的には、完全なソースコードでなく小さなコード片を入力とする「漸近的計算量のレベルの最適化を行う IDE プラグイン」としても利用できると扱いやすいであろう。

2 章では具体化された問いに対する暫定の答えを与える。つまり、最適化を伴うトランスパイラとしての競技プログラミングのソルバの実装方針を説明し、またそのような方針で実装されたソルバでいくらかの問題が解けることを見る。実装方針は機械的な式木の書き換えが中心となる。たとえば、愚直に何度も区間和を取っている部分を検出すればそれを累積和で置き換えたり、愚直な 2 重ループで畳み込みをしている部分を検出すればそれを FFT を用いた畳み込みで置き換えたりする。このような書き換えは “rewrite rule” と呼ばれる規則として表現され、よってソルバの中心部分は「rewrite rules からなる集合」として実装される。

1　競技プログラミングの問題を自動的に解くとはどのようなことか

1.1　そもそもなぜ競技プログラミングの問題を自動で解きたいのか

まず先に、「競技プログラミングの問題を自動的に解けるか？」という問いの意義について述べておこう。なぜ競技プログラミングの問題を自動で解きたいのだろうか？ これは主には以下の 3 種類であろう。

(1.) 競技プログラミングでの実用:「コンテスト中の作業を自動化してレートに繋げたい」「ソルバの実行ログが学習用途に使えるかもしれない」など
(2.) 工学的な興味:「どの程度の問題までなら自動で解けてしまうのか知りたい」など
(3.) 哲学的な動機:「問題が手動でなら解けても自動では解けないなら、本当に解けたとは言えないかもし

れない」「人間がどのようにして問題を解いているのかについて理解を深めたい」

(1.) 競技プログラミングでの実用は分かりやすい。簡単な問題や難しい問題の部分問題を手動でなく自動で高速に解いてしまえるなら、難しい問題の難しい部分に時間を使うことができてコンテストで有利である。もし自分では解けない難しい問題がソルバによって解けるならなおさらである。レートに影響がなくとも、簡単な問題を自動化してしまえるなら便利で楽であろう。

また、教育や学習用途でソルバを使うことができるかもしれない。ソルバが問題を解く過程を詳細に分かりやすく出力すれば、初心者はそれを読んで問題の解き方を学ぶことができるだろう。

(2.) の工学的な興味も分かりやすい。単純に、どの程度の問題までなら自動で解けてしまうのか (そしてどのような問題は自動では解きにくいか) を知れれば面白い。

(3.) は哲学的な動機である。我々は競技プログラミングの問題を手動で解くことができる。それはいったいなぜだろうか？ 問題を解く方法を具体的なアルゴリズムとして説明できるだろうか？ それができないのに「競技プログラミングの問題を解くことができる」と言い切ってしまってよいのだろうか？

たとえば、普通に競技プログラミングをしている人が「ある問題が頭の中では解けました。しかし実装方法については検討も付きません」と発言しているとしよう。ある程度競技プログラミングをやったことのある人であれば、そのような発言に対しては「解けたつもりになっているのが間違いで、そんな状況のことを解けたとは言わない」と思うだろう。つまり、みな「問題が解けたならば実装できるはずだ」と信じているのである。これと似た別の状況を考えてみよう。「ある問題が解けて実装もできた。しかしなぜ私にこれが解けたのかはよく分からない」という状況である。このような状況もやはり「その問題に対する理解が足りていない」とみなされることが多いだろう。理解が足りていないままであれば、もしその問題の類題が出題されたときそれを解けないかもしれない。「問題の解法を正しく理解できたなら、その解法に至る道筋を説明できるはずだ」と信じている人は多いはずである。これらの観察からは次のことが疑われる。つまり「競技プログラミングを十分に理解しているならば、そのソルバも実装できるはずである」。ソルバを実装しようとしてみることは競技プログラミングへの理解に貢献するだろう。

これはさらには、より一般の人間の思考についての問い、つまり「考えるとはどのようなことか」を考えることにも繋がってくる。たとえば、機械的な規則の適用をくり返すことで実装されたソルバがもし人間に匹敵するような成功を収めたとすれば、そこから人間の思考も結局は規則の機械的な適用を素早く行っているだけではないかと疑うことができる。あるいは逆にそのようなソルバがまったく失敗してしまうなら、人間の思考には機械的な規則の適用としては捉えきれない暗黙的な部分があることが疑われる。

1.2 競技プログラミングの問題を人間はどのように解いているのか

「競技プログラミングの問題を解く」と言っても、その範囲は意外に広い。まずこれを整理するところから初めよう。競プロの問題のページが開かれた直後の状態 (たとえば `https://atcoder.jp/contests/abc184/tasks/abc184_c` というリンクをクリックした直後のような状態) から初めて、緑色の「AC」という表示を確認するまでの過程において、具体的にはどのようなステップがあるだろうか？ もちろんこの過程には様々なものが含まれており (例:「問題文中に埋め込まれている図を解釈する」「問題ページをスクロールする」「青い “提出” と書かれたボタンをクリックする」など) いくらでも細かく説明できてしまう。しかし、重要でないものを無視して抽象的な形で整理すると、通常の過程は次の 3 ステップにモデル化できるだろう。

(1.) **読解** 自然言語で与えられた問題を解釈し、数学的な問題として整理する

(2.) **考察** 数学的な問題に対し、数学的な解法を考案する
(3.) **実装** 数学的な解法を、計算機上の具体的な実装として記述する

順番に見ていこう。(1.) は (人間にとっては) 自明なのであまりかえりみられることのないステップである。このステップでは、日本語や英語で書かれた問題を読み、数式などで表現された問題として整理し理解する。たとえば「熊の Limak が人間の高橋くんに合いに AtCoder 王国を訪れている。AtCoder 王国は 1 から N までの番号の付いた島と、それらの間にかかる M 本の橋からなる。今 Limak は s 番目の島にいて、高橋くんは t 番目の島にいる。そして……」のような問題文を読んで理解し、不要な情報を捨象して「N 頂点無向グラフ $G = (V, E)$ と $s, t \in V$ が与えられる。……」のような問題であることを確認する。

(2.) は競技プログラミングにおいて最も難しくかつ面白いとされるステップである。このステップでは、(1.) で理解した問題に対し、実行時間などの制限を満たす解法を考案する。たとえば、考察の結論として「まず重み付きグラフ $G = (V, E, w)$ とすべての頂点 $x \in V$ に対し $s - x$ 最短路を s からの Dijkstra 法で求める。次に重み付きグラフ $G' = (V, E, w')$ とすべての頂点 $x \in V$ に対し $t - x$ 最短路を t からの Dijkstra 法で求め、そして……」などのアルゴリズムが得られる。

(3.) はジャッジサーバによる自動採点のために必要なステップである。このステップでは、(2.) で得たがまだ頭の中にある抽象的なアルゴリズムを、計算機上の具体的なコードへ翻訳する。たとえば「$s - t$ 最短路を Dijkstra 法で求めればよい」というときには、C++ なら `priority_queue<pair<int64_t, int>> que; que.emplace(0, s); while (not que.empty()) …` のようなコードが書かれるだろう。

そして、今回の話題は (2.) の「考察」のステップである。このステップは競技プログラミングという行為において中心となるステップであり、「問題を解く」と言うときも主にこのステップに注目している。つまり「競技プログラミングの問題を自動で解きたい」とは「考察を自動で行いたい」ということである。

1.3 競技プログラミングのソルバは形式化を人間に任せるとよいだろう

競技プログラミングの問題を解くことを「読解」「考察」「実装」という 3 ステップに分けた上で主に「考察」を行いたいのだと言っても、残りのふたつのステップは容易に切り離せるものではない。問題は何らかの形で入力されねばならないし、解法は何らかの形で出力されねばならない。「読解」「実装」という残りのステップについても扱える必要がある。

機械にとって最も面倒なのは「読解」だろう。自然言語の理解という高度な処理が必要になる。競技プログラミングの問題文を以降のステップに支障がでないように読み取るには、表面的に単語を追うだけではおそらく足りず、文章全体の意図を正しく把握する必要があるだろう。競技プログラミングの問題文以外は理解できなくてよいのだとしても、これはかなり困難である。そしてこの「読解」は今回やりたいことではない。ソルバの一部として取り扱うのは諦めて、人間 (あるいは GitHub Copilot のような言語モデル) に任せてしまうべきだろう。

「実装」もそのままの形では機械では扱えない。「数学的な解法を、計算機上の具体的な実装として記述する」と言うときの「数学的な解法」などは、(通常の人間による競技プログラミングにおいては) 人間の頭の中のみ存在するものだからである。人間の頭の中にしかないものは機械には扱えない。「数学的な解法」でなくて「数学的な問題」の場合も同様である。

一方で、機械で扱える形に形式化したものが入力であれば「実装」も機械に可能である。このことはコンパ

イラを思い出せば分かる。F# や Haskell のような高級言語のソースコードを仮想機械の中間言語や実際の機械語に変換する操作はまさに「実装」であろう。

よって、ソルバを用いて競技プログラミングの問題を解くとすると、その過程は以下のような 4 ステップに整理されるだろう。この形であれば可能性はある。

(1.) **読解** (人間が) 自然言語で与えられた問題を解釈し、数学的な問題として整理する
(2.) **翻訳** (人間が) 数学的な問題を、機械上に形式化された問題として入力する
(3.) **考察'** (ソルバが) 記述された形式化された問題に対し、機械的に解法を計算する
(4.) **実装'** (ソルバが) 計算された解法を、具体的な実装として出力する

(1.) と (2.) は合わせて「形式化」をしている。これらのステップはどちらも人間にとっては簡単である。「読解」のステップが簡単であるのは認めてよいだろう。「翻訳」のステップも人間にとっては簡単だとしてよい。ただ読んだままを書けばよいためである。まったく理解できない難しい数式であってもそれを LaTeX で打ち込むだけならそれほど難しくないことがたいていだろうが、これと同じようなものである。

競技プログラミングのソルバが (1.) と (2.) のステップによる「形式化」を人間に任せてしまったとして、そのような条件で残りのステップだけを処理した場合も「競技プログラミングの問題を自動で解けた」と言ってよいだろうか？ これは「自動で解けた」と言ってしまってよいだろう。なぜなら、「形式化」は (機械にとっては難しいとはいえ) 人間にとってはたいてい簡単であるし、通常は重要視されるステップではないからである。困難でも重要でもないステップを省略しても、困難で重要な (3.) の「考察'」のステップをうまく扱うのなら「問題を自動で解く」と言ってよいだろう。様々なことを自動でしてくれるソフトウェアであってもデータの打ち込みは人間の担当であることは多く、今回もそうだったというだけである。

1.4 競技プログラミングのソルバはトランスパイラの形で実装するのがよいだろう

ソルバの入力は「形式化された問題」だろうと言ったが、では「形式化された問題」というのは具体的に何にするとよいだろうか？ (入力となる変数, 出力となる変数, それらが満たすべき制約) という 3 つ組だろうか？ 計算可能な関数のグラフだと考えるべきだろうか？ これはおそらく何でもよい。問題を表現できてさえいれば、入力フォーマットの選択によって問題が解けたり解けなかったりするといったことはほぼないだろう。好きなものを使えばよい。

しかしソルバへの入力形式はソルバ自体の実装のしやすさを大きく左右する。どのような入力にするのはもっとも楽だろうか？ 何度かプロトタイプを実装した経験から、私は「実行可能なプログラム」を「形式化された問題」の代わりに使うのがよいと考えている。「実行可能なプログラム」をソルバの入力とすれば、競技プログラミングの過程は以下の 3 ステップに具体化される。

(1.) **読解** (人間が) 自然言語で与えられた問題を解釈し、数学的な問題として整理する
(2.) **実装"** (人間が) 数学的な問題を、計算機上にその愚直解として実装する
(3.) **考察"** (ソルバが) 記述された実装に対し、計算量的な最適化を行う

このとき (3.) の「考察"」のステップは、入力も出力も共にプログラミング言語である。入力されたコードと同じ計算をするより高速なコードを出力する。これは「最適化」である。つまり「実行可能なプログラム」をソルバの入力とすれば、「競技プログラミングの問題を自動で解くソルバ」は「最適化を伴なうトランスパイラ」となる。

「競技プログラミングの問題を自動で解くソルバ」を「最適化を伴なうトランスパイラ」と言い換えられた。これは大きな進歩である。「なにやらよく分からないすごそうなもの」ではなく「いつもの」を書けばよいことになった。「人工知能」と比べれば「コンパイラ」はたいへんに開発しやすい。既存の技術や資料をたくさん利用できる。また、入力と出力のギャップが小さいことも利点である。入力をそのまま出力するだけのプログラムでさえ、いまや「自明なソルバ」だと主張できる。

なお、「形式化された問題」として「実行可能なプログラム」使うことは明らかなことではない。答えを先に聞くと自然に聞こえるかもしれない。しかし思い出してほしいのは、「問題」と「プログラム」はまったく異なるものだということである。プログラムは実行できるが、問題は実行できない。プログラムは漸近的計算量を持つが、問題は漸近的計算量を持たない。少なくとも私はいくつかのプロトタイプを作るまでこのギャップを乗り越えられなかった。

なお、トランスパイラ以外の形のソルバの可能性としては、たとえば「対話的なソルバ」「過去問データベースと検索エンジンからなるソルバ」などが考えられる。

1.5 競技プログラミングのすべての問題が自動で解けるわけではない

言うまでもないことだろうが、競技プログラミングのすべての問題が自動で解けるわけではないことを注意しておく。期待しすぎるべきではない。機械的な解法と相性の良い問題は解けるだろうが、相性の悪い問題は解けない。少なくともしばらくの間は、再翻訳がおもしろおかしいものだった時代の機械翻訳と同程度の有用性がせいぜいのはずである。

特にトランスパイラとしてのソルバについて言えば、相性の悪い問題にはたとえば以下のようなものがある。

(1.) 愚直解の実装が難しい問題
(2.)「直感」や「ひらめき」を必要とする問題
(3.) 問題そのものが複雑な問題

(1.) は、たとえば幾何の問題である。ソルバの担当範囲以外の部分が難しい。そのような場合は人間が頑張るしかない。考察要素がある問題であってもその考察要素が「愚直に実装するとたいへんだが、うまく工夫すると実装量を減らすことができる」という形であれば、やはりソルバは無力である。

(2.) は、たとえば貪欲法が想定解の問題である。ソルバの担当範囲が難しい。実装にもよるだろうが、基本的にソルバは 1 歩ずつ考察を詰めていけば解けるような問題しか解けないだろう。「証明はできないけどなんとなく正しそう」のような判断は機械とは相性が悪そうである。「突然ですが、ここで $x = f(y, g(y, z))$ とおきます。すると……」のような発見的解法も機械とは相性が悪そうである。手法によっては扱えるかもしれないが、少なくとも、先程の節で述べたような形の機械的解法で解くことはできない。

(3.) は、たとえばゲームの問題がそうだろう。交互ゲームで「先手が必勝である」ことを表現しようと思うだけで、量化子の繰り返しや再帰関数のような扱いにくいものが出てきてしまう。きれいな式で表現できないような問題と機械の相性は悪いだろう。

1.6　ソルバの実践での利用について

競技プログラミングのソルバの、それをコンテスト中に利用するユーザから見たときの様子を検討しておこう。基本的には現在のオンライン整数列大辞典 (OEIS) や Wolfram|Alpha とそう変わらない立ち位置にしかならない。

まず明らかに言えることは、一般にソルバは銀の弾丸にはならないということである。もしどんなに優秀なソルバが開発されたとしても、そのようなソルバで解けてしまう問題は出題前に弾かれるだけである。ソルバが利用できるとしても、それは実際の問題の部分問題に対してのみであろう。また、ソルバですべての問題が解けるわけではないので、ユーザは「問題が自動的に解けるのはどのような場合であるか」を大まかに判断できる必要がある。機械的に解けうる問題であっても入力の与え方次第では解けないことも多いので、この判断を行なえることはソルバの利用のために必要である。しかしそのような判断ができるには結局はユーザ自身の能力が要求される。

また、「競技プログラミングの問題のソルバ」という形であれば入力は問題そのものでしかありえないのだが、コンテストでの実用を考えたときにはそれ以外の可能性も現れてくる。最終的な解法コードのうちでアルゴリズムに本質的に関わる部分はごく一部のみであるので、そのようなコード断片のみを入力とすることもできる。すると、そのようなコード断片のみを扱うものはエディタのリファクタリング支援機能の延長に位置するものになる。それは「IDE のプラグイン」として実現される。ユーザがコード断片を指定すると、その部分の漸近的計算量を落とすなどの書き換えを行ってくれるようなものである。入出力がソースコードの全体でなくコード断片という小さいものになれば、結果の解釈や再利用もより柔軟に行えて便利であろう。

2　最適化を行うトランスパイラはどのように実装されるか

2.1　手作業による機械的な変形で解くことができる問題がある

ソルバのことはいったん忘れて、手作業による機械的な変形で解くことができる問題を AtCoder Beginner Contest 100 の 4 問目 Patisserie ABC (`https://atcoder.jp/contests/abc100/tasks/abc100_d`) を例に見てみよう。これは整理すると以下のような問題である。そこまで難しくはないので、自分がどうこの問題を解いているのかを観察しながら一度自分で解いてみて、それから続きを読み進めてほしい。

問題 2.1. *[Patisserie ABC から引用]*

> 高橋君はプロのパティシエになり, *AtCoder Beginner Contest 100* を記念して,「*ABC* 洋菓子店」というお店を開いた.
>
> *ABC* 洋菓子店では, N 種類のケーキを売っている. 各種類のケーキには「綺麗さ」「おいしさ」「人気度」の 3 つの値を持ち, i 種類目のケーキの綺麗さは x_i, おいしさは y_i, 人気度は z_i である. これらの値は 0 以下である可能性もある.
>
> りんごさんは, *ABC* 洋菓子店で M 個のケーキを食べることにした. 彼は次のように, 食べるケーキの組み合わせを選ぶことにした.
>
> - 同じ種類のケーキを 2 個以上食べない.
> - 上の条件を満たしつつ, *(*綺麗さの合計の絶対値*) + (*おいしさの合計の絶対値*) + (*人気度の合計の絶対値*)* が最大になるように選ぶ.

このとき, りんごさんが選ぶケーキの *(*綺麗さの合計の絶対値*) + (*おいしさの合計の絶対値*) + (*人気度の合計の絶対値*)* の最大値を求めなさい.

◇

形式的に書けば以下のようになる。

問題 2.2. *[*形式化された問題 *2.1]* 長さ N の整数列 $x = (x_0, x_1, \ldots, x_{N-1})$, $y = (y_0, y_1, \ldots, y_{N-1})$, $z = (z_0, z_1, \ldots, z_{N-1})$ と自然数 $M \leq N$ が与えられる。集合 $X \subseteq N = \{0, 1, \ldots, N-1\}$ に対し

$$f(X) = \left|\sum_{i \in X} x_i\right| + \left|\sum_{i \in X} y_i\right| + \left|\sum_{i \in X} z_i\right|$$

と定義する。このとき $\max\{\, f(X) \mid X \subseteq N \wedge |X| = M \,\}$ を求めよ。 ◇

これは機械的に解ける。このとき求める値 y は

$$y = \max\{\, |a| + b \mid \ldots \,\}$$

の形の式で定義されている。$|a| = \max\{\, a, -a \,\}$ であるので、このような式は

$$y = \max\{\, \max\{\, a, -a \,\} + b \mid \ldots \,\}$$

を経由して

$$y = \max\{\, \max\{\, a + b \mid \ldots \,\}, \max\{\, -a + b \mid \ldots \,\} \,\}$$

と変形できる。この変形を可能な限り繰り返すと、求める式は

$$y = \max \left\{ \begin{array}{l} \max\left\{ +\sum_{i \in X} x_i + \sum_{i \in X} y_i + \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i \in X} x_i + \sum_{i \in X} y_i - \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i \in X} x_i - \sum_{i \in X} y_i + \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i \in X} x_i - \sum_{i \in X} y_i - \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i \in X} x_i + \sum_{i \in X} y_i + \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i \in X} x_i + \sum_{i \in X} y_i - \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i \in X} x_i - \sum_{i \in X} y_i + \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i \in X} x_i - \sum_{i \in X} y_i - \sum_{i \in X} z_i \;\middle|\; \ldots X \ldots \right\} \end{array} \right\}$$

となる。次に絶対値記号が取れた $\max\{\, \pm\sum x_i \pm \sum y_i \pm \sum z_i \mid \ldots X \ldots \,\}$ の形の 8 本の式の計算が必要である。$x'_i = -x_i$ とおけば $-\sum x_i = \sum x'_i$ であるなど、符号の正負は同様に扱える。つまり次の式の計算だけを考えればよい。

$$\max\left\{ \sum_{i \in X} x_i + \sum_{i \in X} y_i + \sum_{i \in X} z_i \;\middle|\; X \subseteq N \wedge |X| = M \right\}$$

$\sum_{\phi(i)} f(i) + \sum_{\psi(i)} g(i)$ は $\phi(i) \leftrightarrow \psi(i)$ のとき $\sum_{\phi(i)} (f(i) + g(i))$ で置き換えられる。$w_i = x_i + y_i + z_i$ とおいて次の式の計算だけを考えればよい。

$$\max\left\{ \sum_{i \in X} w_i \;\middle|\; X \subseteq N \wedge |X| = M \right\}$$

この形の式「N 個のものから $M \leq N$ 個選んだときの重み総和の最大値」は十分典型的でかつこれ以上分解できそうにないものであり、これを $O(NM)$ で解く方法は既知のものとしてよいだろう[*1]。よってこれは計

[*1] $O(N \log N)$ や $O(N)$ でもよい。

算できる式であり、機械的な変形により y が $O(NM)$ で計算できることが分かった。N, M の上限と計算時間制限が与えられていれば、これが時間内に計算できることも機械的に判定できる。よってこの問題は機械的に解けると言える。

2.2 このような手動での機械的な変形はプログラムによっても可能である

2.1 節で見たような機械的な変形がプログラムによっても可能であることを、引き続き Patisserie ABC を例に見ていこう。「このようにして機械的に解ける」と主張した上の説明を分析して裏に隠されたものも明示すれば、さらにいくつかの要素に分解することができる。主には以下のような 3 段階のものとして説明できる。

(1.) 自然言語で入力された問題を形式的な形に変形する
(2.) 適用するべき変換規則を選択し適用する
(3.) 十分に簡単な形に変形された問題を実際の実装として出力する

順番に見ていこう。

まず (1.) について。これはソルバによっては扱わず、ソルバの利用者に行わせるものとするのだった。適当な高級言語を用意し、問題をそのままその言語の上に翻訳したものを入力させることになる。入力されたプログラムを解釈した結果は、たとえば図 2.2 のような構文木として理解できる。

次に (2.) について。

問題 2.1 を機械的に解く過程で出現した変形は主に以下の 5 種であった[*4]。

- $|a| = \max\{\, a, -a \,\}$ という等式に基づく変換
- $\max\{\, \max\{\, a, b \,\} + c \mid \dots \,\} \;=\; \max\{\, \max\{\, a + c \mid \dots \,\}, \max\{\, b + c \mid \dots \,\} \,\}$ という等式に基づく変換
- $-\sum_{i \in X} a_i = \sum_{i \in X} b_i$ という等式に基づく変換 (ただし $b_i = -a_i$ とおく)
- $\sum_{i \in X} a_i + \sum_{i \in X} b_i = \sum_{i \in X} c_i$ という等式に基づく変換 (ただし $c_i = a_i + b_i$ とおく)
- $\max\left\{\, \sum_{i \in X} w_i \;\middle|\; X \subseteq N \wedge |X| = M \,\right\}$ から適切な式[*5] への変換

変換規則と構文木が与えられたとき「変換が適用できるかを判定すること」および「変換を適用した結果を得ること」はプログラムによって可能である。実際、このような規則は図 2.2 に示すような木から木への組み換えとして理解できる。

ただしもちろん、これらの変換規則を人の手によって事前に組み込んでおくことが前提である。競技プログラミングをする上で有用な等式を変換の向きと共に列挙することとなる。

変換の個々の適用が可能であることだけでは十分でない。考えられる変換はこれらだけではないため「適用する変換を適切に選択できること」もまた必要だからである。この変換の選択の部分は自明ではない。いまのところ式の複雑さが減少するにように貪欲に変換すればたいていの場合で上手くいくだろうが、実際にはヒューリスティックな探索を行うことになるだろう。

いずれにせよ、トランスパイラの形のソルバはこのような「変換規則の集合」として表現されうる。たとえば実際、Haskell のコンパイラである GHC の最適化機構は “rewrite rule” と呼ばれる規則を用いて、上で見たのとほとんど同じように行われる。

[*4] $a + (b + c) = (a + b) + c$ や、$a = b$ ならば $f(a) = f(b)$ であるといった自明な等式については省略した。
[*5] たとえば列 $w' = \mathrm{reverse}(\mathrm{sort}(w))$ を計算した後に $\sum_{i<M} w'_i$ を計算するような式。

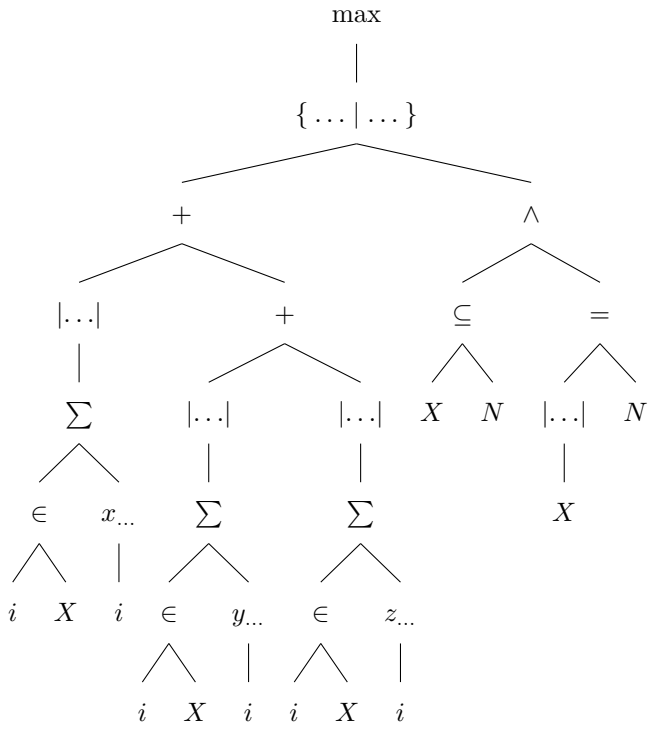

図 1　形式化された問題 2.2 を表現する構文木の一例[*3]

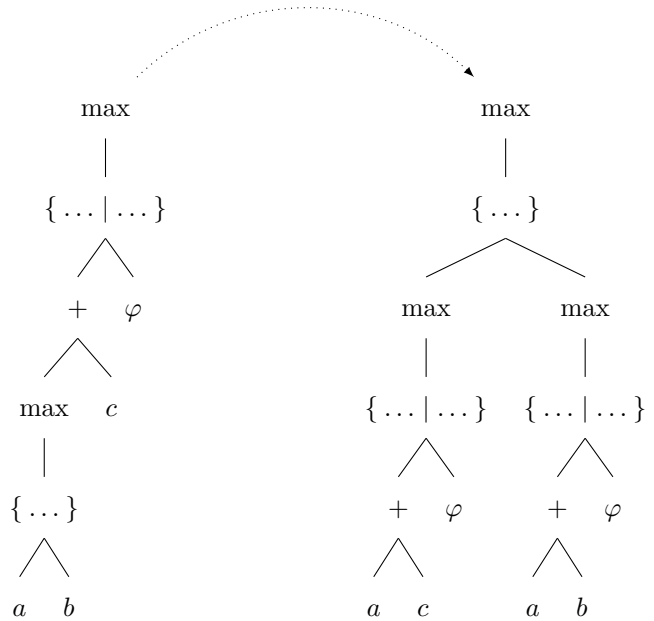

図 2　等式 $\max\{\max\{a,b\}+c\mid\dots\}=\max\{\max\{a+c\mid\dots\},\max\{b+c\mid\dots\}\}$ に基づく変換規則の、構文木の書き換えとしての表現

最後に (3.) について。そもそも操作対象である形式的な数式はそれ自体がある種のソースコードである。これを C++ のような高級言語に変換することは難しくない。

3 先行研究

一般のものであれば、以下のものが近いだろう。

- オンライン整数列大辞典 (OEIS) は整数列の巨大なデータベースである。入出力が共に単一の整数であるような問題では、愚直解を書いて得られた整数列で OEIS を検索することで解法が判明することがある。
- Wolfram|Alpha は数学関連の質問応答システムである。問題を解く過程で出てきた複雑な数式を Wolfram|Alpha に渡すことで解法が得られることがある。
- 新井紀子らのロボットは東大に入れるかプロジェクトは
- angr は CTF などで用いられるバイナリ解析フレームワークである。
- DARPA の Cyber Grand Challenge は CTF の問題を自動で
- GitHub Copilot やこれに用いられている OpenAI Codex のような言語モデルは、自然言語の問題文からその愚直解を生成することができる。Daniel Paleka (valkyrie) によるこれを実際に利用してみての分析が Codeforces の blog にある。
- borzunov/cpmoptimize は Python の関数を行列累乗を用いて書き換える decorator である。これに関してのブログ記事やこれに関しての Hacker News のスレッドには多くの関連するプロジェクトが見つかる。

競技プログラミングでの利用を意図したものに限れば、以下のような事例がある。

- 競技プログラミングの解法コードに貼り付けて利用するためのライブラリは広く開発されている。これらは最も素朴な形の「競技プログラミングの問題のソルバ」だと思うこともできるだろう。たとえば beet-aizu/library がある。
- 個別の問題のソルバとしては wata による「Σ電卓」が知られている (wata-orz/SigmaCalculator, Σ電卓 - てきとーな日記) (2009 年)。
- 実装には至っておらず構想のみであるが、一般の問題の解法の自動生成についての試みが kinaba によってなされている (SRM 531 Div2 250 - cafelier@SRM - TopCoder 部 など) (2012 年)。
- AtCoder Regular Contest のある A 問題の解法の完全自動生成が mkotha によって行われたことがある (`https://twitter.com/atcoder/status/538665089931296768`, `https://atcoder.jp/contests/arc030/submissions/286413`) (2014 年)。
- 最適化などは含まれていないが、競技プログラミングでの利用を意図して作られた言語としては laycurse による cLay がある (cLay 概要（version 20201123-1）- ゲームにっき（仮）別館（仮）) (207 年から)。
- マラソンマッチなどでの利用を意図したプログラミング言語として colun による mmlang がある (colun/mmlang) (2021 年から)。データ構造の差分の計算の自動導出機能を持つ。

一般の競技プログラミングの問題を解くソルバを目指しているものとしては、ある程度の形になっているものは見つけられていない。