変数名を用いないプログラミングの試み

服部 隆志[1]

概要：プログラミングにおいて変数名を適切に付けることは重要であると言われているが、変数の数が多くなると、適切な名前を考えだすのが難しくなってくる。また、プログラムの修正によって変数名が適切でなくなったのにそのまま放置され、ソースコードを読む際に誤解を与える場合がある。さらに、プログラムの動作を説明するには、変数を用いて抽象的に説明するよりも具体的な値を用いる方が分かりやすいことがよくある。
以上のことから、変数に名前を付けないプログラミング言語を提案する。ソースコードは、具体的な値を用いた具体例によって記述し、複数の具体例から「例によるプログラミング」の手法を利用して、任意の値に適用可能な実行可能コードを生成する。これによって、ソースコードそのものが動作の具体的な説明として読めるようになる。すべてを具体例で書こうとするとスケーラビリティの問題があるが、一部分を具体例で書き、残りは通常の変数名で書くことも可能である。

キーワード：例によるプログラミング, 初心者向きプログラミング言語

Proposal of a Programming Language without Variable Names

Takashi Hattori[1]

Abstract: It is important to give suitable names to variables when writing a program. It is difficult, however, to invent suitable names if we have many variables. The names may become not-suitable if the program is changed, which causes misunderstanding. Besides, concrete values are often more helpful than variable names when we try to understand a program.
From the observation above, we propose a programming language that does not use variable names. Source code is written with concrete values. Given multiple concrete values, executable code is generated by the technique of 'programming by examples'. In this manner, the source code can be considered as an explanation of execution. Though there is a scalability problem, it is possible that a part of a program is written with concrete values while remaining part is written with ordinal variable names.

Keywords: Programming by examples, Programming language for beginners

1. はじめに

プログラムを書く際には、ほとんどどんな言語であっても、変数に名前を付けなければならない。大きなプログラムの場合は多数の名前を考えなければならず、適切な名前を考えるのに相当の時間を使ってしまうこともある。なお、関数など他にも名前を付けなければいけないものはたくさんあるが、この研究では変数名だけを取り上げることにする。

[1] 慶應義塾大学 環境情報学部
Faculty of Enviroment and Information Studies, Keio University

変数に適切な名前を付けることが可読性に大きな影響を与えることは明白であるが、どのような名前が適切であるかということは形式的に定義できない。それは、名前というものは形式的には同一性のみが問題とされ、実際にどういう名前を付けるかということは、そこにコメントを書くのと同じような意味しかないからである。プログラムの意味と変数名の間に一貫性を保証するような制約が全くないことから、次のような問題が生じる。

- よく似た使われ方をする変数が多くある時は、適切な名前を考え出すのが難しい。
- 自分には理解しやすい変数名であっても、他人が読む

時に理解しやすいとは限らない。

- プログラムの修正によって変数の使われ方が変わっても、変数名が変更されず、適切な名前でなくなっていることがある。
- 初心者は、変数名の自然言語的な意味からプログラムの意味を推測する傾向が強く、誤解を招きやすい [7]。

上記の問題を解決するため、そもそも変数に名前を付けないようなプログラミング言語が可能であるかどうかを検討する。

2. 具体例を使用したプログラミング

アルゴリズムを厳密に記述するためには、変数を使用して抽象的に記述することは不可避である。しかし、人間に対してアルゴリズムを説明する場合は、抽象的な記述よりも具体的な例を使用して説明する方が分かりやすいことが多い。

現状でも、ソフトウェア開発において具体例を利用する場面はいくつかある。

- ドキュメントにおいて、抽象的な機能の説明だけでなく具体例を書くことにより理解を促進する。これを発展させたものとして、アルゴリズムの動きを説明するためのアルゴリズム・アニメーションがある。
- デバッガで具体的な実行の軌跡を表示させることによって、バグの発見を容易にする。一般に実行記録をすべて表示させると膨大になるので、人間に理解できるように可視化する dynamic program visualization の手法がいろいろと研究されている [4]。
- テストケースとして具体例を与えることで、正しく動くかどうかを検査する。

上のような利用方法は、プログラムは抽象的に書くことを前提とし、それを理解したり、検査したりするために具体例を使うものである。それに対して、プログラムを書くために具体例を使う手法として、「例によるプログラミング」[1], [3], [6] がある。

例によるプログラミングは、いくつかに分類できる。まず、入力と出力の組、あるいは実行前と実行後の状態の組を与えてアルゴリズムを推論するものがあるが、これはごく単純なアルゴリズムしか推論できないという問題がある。次に、具体的な操作の列を与えて、それを汎化することで抽象的なアルゴリズムを得るものがある。これは特に「実演によるプログラミング」と呼ばれることがある。汎化の手法としては、自動的に繰り返しや条件分岐を発見し、最小汎化を行う方法が主流であるが、プログラマが明示的に汎化する場所を指定するもの [5] もある。

例によるプログラミングの研究では、実行例の具体的な表現から抽象的なアルゴリズムの記述を生成するものが中心であるが、スケーラビリティなどの問題からなかなか実用的に普及する段階まで達していない。この研究では、アルゴリズムの基本的な構造の記述は既存言語をそのまま利用し、変数の部分だけを具体的に記述することによって、変数に名前を付けずに実行可能なプログラムを記述することを目指す。

3. Concrete Language の提案

3.1 概要

この章では、変数に名前を付けないプログラミング言語である Concrete Language (仮称) を提案する。前章でも述べたとおり、これは例によるプログラミングの一種であり、複数の具体例の記述から anti-unification[2] で最小汎化を行い、実行用のコードを生成する。

基本となる言語は手続き型言語であれば何でもよいが、ここでは C のサブセットを元にした文法を使用し、実行用コードとして C プログラムを生成することにする。なお、まだ設計中なので構文や機能は変更する可能性がある。

3.2 単一例からの汎化

汎化を行うためには、原則として複数の具体例が必要であるが、いくつかの場合においては単一例から汎化を行う。

3.2.1 代入文の左辺

代入文の左辺に定数が現れた場合は、明らかに変数に汎化する必要がある。例えば、Concrete Language の次のようなコードを考える。

```
1 = 1;
```

左辺の定数 1 は変数に汎化され、次のようなコードが生成される。ただし、x は Concrete Language の処理系が適当に選んだ名前である。

```
int x = 1;
```

3.2.2 同じ定数の出現

すでに変数に汎化された定数と同じ定数は、同じ変数に汎化する。これは最小汎化ではないが、通常はこの方が便利だと思われる。例えば、次のような Concrete Language コードがあるとする。

```
1 = 1;
2 = 1 + 1;
```

1 行目の代入文の左辺 1 は変数 x に汎化されるので、それ以後の定数 1 の出現も x に汎化される。

```
int x = 1;
int y = x + x;
```

たまたま同じ定数が現れると、意図しないところが汎化されてしまうので、汎化してほしくない定数には後ろに :fix をつける。例えば、上の例で 2 行目は x を 2 回足すのではなく、x に 1 を足すという意図であるならば、次のように :fix を付けることで汎化を防ぐ。

```
1 = 1;
2 = 1 + 1:fix;
```

そうすると、次のようなコードが生成される。

```
int x = 1;
int y = x + 1;
```

3.2.3 仮引数

関数定義の仮引数に定数が現れた場合も、明らかに変数に汎化する必要がある。関数定義の本体に同じ定数が現れた場合は、前節で述べたように同じ変数に汎化する。例えば、次のような Concrete Language コードがあるとする。

```
int foo(1) {
  return 1;
}
```

これから、次のようなコードが生成される。

```
int foo(int x) {
  return x;
}
```

3.3 繰り返しの汎化

3.3.1 無限ループの場合

繰り返しを汎化するためには、繰り返しの 1 回目、2 回目、3 回目、…に対応する具体例を適当な個数記述し、それを repeat end で囲む。例えば次のように書く。

```
repeat
  { printf("%d\n", 1); 2 = 1 + 1; }
  { printf("%d\n", 2); 3 = 2 + 1; }
  { printf("%d\n", 3); 4 = 3 + 1; }
end;
```

このコードには 3 個の具体例が含まれているので、その最小汎化を求めると次のようになる。

```
  { printf("%d\n", x); y = x + 1; } }
```

次に、3.2.2 節で述べた同じ定数の汎化を拡張し、代入文の左辺で汎化された定数が、繰り返しの次の回で別の箇所に現れていれば、それは同じ変数に汎化する。上の例では、y に汎化された定数は代入文の左辺であり、繰り返しの 1 回目と 2 回目の値 2，3 が x に汎化された定数の 2 回目と 3 回目の値に一致するため、x と y を同じ変数にすると、次のようになる。

```
  { printf("%d\n", x); x = x + 1; } }
```

さらに、汎化された各変数について、繰り返しの 1 回目で現れる定数 (代入文の左辺を除く) を初期値として、繰り返しの前に代入する。結果として得られるコードは次のようになる。

```
int x = 1;
while (1)
  { printf("%d\n", x); x = x + 1; } }
```

具体例を書く個数に制限はないが、個数が少ないと意図したものとは異なる汎化が行われる可能性があるので、その場合は具体例を多くする必要がある。

3.3.2 ループの終了条件

無限ループの汎化はできたので、次にループの終了条件を記述する。

まず前節と同様に繰り返しの最初の数回の具体例を書き、その後に '...' と繰り返しの最後の 1 回の具体例を書く。さらに、end の後に because と終了条件の具体例を書く。例えば、前節の例で 10 回繰り返したら終了したい場合、次のように記述する。

```
repeat
  { printf("%d\n", 1); 2 = 1 + 1; }
  { printf("%d\n", 2); 3 = 2 + 1; }
  { printf("%d\n", 3); 4 = 3 + 1; }
  ...
  { printf("%d\n", 10); 11 = 10 + 1; }
end because (11 > 10:fix);
```

終了条件の具体例には、繰り返しの最後の 1 回の具体例と同じ定数を使用し、同じ変数に汎化されるようにする。上の例では 11 がそうなっているが、10 は変数ではなく本来の定数を意図している。ところが、たまたま繰り返しの最後の 1 回の具体例にも同じ 10 が現れているため、汎化を防ぐために:fix を付ける。これによって得られるコードは次のようになる。

```
int x = 1;
while ( !(x > 10) )
  { printf("%d\n", x); x = x + 1; } }
```

3.4 条件分岐

条件分岐がある場合は、それぞれの分岐に対応した具体例を複数個記述する。分岐の位置には because と条件の具体例を書く。条件は、否定が付いたものと付かないものの二種類がなくてはならず、否定を除いて汎化したものが生成される条件式になる。例えば、階乗を計算する関数 fact は次のようになる。

```
int fact(3) {
  because (3 != 1)
    return fact(3 - 1) * 3;
  end;
}


int fact(1) {
```

```
  because !(1 != 1)
    return 1;
  end;
}
```

これから、次のようなコードが生成される。

```
int fact(int x) {
  if (x != 1) {
    return fact(x - 1) * x;
  else {
    return x;
  }
}
```

3.5 不明な値

具体例を書くと言っても、実行してみないと具体的な値がわからない場合がある。その場合は、不明な値として書く。

- 式の中に書くべき値が不明であるが、その箇所は他の具体例から変数に汎化される場合、不明な値として '_' を書く。
- 代入文の左辺に書くべき値が不明であり、その値がそれ以後の式で参照される場合、代入の記号を '~=' に変更し、左辺には任意の値を書く。それ以後の式は、その任意の値が計算によって得られた値であると仮定して書いていく。実際に実行されるのは変数に汎化されたコードであるから、Concrete Language の段階で書いた値が正しい値とまったく異なっていても問題ない。

例えば、1 から入力された値までの合計を計算するコードは次のようになる。ただし、input() は標準入力から整数を入力する関数[*1]である。

```
1000 ~= input();
repeat
  { 1 = 0 + 1; 2 = 1 + 1; }
  { 3 = 1 + 2; 3 = 2 + 1; }
  { 6 = 3 + 3; 4 = 3 + 1; }
  ...
  { 9999 ~= _ + _; 1001 = _ + 1; }
end because 1001 > 1000
printf("%d\n", 9999);
```

最初に input() によって数値を入力するが、その値は当然実行しないとわからないので、代入記号を '~=' にし、仮に 1000 が入力されたのものとする。

繰り返しの中の式は最初の 3 個の具体例で汎化されるので、最後の具体例では正しい値を書かずに '_' を使って構わない。ただし 1001 という値は終了条件で参照されるので書く必要がある。また、合計の値も printf で参照されるので値を書く必要があるが、ここでは不明な値として仮に 9999 と書くことにする。printf のところでも 9999 と書くことにより、同じ変数に汎化される。

結果として次のようなコードが生成される。

```
int x = input();
int y = 0; int z = 1;
while ( !(z > x) )
  { y = y + z; z = z + 1; }
printf("%d\n", y);
```

3.6 配列

配列変数に関しては、配列変数名の代わりに、要素の具体的な値を大括弧で囲んで書くことで具体例とする。ただし、配列の要素に対する代入は、代入される値と、どの配列のどの位置に代入されるかという情報が必要なので、代入文の左辺にはまず代入される値を書き、その後に where と配列の具体的な値、添字の値を書く。例えば、次のようなコードを考える。

```
int foo([3, 7, 4, 1], 2) {
  (5 where [3, 7, 4, 1][2]) =
    [3, 7, 4, 1][2] + 1;
  return [3, 7, 5, 1][2]
}
```

仮引数に [3, 7, 4, 1] が現れるので、これが配列変数 x に汎化される。関数本体の中の [3, 7, 4, 1][2] は配列の 3 番目の要素 4 を表す。代入文の左辺には、代入される値である 5 と、代入する位置を示す [3, 7, 4, 1][2] の両方を書かなければならない。この代入文以後は [3, 7, 5, 1] が同じ変数 x に汎化される。

結果として次のようなコードが生成される。

```
int foo(int x[], int y) {
  x[y] = x[y] + 1;
  return x[y];
}
```

4. 考察

4.1 ポインタ型

上に挙げた例はすべて int 型のデータを使用したプログラムであったが、他にも char, float, double など定数を記述できるデータ型であれば同様に使用できる。構造体型も、配列のように定数を記述する構文を用意すれば使用可能と思われる。しかし、ポインタ型のように具体的な定数を陽に書かないデータ型は使用できない。

ただし、ポインタ型を使用する部分は通常通り変数名を

[*1] scanf() は、ポインタを具体例で書くのが難しいため、使用しない。

用いて記述し、それ以外のデータ型に対しては具体例を記述して汎化することで、通常のプログラムと Concrete Language のプログラムを混合して使用することは可能である。

4.2 テスティング

代入文は、左辺が定数であることから、等式と考えることもできる。したがって、等式が成り立つという制約をテスティングの時に利用可能である。

- 代入文の右辺に関数呼び出しを含まない場合、定数だけを含む等式となるので、両辺が等しいかどうかテストすることができる。
- 代入文の右辺が単一の関数呼び出しの場合、実引数と返り値が定数で記述されているので、その関数のユニットテスト用のテストケースとして利用できる。

代入文はできるだけ上記の二種類のどちらかになるように分解してコーディングすると、ソースコードを書くことがそのままテストケースを書くことになる。例えば、次のような代入文があるとする。

```
10 = f(1) + g(2);
```

このままではテストに利用できないが、次のようにすれば、関数 `f, g` のユニットテストのテストケースとして使うことができる。

```
4 = f(1);
6 = g(2);
10 = 4 + 6;
```

このようにすると余分な変数が生成されるが、コンパイラ言語であれば最適化されるので問題ない。

4.3 スケーラビリティ

Concrete Language は通常のプログラムと比べて記述すべき量が多くなるが、まだ実装が完成していないため、どの程度多くなるかは測定できていない。特に問題になると予想される点をいくつか下に挙げる。

- 繰り返しがネストしている場合、内側の繰り返しで書かなければいけない具体例の個数はべき乗オーダーで増えていく。例えば、四重の繰り返しを生成したい時に、それぞれの繰り返しに対して 3 個の具体例が必要とすると、最も内側の繰り返しの中の文は、$3^4 = 81$ 個の具体例を書かなくてはならなくなる。この問題に対しては、既に他の箇所で十分な個数の具体例が書いてあれば省略可能にすべきだと思われるが、どのような構文にするのがよいかは検討中である。
- 関数や繰り返しの中に含まれる文の数が多い場合、複数の具体例を書くのは、ほとんど同じ構造を何回も書くことになるので面倒である。エディタの機能として、1 個目の具体例をテンプレートとして定数部分だけを書き換えていくことができれば便利であると思われる。
- 具体例として大きな配列を書くのは非常に困難である。小さな配列の具体例を書いて、大きな配列用のコードを生成できるようにするか、配列の要素を省略して書く記法が必要になると思われる。

4.4 リーダビリティ

Concrete Language は通常のプログラムと比べて、具体的な値を用いているので理解しやすいのではないかと思われるが、変数に名前が付いていないことから、意味を類推する手がかりが少ないという可能性もある。この点については、まだ実用的な規模のプログラムを書いていないので検証できていない。

ソースコードに具体例を書かなくても、デバッガで実行履歴を見れば同じ効果が得られるのではないかということも考えられるが、ソースコードに書く場合は、繰り返しの途中を省略したり、境界値など重要な値を選んで具体例を書くことができるので、より人間に理解しやすいものになるのではないかと思われる。

5. おわりに

例によるプログラミングの一種として、変数名を使わないプログラミング言語 Concrete Language を提案した。具体例を使って記述することにより、リーダビリティの向上や、テスティングの際に利用できる情報の増加などが期待できる。

スケーラビリティの問題や、ポインタ型など本質的に抽象的なデータ型の存在から、すべてを具体例で記述するのは難しいと言わざるを得ない。しかし、通常のプログラムの中で、具体例で書いた方が理解しやすいと思われる部分だけを Concrete Language で記述することも可能であり、有効性があるのではないかと思われる。

ドキュメントにアルゴリズムの説明を書く場合、自然言語による説明、フローチャートなどの図、具体的な実行例などを書くことが多い。Concrete Language は、具体的な値を記述し、しかもプログラムと同じ厳密さを持っているので、説明に適しているのではないかと思われる。

初心者がプログラミングの学習を行う際に、具体例を使って記述することで学習が容易になる可能性がある。さらに、通常のプログラムと実行履歴から Concrete Language を生成する逆方向の変換も用意することで、自分で書いたプログラムの誤りを理解し、修正することも容易になるのではないかと思われる。

Concrete Language はまだ処理系の実装が完了していないので、今後は実装を完成させ、上記の予想の検証を行う予定である。

参考文献

[1] Bauer, M. A.: Programming by examples, *Artificial Intelligence*, Vol. 12, No. 1, pp. 1 – 21 (1979).

[2] Bulychev, P. E., Kostylev, E. V. and Zakharov, V. A.: Anti-unification Algorithms and Their Applications in Program Analysis, *Perspectives of Systems Informatics* (Pnueli, A., Virbitskaite, I. and Voronkov, A., eds.), Lecture Notes in Computer Science, Vol. 5947, Springer Berlin Heidelberg, pp. 413–423 (2010).

[3] Cypher, A.(ed.): *Watch What I Do: Programming by Demonstration*, MIT Press (1993).

[4] Diehl, S.: *Software Visualization*, Springer (2007).

[5] Kahn, K.: Generalizing by Removing Detail, *CACM*, Vol. 43, No. 3, pp. 104–106 (2000).

[6] Lieberman, H.(ed.): *Your Wish Is My Command : Programming by Example*, Morgan Kaufmann (2001).

[7] 服部隆志: 初心者教育に適したサンプルプログラムの書き方, 夏のプログラミング・シンポジウム「ソフトウェアの品格〜たとえば鑑賞に耐えるプログラムを目指して〜」報告集, 情報処理学会, pp. 71–74 (2009).