解説 基 般

プログラミングするプログラム

—自動プログラム作成最前線—

森畑 明昌（東京大学大学院総合文化研究科）

プログラムがプログラムを書いてくれる?

皆さんが同僚や友達から「Excel で，ある列に姓，次の列に名前があるとき，フルネームの列を作るにはどうすればいいの?」と聞かれたらどう答えるだろうか．もちろん，文字列連結を行う Excel スクリプトを書けばよいのだが，これは言うほど簡単ではない．Excel の膨大な機能から所望のものを見つけ出すのは，良くて面倒，詳しくない人にとってはほとんど不可能だ．

専門的には，これは「自動プログラム作成（program synthesis）」の問題に属する．自動プログラム作成とは，ユーザの目的を達成するプログラムを，その目的を示唆する断片的な情報から自動的に作り出す技術である．つまり，プログラムにプログラムを書かせる技術，と言ってもよい．この分野は最近長足の進歩を遂げている．本稿ではその最先端を紹介する．

FlashFill：入出力例からの文字列処理プログラム自動作成

実は，前述の姓名の例は Microsoft の研究員である Gulwani 氏のエピソードである☆1．Gulwani 氏は自動プログラム作成の第一人者であり，この経験に触発されて FlashFill[1] という機能を Excel2013 にもたらした．すでにご存じかも知れないが，未経験の方はぜひ使ってみてほしい．

使い方は実に簡単．1，2 行，欲しい結果を入力したら，後は「フラッシュフィル」ボタンを押すだけだ．図 -1 は一例である．A 列は姓，B 列は名前であり，C 列にフルネームを入れたいとする．このとき，1 行目だけ手入力すれば，後の行はボタン 1 つで埋めてくれる．別の例を図 -2 に挙げる．今度は住所録（A 列）から郵便番号を抽出する．ただし，その後の処理の都合上，ハイフンは取り除きたいとしよう．この場合は，2 行手で書く必要があるが，3 行目以降は自動的に埋めてくれる．

FlashFill の実現

FlashFill は，自動プログラム生成技術に基づき，文字列を計算するプログラムを作り出している．ここでは図 -1 を例に簡単にそのアイディアを説明する．

目標は A 列の「佐藤」，B 列の「大輔」から「佐藤大輔」を出力するプログラムを作ることだ．これは一見とても簡単だ．文字列を連結する CONCATENATE 関数を使えばよい．しかし，実はほかにも無数の可能性がある．いくつか例を挙げよう．

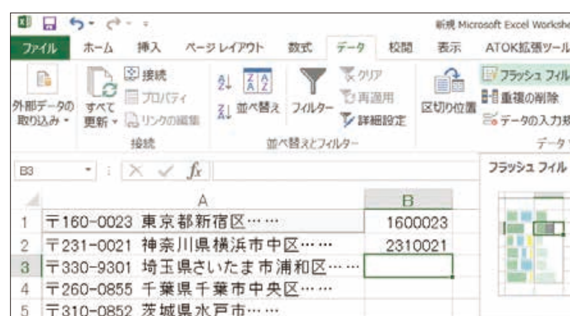
- ほかの列の値が何であろうと無視して「佐藤大輔」を出力するプログラム．
- A 列の後ろに「大輔」をつなげるプログラム．
- B 列の前に「佐藤」をつなげるプログラム．
- A 列の 1 文字目と B 列の 2 文字目の間に「藤大」をつなげるプログラム．

本質的には，システムはこれら無数の可能性を考慮しなければならない．しかも，その中から我々の意図に合うものを見つけ出さなければならない．そんなことは可能だろうか?

FlashFill は実際にこれを実現している．まず，入

☆1 http://research.microsoft.com/en-us/news/features/flashfill-020613.aspx

図 -1　FlashFill による姓名連結



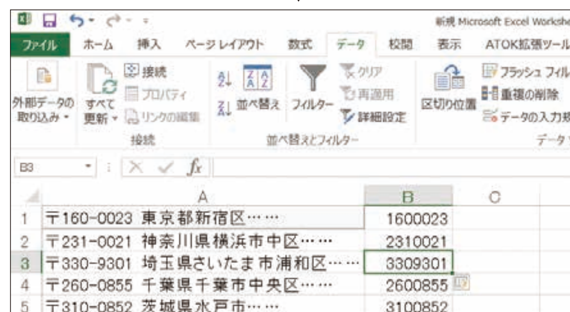


図 -2　FlashFill による郵便番号抽出

```
int lsb(unsigned int x) {
  for (int i = 0; i < 32; ++i)
    if (x & (1 << i)) return (1 << i);
  return 0;
}
```

図 -3　最下位の 1 のビットを求める関数 lsb

力となる列から出力となる列を作るプログラムの可能性をすべて求める．ここで可能性として考えるプログラムは，入力列中の一部を切り出し，並び替え，必要なら適当な文字を挿入し，連結して出力列を得るものである．可能性は各行ごとに調べ，すべての行で共通するものだけを残す．次に，これらの可能性の中で，さまざまな基準に照らして最も望ましいと予想されるものを求めユーザに提示する．大まかには，できる限り入力（特にある列全体）を使い，少ない回数の文字列連結で出力を得るものを優先する．そのため，図 -1 の例では CONCATENATE 関数を 1 回使うだけのものが最も望ましいと判断されている．

スケッチング：分からない部分はシステムに聞く

最近，自動プログラム作成の研究はちょっとしたブームになっており，FlashFill のように実用的なシステムとして広く使われるようになったものまで現れ始めている．次は，近年のブームを牽引した，スケッチング [2] という技法を紹介する．

スケッチングでは，ユーザはプログラムの概形は与えるが，その詳細を書き切らなくてもよい．細部はシステムが自動的に埋めてくれる．例として，ビット列の最下位の 1 のビットを発見するプログラムを考えよう．たとえば，01100100 には 00000100，01011010 には 00000010 を出力できればよい．すぐに思いつくのは**図 -3** のループを用いたプログラムだが，実はこれをビット演算で高速に実現できる．

いま「ビット反転・ビット and・加算を組み合わせればよかったはず」としか覚えていなかったとしよう．とはいえ，うろ覚えであっても，**図 -4** のようなプログラムなら書ける．ここで ?? はシステムに埋めてほしい部分である．つまり，x に何かを足してビット反転したものと x に何かを足したもののビット and をとる」というプログラムを表している．2 行目の assert 関数

```
y = ~(x + ??) & (x + ??);
assert(y == lsb(x));
```

図 -4　最下位の 1 のビットを求めるプログラムの概形

```
y = ~(x + 0xFFFFFFFF) & (x + 0);
assert(y == lsb(x));
```

図 -5　図 -4 から得られるプログラム

```
xold = x; yold = y;
if (??) x = x ^ y; else y = x ^ y;
if (??) x = x ^ y; else y = x ^ y;
if (??) x = x ^ y; else y = x ^ y;
assert(y == xold && x == yold);
```

図 -6　x と y の値を入れ替えるプログラムの概形

```
xold = x; yold = y;
if (0) x = x ^ y; else y = x ^ y;
if (1) x = x ^ y; else y = x ^ y;
if (0) x = x ^ y; else y = x ^ y;
assert(y == xold && x == yold);
```

図 -7　図 -6 から得られるプログラム

は「y の値は先ほどの lsb 関数の結果と同じにならなければならない」とシステムに伝えている．スケッチングシステムは，これに対し，**図 -5** のように ?? の部分を正しく埋めたプログラムを作ってくれる．

次の例として，変数 x と変数 y の値を入れ替えるプログラムを考える．もちろん，片方の値を一時的に保存する変数を使えば簡単だが，ビット xor を使えばそのような変数は必要ない．どうやればいいだろうか？

ともかく適当な順序でビット xor を繰り返せばよいのだ，という発想で与えたプログラム概形が**図 -6** である．3 回ほどビット xor を繰り返せ，という意図である．なお，xold と yold という変数は結果が望むものであることを assert 関数を介してシステムに伝えるためのものである．スケッチングシステムにこれを入力すると，**図 -7** のプログラムを出力する．if 文の条件が適切に埋められており，欲しかったのは y = x ^ y; x = x ^ y; y = x ^ y; というプログラムだったと分かる．

以上のように，分からない数値を自動的に埋めてくれるのがスケッチングである．これを使うと，効率のためにテクニカルなプログラムを書く際などに，大まかなロジックから完全なプログラムを得ることができる．しかも，スケッチングで用いられている技法をさらに発展させることで，プログラム全体を作り上げることもできる．たとえば，Gulwani ら [3] は，「ビット演算や加算を数回ずつ使う」という情報だけから，「プログラムの概形の概形」に相当するものを自動的に構成することで，図 -5 や図 -7 のようなプログラムを自動的に構成するシステムを提案している．

スケッチングの実現

スケッチングではプログラム中の変数が障壁となる．図 -4 であれば，変数 x がどんな値であっても正しく y を計算しなければならない．もし x の値が決まっていれば問題はずいぶん簡単になるはずだ．

実は，スケッチングシステムはまさにこの考えに基づいている．システムはまず適当な値で変数を埋め，その場合には正しいプログラムを生成する．たとえば，勝手に x = 01100100 とし，この場合に 00000100 を返すプログラムを作るのだ．もちろん，こうして得られたプログラムは，ほかのケースでも正しい結果を返すとは限らない．そこで次に，得られたプログラムが正しい結果を返さない変数値（反例）を探しだす．そして今度は，先ほどの変数値だけではなく見つかった反例でも正しく結果を返すプログラムを生成する．このような手続きを，反例が見つからなくなるまで繰り返す．

このアプローチをとるとなると，変数値が決まった場合のプログラムの生成，および反例発見の方法が課題となる．スケッチングシステムはこれらを充足可能性問題（SAT）と呼ばれる問題に翻訳し，それに対するソルバ（SAT ソルバ）に解かせている．SAT や SAT ソルバについては本稿の範囲を超えるが，おおざっぱには，SAT はある種のパズルであり，SAT ソルバはそのパズルを解くシステムだと思えばよい．近年 SAT ソルバは目覚ましい進歩を見せており，かなり大きく複雑な問題も高速に解くことができる．スケッチ

```
public class Utils {
  public void backupFile(String fname) {
    String bname = fname + ".bak";
    ▯
  } copy file fname to bname
}   FileUtils.copyFile(new File(fname), new File(bname))
    FileUtils.copyFile(new File(bname), new File(fname))
    FileUtils.copyFileToDirectory(new File(fname), new
    File(bname))
    FileUtils.copyFileToDirectory(new File(bname), new
    File(fname))
    FileUtils.copyFile(<arg>, new File(fname))
```

図 -8　AnyCode の利用イメージ [4)]

ングの技術的なポイントは，反例発見を繰り返すというアプローチをとることによって，その高速な SAT ソルバを活用できている点にある．

AnyCode：自然言語での説明からプログラムを自動補完

プログラムを書いていると，ライブラリ関数の呼び出し方が分からなくて困ることがある．関数名を数文字入力すれば補完候補を挙げてくれる統合開発環境もあるが，完全に忘れてしまっていたり，そもそも知らなかったりする場合はどうしようもない．また，引数にどのような値を渡せばよいのか悩むことも多い．何とかならないだろうか？

AnyCode[4)]はこのような状況で力を発揮する．AnyCode は統合開発環境での補完を強力にしたようなもので，自然言語を使える点が特徴である．図 -8 に AnyCode を利用するイメージを示した．ここでは，fname という名前のファイルのバックアップとして bname という名前のファイルを作りたいのだが，ファイルのコピーのための関数を忘れてしまった状況を考えている．このとき，その関数を挿入したい場所にカーソルを合わせ，

copy file fname to bname

とシステムに入力する．すると以下のような補完候補が出力される．

- FileUtil.copyFile(new File(fname), new File(bname))
- FileUtil.copyFile(new File(bname), new File(fname))}
- FileUtil.copyFileToDirectory(new File(fname), new File(bname))

⋮

ここで最初の候補を選べば，無事に fname を bname へコピーできる．

ここで注目すべき点が 2 点ある．まず，入力はただの英語であること．今回は「copy」「file」が実際に呼び出す関数名の一部であったが，ユーザがそれを意図したわけではない．もう 1 つは，関数名だけではなく，その引数を含め完全な関数呼び出し式を補完候補として提示していることである．このため，ユーザはライブラリ関数の利用方法をそれほど悩まなくて済む．

もう 1 つ例を挙げよう．今度は新しいファイルを作りたいとする．このとき，

make file

と入力すると，以下のような補完候補が得られる．

- new File(<arg>).createNewFile()
- new File(<arg>).isFile()
- new File(<arg>)

⋮

まず目を引くのは「make」という単語が候補に現れていないことである．ライブラリ関数名では代わりに「create」が使われているのだ．AnyCode はこのように意味を踏まえた検索を行う．また，<arg> も目を引く．これは，この部分に適切な引数を与えなければならないことを表す．往々にしてユーザからの入力は不十分である．そのような場合でも，できる限りの式を構築し，足りない部分を <arg> で表してユーザに提示するのだ．

AnyCode の実現

AnyCode の実現の方針は FlashFill に近い．ユーザの問合せにマッチしそうなライブラリ関数を列挙し，

	意図の表現	探索範囲の制限	探索手法	生成プログラムの大きさ
FlashFill	入出力例	文字列操作	全探索	小
スケッチング	プログラム概形	?? を埋める	反例検索＋SAT ソルバ	小
AnyCode	自然言語	ライブラリ関数	学習した確率モデル	小

表 -1　FlashFill，スケッチング，AnyCode の比較

それを良さそうなものから順にユーザに提示する．より良さそうな候補を高速に発見するために，AnyCode はさまざまな工夫を行っている．以下，図 -8 のケースを例に説明する．

まず，品詞や修飾・被修飾関係などの構文構造を利用している．今回の例であれば，ユーザの入力を構文解析すれば，copy が動詞，file がその目的語で変数名 fname と関連し，変数名 bname は前置詞 to とともに copy を修飾する副詞句をなす，という情報が得られる．さらに，ライブラリ関数の名前や型に対しても同様の解析を行っておく．たとえば，`File` 型の引数を 2 つとり `Unit` 型の結果を返す `copyFile` 関数は，動詞 copy の目的語が file であり，さらに file および unit という名詞を含む文と解釈される．これら構文解析の結果を照合し，文の主要な要素，たとえば主語・動詞・目的語など，が対応するかを調べることで，ユーザが意図したであろうライブラリ関数を発見する．なお，各要素がどの程度対応しているかの計算は類義語データベースに基づいて行う．このアプローチにより，問合せの細部に影響されることなく，ユーザの意図に合った結果を得ることができている．

もう 1 つの重要な工夫は，事前に GitHub 中のプログラムを解析し，各ライブラリ関数がどのような形で呼び出される可能性が高いかを表す確率モデルを作る点である．この確率モデルを用いることで，実際のプログラムに実際に現れそうな式を優先して生成している．`copyFile` 関数であれば，引数に `new File(var)`（`var` は文字列型の変数）の形の式が現れやすいということを事前に学習しておく．この学習結果を入力と比較すれば，`var` の部分が `fname` や `bname` になったものが補完候補として適切であることが分かる．

自動プログラム作成の進化：その理由と展望

ここまで 3 つの自動プログラム作成手法を概観してきた．使ってみたいと思えるものはあっただろうか．

自動プログラム作成自体は古くからの研究トピックである．しかし，自動プログラム作成は以下のような理由から非常に難しく，一時期は実用化は諦められかけていた．

- ユーザの意図をシステムに伝えるのが難しい．ユーザが意図を完全に表現できるなら，それはもはやほぼプログラムであり，自動プログラム作成のありがたみがない．一方，不完全な情報からプログラムを自動的に作るのは非常に難しい．
- 考えられ得る答の候補が膨大．あらゆるプログラムが候補となるため，可能性は無数にある．その中から，ユーザの意図にマッチするものをうまく抽出しなければならない．しかも，ユーザの意図が不完全に表されている場合などでは，それにマッチするプログラムは多数ある．
- 大きなプログラムにスケールしない．比較的初期の研究でも，小さなプログラムの自動作成に成功したという報告はある．しかし，上記問題のため，大きく複雑なプログラムの作成はきわめて困難であり，そのため実用的ではないと思われた．

これらの課題に最新の研究はどう対処しているのだろうか．**表 -1** に本稿で取り上げた手法の比較を示す．

まず，3 件とも作るプログラムの類型，つまり探索すべきプログラムの範囲をかなり制限している．FlashFill では文字列操作処理，スケッチングでは `??` を埋めて得られるものだけ，AnyCode は基本的にライブラリ関数の呼び出しだけだ．システムが実際に生成しているプログラムもお世辞にも大きくはない．し

かし，近年では小さなプログラムの需要というのは案外大きい．ライブラリやフレームワーク等が充実してきたため，既存の処理をうまく組み合わせさえすれば，比較的小さなプログラムで複雑なことが実現できるようになった．そのため，小さな，しかも限られた形式のプログラムであっても，自動で作ることができればメリットはかなりある．特に，FlashFill と AnyCode はこのようなプログラミングを明確に指向したデザインになっている．

とはいえ，依然として考えるべきプログラム候補は膨大である．実は，候補探索の手法が改善されたことが近年の発展を支えている．たとえば，スケッチングが用いている SAT ソルバや，AnyCode が使っている学習ベースのアプローチは，近年大きな進歩を見せたものである．このような高度な探索手法のおかげで，適切な候補を現実的な時間で発見することができている．

さらに，AnyCode はユーザの意図を推し量るために自然言語処理の手法を利用している．筆者の知る限り，このような高度な技法を用いてユーザの意図を捉えようとしているものは比較的少ない．が，ユーザの意図の理解が自動プログラム作成における最大の課題であることを考えると，今後はこのようなアプローチのシステムが増えるだろうと予想している．

以上のような経緯で自動プログラム作成技術は進化してきた．FlashFill のような例外はあるが，現時点ではまだ実用レベルには達していないものも多い．しかしそう遠くないうちに，統合開発環境などから，一般のプログラマが自動プログラム作成技術を利用できるようになるだろう．それらはもしかすると，皆さんを日夜悩ませている問題を解決してくれるかもしれない．

参考文献

1) Gulwani, S., Harris, W. R. and Singh, R. : Spreadsheet Data Manipulation using Examples, *Commun. ACM*, 55(8), pp.97-105 (2012).
2) Solar-Lezama, A. : Program Sketching, *STTT*, 15(5-6), pp.475-495 (2013).
3) Gulwani, S., Jha, S., Tiwari, A. and Venkatesan, R. : Synthesis of Loop-free Programs, In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pp.62-73 (2011).
4) Gvero, T. and Kuncak, V. : Synthesizing Java Expressions from Free-form Queries, In *Proc. 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015*, pp.416-432 (2015).

（2016 年 2 月 8 日受付）

森畑明昌（正会員） morihata@graco.c.u-tokyo.ac.jp

2009 年東京大学大学院情報理工学系研究科博士後期課程修了．博士（情報理工学）．2008 年日本学術振興会特別研究員，2010 年東北大学電気通信研究所助教を経て，2014 年より東京大学大学院総合文化研究科講師となり現在に至る．プログラミング言語の基礎理論，特にプログラム変換・生成を用いたアルゴリズム構成に興味を持つ．