部分評価に基づく繰返し処理内のポインタ値の変化の理解支援

酒匂 駿[1]　高畑 竜馬[2]　吉田 敦[3]　蜂巣 吉成[3]　桑原 寛明[3]

概要：本論文では，繰返し処理におけるポインタ変数の変化を把握するために，ポインタの値を部分評価によって計算し，ポインタの値の埋め込みおよび間接参照値への置換えの 2 種類の表現からなる可視化手法を提案する．また，手法に基づいたツールを実装し，現実的なソースコードに適用できるか評価を行った．その結果，評価対象とした 19 の関数のうちポインタの値の埋め込みを用いた可視化が 19 の関数で適用可能であり，間接参照値への置換えを用いた可視化が 17 の関数で適用可能という結果になった．考察として，模擬実験を行いツールの有無での理解支援への効果を確認し，誤読の可能性について議論した．

A Method for Understanding Changes in Pointer Values during Iteratives based on Partial Evaluation

Abstract: In this paper, we propose a method for understanding changes in pointer values during iteratives. The method provides two distinct views of source texts. The first, called the abstract expansion view, inserts relative pointer values computed through partial evaluation after the pointer variable references. The second, called the concrete expansion view, replaces dereferencing expressions of pointer variables with their evaluated results. We have implemented a tool based on the method and evaluated its applicability by applying it to a set of functions in open-source software. The tool generated abstract expansion views for 19 of the 19 functions and concrete expansion views for 17 of the 19 functions. We also discuss the effectiveness of the method in understanding changes in pointer values with conducted a simulated experiment of using tool and without tool, and the possibility of misunderstanding.

1. はじめに

プログラミング学習においてコードリーディングは効果的な手段の一つである．しかし，C 言語の仕様を一通り習った初学者にとって，実用的なソースコードの理解は難しい．その原因の一つとして，C 言語で使われるポインタがあり，コードリーディングの取り組みへの障害となる．これは，ポインタの指す先やその変化，すなわち，ポインタがバッファ上のどこを指し，その指し先がどう移動していくのかをうまく把握できないからである．実用的なソースコードでは間接参照やインクリメントが複合的に使われ，ポインタの把握がさらに難しくなる．そこで，本論文では，ポインタを用いた典型的な処理であるバッファ操作を対象として，把握すべきポインタの指す先やその変化をソースコード内に表現する理解支援手法を提案する．元のソースコードと提案手法で生成するソースコードを見比べながら読むことで，ポインタの指す先やその変化をイメージしやくなり，コードリーディングに取り組みやすくなる．なお，本論文での「ポインタ」とはメモリのアドレスで表現される「指し示すもの」という抽象的な概念を指し，そのポインタとしてのアドレスを保持する変数を「ポインタ変数」と呼ぶ．

本論文で提案する表現方法に「抽象展開」と「具象展開」の 2 種類がある．抽象展開は，ポインタの相対的な変化値を部分評価と記号計算により求め，変数参照に埋め込む方法である．具象展開は，ポインタに関わる具体的な値を与え，部分評価により式を評価値に置換えていく方法である．

[1] 南山大学大学院理工学研究科
Graduate School of Science and Technology, Nanzan University
[2] 南山大学理工学部卒，現株式会社トヨタシステムズ
Graduate of the Faculty of Science and Technology, Nanzan University, currently TOYOTA SYSTEMS, Nakamura, Nagoya, 450-6332, Japan
[3] 南山大学理工学部
Faculty of Science and Technology, Nanzan University

抽象展開は，元のソースコードの動作を一般性を保ったまま理解することに役立ち，具象展開は具体的な値における動作例を見ることで動作イメージを明確にすることに役立つ．なお，本論文ではバッファ処理を例として用いるが，これは，ポインタが動的に変化する処理で，かつ，本提案手法が効果的な例であるからである．ポインタの演算や繰返し文の取り扱いなどは他の処理にも適用可能である．

以下では，支援の全体像を示したあと，2 つの展開方法についてそれぞれ提案する．その後，オープンソースソフトウェアへの適用可能性について評価し，理解支援への効果について議論する．

2. 関連研究

実行時の変数の値の変化を可視化する手法を小山ら [4] が提案している．変数の値を表形式で提供するので，元のソースコードとの対応が取りにくい．また，1 つの実行例に基づくので，一般性を持った形での支援ができない．

蜂巣ら [2] は，実引数に与えられたポインタが呼出し先にどう関わるかを図示する手法を提案している．また，ポインタの可視化が可能なツールとして PythonTutor[8] があり，ポインタ，間接参照値を図示する．これらは実行におけるある時点の状態を図示しており，変化の過程を理解するには複数の図を見比べる必要がある．ソースコードと図では表現方法が異なり，ソースコードを理解するための対応関係を把握しにくい．

プログラムの動作の確認には一般的にはデバッガが用いられるほか，Cross ら [1] の提案したプログラムの動的な可視化や Ishizue ら [3] の提案した実行時のメモリの可視化，支援ツールの比較を行った Sorva ら [5] が取り上げた支援ツールがある．ただ，いずれもある実行時点の状態を確認する支援であり，一連の繰返しの中でどのようにポインタが変化するかを俯瞰するのには適さない．

本論文の提案手法では，部分評価を用いてポインタの変化をソースコード内に埋め込むので，実行時の変化や元のソースコードとの関係を把握しやすい．一方，テキストのみでの支援となるで，図示と比べ，直感的なわかりやすさに欠ける欠点がある．

3. ポインタ変数の変化の理解支援の提案

3.1 提案の概要

バッファ操作処理を理解する際には以下の点を把握する．

(1) ポインタが指しているバッファ位置
(2) ポインタの間接参照値
(3) 繰返し文中での (1)，(2) の継続的な変化

(1)，(2) の把握はポインタを理解する際に必要な基本的な要素であり，(3) の把握は繰返しを使用して操作を行うバッファ操作処理に必要な要素である．しかし，ポインタという抽象的な概念に基づいて，頭の中でポインタとバッファの関係をイメージし，かつ，その変化を正確に把握していくことは容易ではない．

本論文では，(1) から (3) の把握を支援するために，ポインタに関わるソースコードの理解支援として，ポインタ変数に対する参照の情報の付加や間接参照値への置換えをコード内に表現する理解支援手法を提案する．これにより，プログラムの動作過程でのポインタの指す先やその変化が明示的になり，ソースコードを読む際にポインタをイメージしやすくなる．ソースコード内に表現するための変換には抽象展開と具象展開の 2 種類がある．抽象展開では，プログラムの実行に必要な入力は与えず，ソースコードから静的に求まる値に基づいて部分評価を適用し，ポインタ変数にその値の情報を付加する．具象展開では，入力を与え，その入力に基づいて部分評価を適用し，ポインタの間接参照を参照先の値に置換える．提案手法の全体像を図 1 に示す．学習者は，理解したい繰返し文を含むソースコードと初期値となる変数の値を支援ツールに入力する．支援ツールは繰返し文に対する 2 つの展開表現を生成し，出力する．また，最終的に理解したいのは展開前のソースコードであることから，図 2 のように，2 つのコードを並べて表示し，コードに色をつけることで，対応関係を把握しやすくする．

この手法では，抽象展開におけるポインタの扱いと，繰返し文の扱いが問題となる．まず，ポインタ変数はメモリのアドレス値を持つが，メモリの割り当て方はコンパイラに依存し，その値自体に意味はない．そのような値を示すことはむしろ理解を妨げることになる．理解に必要な情報はポインタの変化であるので，ポインタは初期値からの相対値で表現する．以下，この値を「ポインタ相対値」と呼ぶ．

繰返し文については，繰返しごとに変数の値が変化するので，ソースコード上の 1 つの変数参照に複数の値が存在することになり，単純にそれらをすべて提示しても理解にはつながらない．そこで繰返し文は 1 回の繰返しごとに展開し，展開された箇所の変数参照に情報の付加または置換えを行う．具象展開では，具体的な値が与えられ，繰返し回数が決まり，繰返し文をすべて展開することで繰返し文自体を削除できる．一方，抽象展開は，具体的な値が与えられないので，繰返しの回数が不明であり，繰返し文を除去することができない．そこで，繰返し回数を記号で表現し，繰返し内および繰返し後のポインタ相対値は，その記号を含んだ値で表現する．

3.2 前提条件

本論文で対象とするバッファ操作は，繰返し文によりバッファを走査しながら変換や判定を行うものを想定する．また，提案手法の実現が可能な範囲として，次の条件を満すものとする．

- 繰返し文ではポインタ変数を参照している．

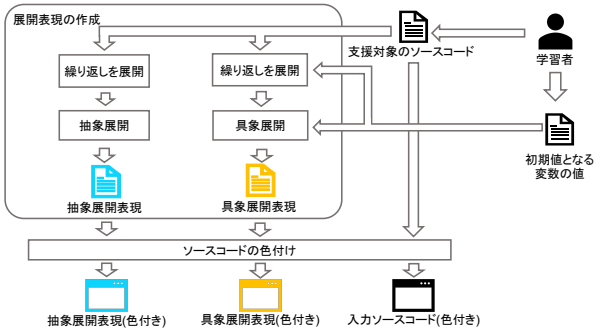

図 1 提案手法の全体像
Fig. 1 Overview of purpose method

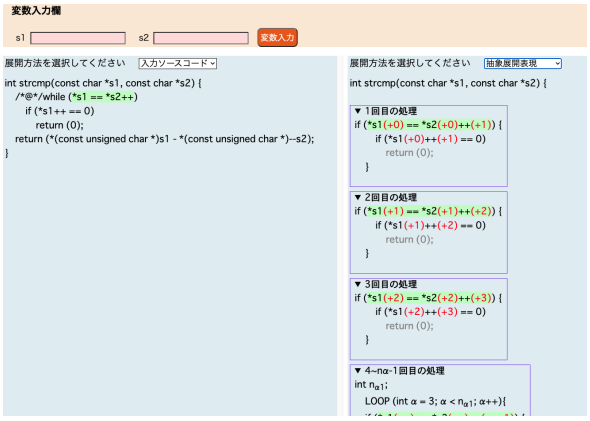

図 2 ツールのインタフェース
Fig. 2 The interface of the tool

- 繰返し文の入れ子は 2 重までとする．
- 関数呼出しを含まない．
- 前処理指令を含まない．

本論文の支援を適用するために，ポインタ変数が含まれていることを前提とする．繰返し文の展開量が膨大になり理解支援に繋がらなくなるのを防ぐため，繰返し文の入れ子は 2 重までに制限する．また，部分評価の際に使用する CFG の作成が困難なことから，関数呼出しおよび前処理指令を含まないソースコードを対象とする．この条件の緩和は今後の課題である．

現実的なソースコードにおいてもこれを満たすものはあり，本論文では，具体例として NetBSD[9] のライブラリの文字列処理関数 (string.h) を用いる．NetBSD は，移植性が高い素直な実装になっている．

3.3 抽象展開におけるポインタ相対値の求め方と提示方法

抽象展開では，まずポインタ相対値を求める必要がある．相対値であるので，基点となるポインタを定め，そこからの差を求める．基点は，次のものとする．

- 宣言により与えられた初期値
- 部分評価で右辺値が求まらない代入式での代入値

ポインタ相対値は，それぞれの基点から計算可能な範囲について求める．なお，宣言されただけの変数は，本来，初期化されていないが，ここでは便宜上，初期化されたものとする．仮引数の場合には関数呼出し時の実引数が初期値となる．

```
1  char *p(+0), *q(+0);
2  *q(+0) = *p(+0);
3  q(+0)++(+1); p(+0)++(+1);
4  *q(+1) = *p(+1);
5  q(+1)++(+2); p(+1)++(+2);
6  *q(+2) = *p(+2);
```

図 3 ポインタ相対値の表示例
Fig. 3 An example view of inserting relative pointer values

算出されたポインタ相対値は各ポインタ変数の出現の直後に挿入する．具体的には，ポインタ変数の出現を p，その出現時のポインタ相対値を N としたとき，ポインタ変数の出現の後ろに $(+N)$ を付け，次のように表記する．

p$(+N)$

C 言語の構文を満たさない表現であるので，挿入されても誤解なく読み解ける．なお，本論文では下のソースコードのテキストと区別をしやすくするために，ポインタ相対値の記述を斜体で表現している．また，実装したツールでは色を変更する方法を採用している．

ポインタの演算ではインクリメントとデクリメントが良く使われるが，値の参照と代入の両方が行われるので，参照された値と代入値の両方を示す必要がある．また，各演算子は後置と前置で処理が異なるので，その区別ができる必要がある．そこで，前置の場合は式の直後に 2 つ連続して並べる．ポインタ変数の出現を p，加減算が評価される前のポインタ相対値を $(+n_0)$，加減算の評価後のポインタ相対値を $(+n_1)$ としたとき，次のように表記する．

++p$(+n_0)(+n_1)$

一方，後置の場合は，演算子の適用後に値が変化することを示すために，次のように変数ではなく，演算子の直後に挿入する．

p$(+n_0)$++$(+n_1)$

挿入例を図 3 に示す．ポインタ変数 p，q について，それぞれポインタ相対値を埋め込んだものである．1 行目ではポインタ変数の宣言がされているのでポインタ相対値 $(+0)$ が埋め込まれており，その後 3，5 行目でインクリメントされることで $(+1)$，$(+2)$ とポインタ相対値が増加している．

3.4 具象展開におけるポインタの間接参照値の求め方と提示方法

具象展開では，入力が与えられているので，ポインタの間接参照値は計算可能な式を順次評価していくことで求められる．ただし，ポインタの間接参照値を求めるにあたり，評価に必要な変数の値が算出できていない場合には，その間接参照値は計算しない．

間接参照のうち値が求まるものは，その式の出現を間接

```
1 *q = *p;
2 q++; p++;
3 *q = +p;
4 q++; p++;
5 *q = *p;
```

```
1 p = "Hello";
2 *q = 'H';
3 q++; p++;
4 *q = 'e';
5 q++; p++;
6 *q = 'l';
```

図 4 間接参照値への置換例
Fig. 4 An example view of replacing dereferencing expressions

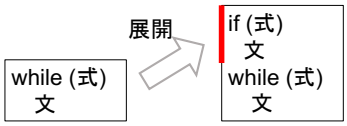

図 5 ループ展開の例
Fig. 5 An example of loop expansion

参照値に置換える．なお，バッファ操作を想定しているので，間接参照により整数値や文字といった値が求まるものとし，ポインタへのポインタは扱わないものとする．間接参照値への置換の例を図 4 に示す．図 4 では，ポインタ変数である p に文字列"Hello"の先頭へのポインタが代入されており，その代入式を元に左の図の 1 行目の*p が右の図の 2 行目では'H' に置き換わっている．同様に左の図の 3，5 行目の*p も右の図では'e' と 'l' に置き換わっている．

ポインタ相対値も計算はしているが，ポインタ相対値の変化は置換えた値から読み取れるので，変換後のコードの複雑化を避けて，具象展開ではポインタ相対値は表示しない．また，間接参照によりポインタが指す先の値を求める式のみを対象とし，代入の左辺は左辺値を求めているので置換えない．

あらかじめ入力を与える必要があるが，利用者がツールを介して入力するものとし，入力された値はコード上に，入力対象の変数への代入式として表現するものとする．これにより，コード上で入力内容が明確になるとともに，部分評価にあたってはコード内の代入のみで処理できるので，実現が容易になる利点がある．

3.5 繰返し文の展開

繰返し文の展開とは，1 回の繰返しで評価される式や文を繰返し文の前に配置することを基本操作とし，それを繰返し適用することである．展開の操作を図 5 に示す．図 5 では， while 文の 1 回分が展開されることで，if 文として while 文の上部に追記されている．if 文の形をしているのは， while 文の条件式での判定を実行するためである．

繰返し文の展開内容は抽象展開表現と具象展開表現で異なる．抽象展開表現では，初期値がなく繰返し回数が決まらないので，最初の 3 回分の処理の展開，そのあとに任意の回数の繰返し，最後の処理の展開の 3 つの部分で表現する．具象展開表現では，完全に展開される方が理解性を高

```
1 char *strcpy(char *s1, const char *s2){
2   char *tmp = s1;
3   while (*s1++ = *s2++)
4     ;
5   return tmp;
6 }
```

図 6 strcpy.c[7] のソースコード
Fig. 6 Source code of strcpy.c[7]

```
1 char *strcpy(char *s1(+0), const char *s2(+0)){
2   char *tmp(+0) = s1(+0);
3   if (*s1(+0)++(+1) = *s2(+0)++(+1)) {
4     ;
5   }
6   if (*s1(+1)++(+2) = *s2(+1)++(+2)) {
7     ;
8   }
9   if (*s1(+2)++(+3) = *s2(+2)++(+3)) {
10    ;
11  }
12  int nα1; /*nα1 は任意の繰返し回数*/
13  LOOP (int α = 3; α < nα1; α++){
14    if (*s1(+α)++(+α+1) = *s2(+α)++(+α+1)) {
15      ;
16    }
17  }
18  if (*s1(+nα)++(+nα+1) = *s2(+nα)++(+nα+1)) {
19    ;
20  }
21  if (*s1(+nα+1)++(+nα+2) = *s2(+nα+1)++(+nα+2)) {
22    ;
23  }
24  return tmp(+0);
25 }
```

図 7 図 6 の抽象展開表現
Fig. 7 Abstract expansion of Fig. 6

めることから，初期値は繰返し回数に関わるすべての変数に入力し，繰返しの条件式が偽になるまで展開する．また，入力ソースコードと展開表現では形が異なるが，実行順序は一致するように展開している．展開表現の例として，文字列をコピーする strcpy 関数 [7] のソースコードである図 6 を用いる．図 6 の while 文に対して抽象展開を用いた結果を図 7 に，図 6 の while 文に対して具象展開を用いた結果を図 8 に示す．以降，各展開表現についてそれぞれ説明する．

3.5.1 抽象展開表現

抽象展開表現では繰返し文の展開とポインタ相対値の表示を行う．繰返し回数は求められないので，繰返しの 3 回分を展開したあと，残りは繰返しのまま残す．また，繰返しの終了時の処理を理解しやすくするために，繰返し終了時の処理も展開する．すなわち，繰返し回数を n とすると，次の 3 つの要素で構成する．

- 初めの 1，2，3 回目の処理

```
1  char *strcpy(char *s1, const char *s2){
2    char *tmp = s1;
3    s1 = "hello";
4    s2 = "hey";
5    if (*s1++ = 'h') {
6      ;
7    }
8    if (*s1++ = 'e') {
9      ;
10   }
11   if (*s1++ = 'y') {
12     ;
13   }
14   if (*s1++ = '\0') {
15     ;
16   }
17   return tmp;
18 }
```

図 8 図 6 の具象展開表現
Fig. 8 Concrete expansion of Fig. 6

- 4〜$n-1$ 回目までの処理
- 終わりの n 回目と繰返し終了時の処理

このうち，4〜$n-1$ 回目までの処理は，繰返しを表す独自表現 LOOP を用いる．形式は for 文と同じであるが，元のソースコードに for 文があった場合に，それと混同しないように，見掛け上異なる予約語を用いている．実体としては，LOOP は for に置き換わる前処理マクロである．繰返しを初めの 3 回分展開することで，繰返し文での具体的な変化の仕方を確認することができ，繰返し終了時の処理を展開することで，繰返しの終了条件の確認が行える．

繰返し終了時の処理とは，繰返し文の最後に行う条件式の評価である．繰返し 1 回分を繰返し文の後ろに，初めの方の展開と同じく if 文を用いる．ただし，終了時には条件判定は偽になるので，if 文の内部の処理は実行されない．なお，偽になる if 文は，そのままでは真と誤解される可能性がある．そこで，実装したツールでは，実行されない内部処理を目立たない色で表現することで，誤読を防ぐ工夫をしている．

理解支援として，繰返しの条件が偽となって，繰返しを終了する過程を示したいので，return 文や break 文などのジャンプ文が実行されないよう，それが実行を制御する if 文は条件を偽として扱う．

展開されない繰返し部分では，繰返し回数を数えるカウンタ変数を用いるが，ソースコード内で使われている変数名と重ならないよう α，β といったギリシャ文字を用いる．繰返し文が入れ子の場合は，それぞれの繰返し文で異なるギリシャ文字を用いる．また，繰返し回数の上界は，カウンタ変数に合わせて n にギリシャ文字の添字をつけた変数 n_α，n_β を用いる．なお，入れ子関係にはない繰返し文が複数存在する場合には，添字に番号をつけて $n_{\alpha 1}$，$n_{\alpha 2}$ を用いる．

```
1  int strncmp(const char *s1, const char *s2, size_t n){
2    if (n == 0)
3      return (0);
4    do {
5      if (*s1 != *s2++)
6        return (*(const unsigned char *)s1 - *(const
             unsigned char *)--s2);
7      if (*s1++ == 0)
8        break;
9    } while (--n != 0);
10   return (0);
11 }
```

図 9 strncmp.c[9] のソースコード
Fig. 9 Source code of strncmp.c[9]

3.5.2 具象展開表現

具象展開表現では学習者が特定の変数の値をツールに入力することで，繰返し文の展開と間接参照値が表示される．この表現ではソースコードに出現するすべての変数の値をツールに入れることなく一部の変数の値の入力のみでの展開が可能である．入力できる変数候補は関数の仮引数と関数内の局所変数である．変数の値をツールに入れると，その変数の代入式が支援対象の繰り返し文の直前に追記され，その情報を元に繰返し文を展開する．

また，具象展開表現では繰返し回数を決める変数の値をツールに入力しているので，繰返し条件が偽になるまで展開され，条件式の評価が可能な場合はそれを行う．条件式の評価をすることで，実行される文を明確化し，学習者が読むコード量を減らす．また，抽象展開と同様に，実装では条件式の真偽について誤解が生じないよう，実装したツールでは目立たない色に変更する工夫をしている．

3.5.3 繰返し文の指定

複数の繰返し文があるとき，それらすべてを展開すると元のソースコードとの対応関係が取りにくくなるため，指定指示子`/*@*/`を用いて，繰返し文の直前に指定指示子を置くことで展開する繰返し文の指定を行える．

3.5.4 break 文，continue 文の展開

繰返し文を展開したとき break 文と continue 文は使用できない．そこで，break 文と continue 文は goto 文に置き換える．ジャンプ先のラベル文は break 文の場合には，それが出現した繰返し文の直後にラベル文を置き，continue 文の場合は，展開した各実行文の直後にラベル文を置く．例として繰返し文に break 文を含む図 9 を用いる．図 9 の 8 行目の break 文を展開後の図 10 では goto 文に置換え，break 文のジャンプ先である繰返し文の後ろにラベル文 BREAKGOTO を埋め込むことで break 文の処理の整合性を保ちつつ繰返し文の展開を行う．

3.5.5 2 重繰返し文の展開

2 重繰返し文の場合でも展開操作を再帰的に適用することで繰返し文の展開が可能である．しかし，すべて展開を

```
1  int strncmp(const char *s1, const char *s2, size_t n){
2    if (n == 0)
3      return (0);
4    {
5      if (*s1(+0) != *s2(+0)++(+1))
6        return (*(const unsigned char *)s1(+0) - *(const
             unsigned char *)--s2(+1)(+0));
7      if (*s1(+0)++(+1) == 0)
8        goto BREAKGOTO;
9    }
10   if (--n != 0) {
11     if (*s1(+1) != *s2(+1)++(+2))
12       return (*(const unsigned char *)s1(+1) - *(const
             unsigned char *)--s2(+2)(+1));
13     if (*s1(+1)++(+2) == 0)
14       goto BREAKGOTO;
15   }
16   if (--n != 0) {
17     if (*s1(+2) != *s2(+2)++(+3))
18       return (*(const unsigned char *)s1(+2) - *(const
             unsigned char *)--s2(+3)(+2));
19     if (*s1(+2)++(+3) == 0)
20       goto BREAKGOTO;
21   }
22   if (--n != 0) {
23     if (*s1(+3) != *s2(+3)++(+4))
24       return (*(const unsigned char *)s1(+3) - *(const
             unsigned char *)--s2(+4)(+3));
25     if (*s1(+3)++(+4) == 0)
26       goto BREAKGOTO;
27   }
28   int n_{α1};
29   LOOP (int α = 4; α < n_{α1}; α++){
30     if (--n != 0) {
31       if (*s1(+α) != *s2(+α)++(+α+1))
32         return (*(const unsigned char *)s1(+α) - *(
               const unsigned char *)--s2(+α+1)(+α));
33       if (*s1(+α)++(+α+1) == 0)
34         goto BREAKGOTO;
35     }
36   }
37   if (--n != 0) {
38     if (*s1(+n_α) != *s2(+n_α)++(+n_α+1))
39       return (*(const unsigned char *)s1(+n_α) - *(
             const unsigned char *)--s2(+n_α+1)(+n_α));
40     if (*s1(+n_α)++(+n_α+1) == 0)
41       goto BREAKGOTO;
42   }
43   if (--n != 0) {
44     if (*s1(+n_α+1) != *s2(+n_α+1)++(+n_α+2))
45       return (*(const unsigned char *)s1(+n_α+1) - *(
             const unsigned char *)--s2(+n_α+2)(+n_α+1));
46     if (*s1(+n_α+1)++(+n_α+2) == 0)
47       goto BREAKGOTO;
48   }
49   BREAKGOTO:;
50   return (0);
51 }
```

図 10 図 9 の抽象展開表現
Fig. 10 Abstract expansion of Fig. 9

行うと文量がかなり増加する．そこで，抽象展開表現の場合は 2 重繰返し文の内側の展開と内側と外側両方の展開の 2 パターンを採用する．内側の展開は外側の繰返し文をそのままに，内側の繰返し文のみ対して最初の 3 回分の処理の展開，そのあとに任意の回数の繰返し，最後の処理の展開を配置する．両方の展開は内側の繰返し文を任意の回数の繰返しと最後の処理の展開の 2 部分のみに展開し，それを実行文として外側を最初の 3 回分の処理の展開，そのあとに任意の回数の繰返し，最後の処理の展開を配置する．

3.5.6 do-while 文の展開

do-while 文は for 文と異なり必ず 1 回実行文が実行されるという特徴がある．そこで，do-while 文の展開では通常の展開に加えて繰返し文の条件式を持たない実行文の展開を始めに行う．

3.5.7 無限ループの展開

無限ループでは繰返し回数に上界がないので，繰返し文の最後の処理を展開しない．無限ループの条件の書き方は様々あるが，本論文では簡単のため明示的に条件式に 1 を入れたものを無限ループとして扱う．

4. 実装と評価

提案手法が実現可能であることを示すために，ソースコードから抽象展開表現と具象展開表現の 2 つを生成するツールを実装した．また，NetBSD[9] のライブラリの文字列処理関数 (string.h) に適用して提案手法の実現可能性の評価をした．実装では以下の要素は対象外とした．

- 配列
- 構造体
- 2 重繰返し文
- 各分岐に繰返し文を含む if 文

配列，構造体は本論文の本質ではないことから，2 重繰返し文の展開および各分岐に繰返し文を含む if 文は評価の対象にしたライブラリにあまり含まれておらず，実装の複雑さを避けることから行わなかった．ただし，2 重繰返し文の展開は単体の繰返し文の展開の自然な応用であり，それが実現可能であることは単体の繰返し文について実現ができれば明らかである．これらの実装は今後の課題とする．

4.1 実装

4.1.1 展開表現の作成方法

ツールの実装にあたり，字句解析やソースコードの書き換えには TEBA[6] のツールを用いた．TEBA を用いて属性付き字句系列を生成し，字句列の書き換えパターンを用いて for 文，while 文，do-while 文の展開を行う．また，部分評価については，TEBA の部分評価器を拡張して実現した．TEBA の部分評価器では属性付き字句系列から CFG を生成し，生成された CFG の実行経路を辿ることでソースコードの部分評価を行う．ただし，この評価器はポイン

タがないプログラムに限定されたプロトタイプである．そこで，ポインタ相対値および間接参照値を求めるためにポインタ変数の宣言時に記号表の変数に被りがないような値を仮想アドレス値として記録するよう部分評価器の拡張を行った．ポインタ相対値は元の仮想アドレス値との差を求めて表示し，間接参照値は仮想アドレス値が指している値から求める．

4.1.2 提示方法

抽象展開表現や具象展開表現を単独で学習者に表示しても，元のソースコードへの理解支援としては効果が薄い．そのため，学習者への提示は Web を用いて，図 2 のように入力ソースコードと展開表現を並べて表示することで常に照らし合わせながら理解することができる．また，本提案手法では LOOP 表現やポインタ相対値の表示，間接参照値への置換え等，独自の表現方法を用いている．そこで，HTML と CSS を用いることにより入力ソースコードと独自表現を色で区別している．繰返し文を展開することにより，元のソースコードとの対応関係を取るのが難しくなるため，繰返しの初期化式，条件式，変化式にそれぞれ対応したハイライトを付け，展開した繰返し処理ごとに枠で囲うことで明確にしている．

4.2 評価

提案手法が現実的なソースコードの理解支援に使えるか確認するために，オープンソースソフトウェアの関数群に適用可能か評価した．適用可能の定義として抽象展開表現と具象展開表現でそれぞれ 4 つの基準を設定し，その定義を元に関数群に対して抽象展開表現，具象展開表現が適用可能か判定した．

文字列操作関数は入力する文字列を繰返し文で 1 文字ずつ読み取ってバッファを操作するので，繰返し文内でポインタが頻繁に変化する．これらの関数に本論文の手法を適用することで頻繁に変化するポインタをコード上に可視化でき，それが理解支援に繋がるため，評価の対象にした．NetBSD[9] の string ヘッダには 47 の関数があるが，3.2 節で挙げた前提条件を満たし，2 重繰返し文を含まない関数は 47 の関数のうち 19 であった．これらを本論文の評価の対象とする．

4.2.1 抽象展開表現の基準

抽象展開表現では繰返し文の展開とポインタ相対値の計算が 3.3 節で挙げた仕様通りにできているかを評価する．本論文の仕様に合わせて以下の基準を定めた．

A1 1〜3 回，4〜$n-1$ 回，n 回目，繰返し終了の処理，の形で展開可能か．

A2 ポインタ相対値の表示が宣言または計算不可能な代入時に 0 かどうか．

A3 繰返し回数に応じてポインタ相対値の計算ができているか．

A4 if 文の中に return 文，break 文，continue 文があるものと繰返し終了の処理を実行されない処理としているか．

A1 の基準は抽象展開表現の繰返し文の展開に着目している．抽象展開表現は繰返し文の式の値によらず，1〜3 回，4〜$n-1$ 回，n 回目，繰返し終了の形を基本としているので，繰返し文に対してこの形での展開可能かを A1 の基準とする．A2 以降の基準はポインタ相対値の計算に着目している．A1 の基準を満たした場合でも繰返し文中の実行文によっては計算がずれる場合がある．ずれる場合がどのような文のときなのかを明確にするために，ポインタ相対値の計算に関してさらに細かい基準を設けるべきであると考え，A2〜4 の基準を設けた．A2 の基準ではポインタ相対値の基点に着目している．基点からの相対位置でポインタ相対値の値が決まるので，その基点が正しいかを確認する．A3 の基準は任意の繰返し文や繰返し終了処理は α，n_α の記号を含めたポインタ相対値が表示されているので，それが正しいかを確認するために，繰返し回数に応じたポインタ相対値になっているかを確認する．A4 の基準は本来，条件が偽になる if 文の内部を実行するとその後の実行文のポインタ相対値がずれるので確認する．以上の 4 つの基準を満たした場合は抽象展開表現を適用可能とする．

4.2.2 具象展開表現の基準

具象展開表現では次の 4 つの基準を設定する．

C1 初期値が設定された繰返しに関わる変数を元に繰返しの展開可能か．

C2 初期値が設定された変数が繰返し文前に追記されているか．

C3 間接参照値が求まるときはその値に置換え可能か．

C4 実行経路が正しく明確化されているか．

C1 の基準は入力された変数に対して繰返し文が展開される回数に着目する．具象展開表現では繰返し回数を決める変数の値を入力しているので，展開可能かの確認をその設定された初期値に対して展開される回数で判定する．C2 以降の基準は，間接参照値への置換えが正しくできているかに着目する．こちらも抽象展開表現の基準と同様に，展開可能でも繰り返しの文の内容によっては，置換えが途中でずれたり，できない場合が発生すると考えられるので，繰返し文の展開と間接参照値への置換えに対してそれぞれ基準を設けた．C3 の基準は初期値が設定された変数の情報がなければ，その後の展開と間接参照値への置換えができないので，繰返し文前に追記されているかを確認する．C4 は部分評価の結果よって文が薄い色で表示されて学習者が読む部分が明確化するので，それが正しいかを確認するために実行経路が明確になっているかを着目する．以上の 4 つの基準をすべて満たした場合は具象展開表現を適用可能とする．

表 1 抽象展開の適用可能な関数の割合
Table 1 Percentage of applicable functions of abstract expansion

	A1	A2	A3	A4
適用可能な関数	19	19	19	19
適用可能な関数の割合	100%	100%	100%	100%

表 2 具象展開の適用可能な関数の割合
Table 2 Percentage of applicable functions of concrete expansion

	C1	C2	C3	C4
適用可能な関数	19	19	19	17
適用可能な関数の割合	100%	100%	100%	89%

表 3 すべての評価条件を満たす関数の割合
Table 3 Percentage of applications to the functions satisfying all condition

	抽象展開表現	具象展開表現
適用可能な関数	19	17
適用可能な関数の割合	100%	89%

4.2.3 評価結果

抽象展開表現と具象展開表現のそれぞれで各基準について評価した結果，抽象展開表現では表 1 のように，具象展開表現では表 2 のようになった．表 1 と表 2 で各基準をすべて満たした関数の数と評価対象の関数の数の割合を表 3 に示す．割合は小数点第 1 位以下を切り捨てで示している．

対象の関数のうち抽象展開表現はすべての関数で適用可能であった．具象展開表現は 17 の関数が適用できて約 89% が適用可能だった．抽象展開表現は繰返しに関わる変数に初期値が設定されていなくても展開するため，具象展開表現より適用しやすいことが表 3 の割合からもわかる．

具象展開表現が適用できなかった要因として実行文に関数呼び出しを含む関数が挙げられる．関数呼び出しは実装ができておらず，評価が正しく行われないが，評価するよう実装できれば問題なく適用が可能と思われる．

5. 考察

5.1 理解支援の評価

理解支援の評価を行うには，実際に被験者を用いた比較実験が望ましいが，被験者の数を十分に揃えられなかったので，ここでは具体例による効果の検証を行う．まず，具体的な事例に基づいて，理解にどう貢献するか検証した．また，提案手法ではソースコードを読むときに頭の中に把握すべき情報を減らすことができると考えられることから，記憶量の観点から効果を考察する．また，提案手法はコード量を増やすことで逆に理解を妨げる可能性があるので，コード量の観点での課題を考察する．

5.1.1 具体例による効果の検証

具体例による効果の検証は図 9 の strncmp を用いて行う．strncmp の直感的な理解は，2 つの文字列のポインタを同時にずらしながら，不一致がないかを確認していき，文字数 n に到達するかナル文字に遭遇するまで変化していく処理である．しかし，図 9 では同時に変化しておらず，`s1` は 7 行目でインクリメントされ，`s2` は 5 行目と 6 行目でそれぞれインクリメントとデクリメントが行われている．この処理を正確に理解するためには，6 行目の return 文では 2 つの文字列について先頭から同じ文字数だけ進んだ先の文字を比較していることを理解する必要がある．また，7 行目の条件判定では，2 つの文字列のポインタが共にナル文字に到達している必要があり，条件判定によって，同じ相対位置の文字が等しいこととそれらの文字がナル文字であることの判定をしていることを理解する必要がある．

支援ツールを用いた場合，前者については，図 10 の 6 行目のポインタ相対値 *(+0)* など 4 回の展開や 39 行目の一般化されたポインタ相対値 *(+n_α)* から相対的に同じだけずれていることがすぐ読み取れる．後者については，図 10 の 5 行目で，`*s1`*(+0)* と`*s2`*(+0)* を比較していること，7 行目で`*s1`*(+0)* を比較していることから，同じ相対位置の文字を対象にしていることがわかる．また，この関係が 4 回の展開および一般化された繰返しについても同じ関係を持っていることから，すぐに読み取れる．

支援ツールを用いない場合，ここに登場する *(+0)* や *(+1)* の情報をすべて頭で把握しておく必要がある．これは，ポインタの数が増えたときや，このように繰返しの中で条件分岐が組み合わさっているときには把握するべき要素が増えることになる．

具象展開表現では，具体的な文字列を用いた場合の，比較されている文字を確認することができる．図 9 の具象展開表現である，図 11 では，入力が *s1="hello" s2="hey" n=3* の結果を示し，実際に 2 つの文字列のポインタが同時にずれていることを確認できる．

5.1.2 記憶量

ここでは，理解をする際に頭の中で思い浮かべるポインタの量を記憶量として考察する．

繰返しを伴うポインタ変数の処理の典型であるバッファ操作処理の理解には，繰返し毎のポインタの値と指す値が必要になる．本論文ではポインタの値を部分評価により計算し，ポインタ相対値の埋め込みと間接参照値への置換えによる可視化を行う．ポインタ相対値を埋め込むことで理解の過程でポインタの値の変化を正確に記憶する必要がなくなり，間接参照値への置換えでポインタの間接参照値を常に把握する必要がなくなる．可視化された結果を参照することで理解に必要な値を補完できる．間接参照値への置換えには変数の入力が必要なため，バッファ操作処理の理解に効果的な入力を考える必要があるが，頭の中で記憶す

```
1  int strncmp(const char *s1, const char *s2, size_t n){
2    if (n == 0)
3      return (0);
4    s1 = "hello";
5    s2 = "hey";
6    n = 3;
7    {
8      if ('h' != 'h')
9        return ('h' - 'h');
10     if ('h' == 0)
11       goto BREAKGOTO;
12   }
13   if (2 != 0) {
14     if ('e' != 'e')
15       return ('e' - 'e');
16     if ('e' == 0)
17       goto BREAKGOTO;
18   }
19   if (1 != 0) {
20     if ('l' != 'y')
21       return ('l' - 'y');
22     if ('l' == 0)
23       goto BREAKGOTO;
24   }
25   BREAKGOTO:;
26   return (0);
27 }
```

図 11 図 9 の具象展開表現
Fig. 11 Abstract expansion of Fig. 9

る必要があるポインタの値と指す値の量は減少する．

5.1.3 コード量

繰返しを伴う変数に埋め込みや置換えを行うにはソースコード上の変数に複数の状態がないようにする必要がある．そこで，抽象展開や具象展開で繰返し文の展開を行った．

抽象展開では繰返し文の一部を LOOP としてまとめて表現するため展開後のコード量が膨大になることはないが，2 重ループでは繰返し文の展開を再帰的に適用するため，コード量が膨大になる．変化の方法をある程度把握した箇所に対して展開の数や展開する繰返し文の選択を行えるような仕組みを導入することで，2 重以上のループについても，コード量を抑えた形で抽象展開を適用できる可能性がある．

具象展開では繰返しに関わる変数の入力を必須としているため，繰返し文が繰返し終了の処理に展開される．しかし，入力する値は学習者に依存するので入力によっては展開後のコード量が大きくなる可能性がある．

5.2 ポインタ相対値の基点

複数の事例に適用して確認をしたところ，提案手法には改善の余地があることがわかったので，その改善方法について検討する．ポインタ相対値の基点が以下の点で誤解を生む可能性がある．考察する対象のソースコードを図 12，図 12 の抽象展開表現を図 13 に示す．

```
1  char sample(char *s1, char *s2) {
2    while(*s1++ != '\0')
3      ;
4    char *s = s1;
5    char *t = s2;
6    return *s;
7  }
```

図 12 ポインタ相対値の基点が誤解を生む例
Fig. 12 An example of causing misunderstanding by the definition of the base points of relative pointer values

```
1  char sample(char *s1, char *s2) {
2    if (*s1(+0)++(+1) != '\0') {
3      ;
4    }
5      if (*s1(+1)++(+2) != '\0') {
6      ;
7    }
8    if (*s1(+2)++(+3) != '\0') {
9      ;
10   }
11   int n_{α1};
12   LOOP (int α = 3; α < n_{α1}; α++){
13     if (*s1(+α)++(+α+1) != '\0') {
14       ;
15     }
16   }
17   if (*s1(+n_α)++(+n_α+1) != '\0') {
18     ;
19   }
20   if (*s1(+n_α+1)++(+n_α+2) != '\0') {
21     ;
22   }
23   char *s = s1(+(n_α+2));
24   char *t = s2(+0);
25   return *s(+0);
26 }
```

図 13 図 12 の抽象展開表現
Fig. 13 Abstract expansion of Fig.12

```
1  char *s = s1(+n_α+2);
2  char *t = s2(+0);
3  return *s(+s1+0);
```

図 14 代入されたアドレスを基点とした例
Fig. 14 Example based on assigned address

(1) 別のアドレスを指しているときに，それを区別する情報を示していない．
(2) 計算できない値の代入時のポインタの値が基点となるので，別のポインタの値を引き継げない．

(1) では，図 12 のポインタ変数 s と t のように複数のポインタ変数が別々のアドレスを指していても本論文の手法では *(+0)* をポインタ相対値として示す．図 13 では while 文の展開結果である 2〜22 行目で計算された s1 のポインタ相対値が $(+n_\alpha+2)$ になっており，s1 が s に代入されて

いる．この場合，指し先の進んだ位置はわかるが，s と s2 が別々のアドレスを指しているかが分からないので，コードを読んでいるときに同一のアドレスを指していると勘違いする可能性がある．これを解決するには，ポインタ相対値の基点として代入されたアドレスを用い，異なる基点の値を明確化する必要がある．例えば，図 14 ような基点を用いることで，ポインタ変数 s1 のアドレスを基点としていることが明確化されて，そこからの相対位置を示せる．

(2) では，図 12 のポインタ変数 s や t ようにポインタ変数に代入した際に，代入した変数のアドレスからの相対位置が本論文の手法では扱えず，すべて 0 からの表示になる．問題の本質としてはアドレス上の相対位置を考慮する必要がある点で (1) と同じであり，(1) が解決されれば代入した変数のアドレスからの相対位置であることが示され (2) も解決される．

6. おわりに

繰り返しを伴うポインタ変数の処理では，ポインタの値や指す値が繰り返しごとに継続的に変化するので，ポインタを頭の中で保持しつつ，ポインタの値や指す値を更新していきながらコードを読み進めていくが，容易ではない．ポインタに関する処理の理解を容易にするには，ポインタを頭の中で保持する量を削減する必要があり，削減するにはポインタをいつでも確認できるように可視化する必要がある．本論文では，バッファ操作処理の理解をするために，ポインタの値を部分評価により計算し，ソースコードに対するポインタ相対値の埋め込みおよび間接参照値への置換えの 2 種類の独立する表現から構成される可視化手法を提案した．また，提案手法を実現するために展開表現の生成を行うツールを実装した．評価では，NetBSD のライブラリの中の string ヘッダにある文字列操作関数を検証対象とし，抽象展開表現は 100%，具象展開表現は約 89%が適用可能であった．考察では，具体例に基づいて効果を検証し，支援ツールを用いることで複数のポインタが変化する際の位置関係の把握が容易になることを確認した．また，頭の中で記憶する量と展開後のソースコードの可読性について確認した．その結果，ポインタの展開表現によって記憶量の削減が確認でき，抽象展開での LOOP 表現を含んだ照らし合わせ，具象展開での繰り返しに関わる変数の入力が使用者に依存して変わることが確認できた．今後の課題としてツールの拡張，ポインタ相対値の基点の改善が挙げられる．

参考文献

[1] Cross, J., Hendrix, D., Barowski, L. et al.: Dynamic Program Visualizations: An Experience Report, SIGCSE '14, pp.609–614 (2014).
[2] 蜂巣吉成，森本朱音，松尾翔馬，加藤ちひろ，吉田敦，桑原寛明：ポインタ型の仮引数を持つ関数の呼出しに対するプログラミング学習支援ツールの提案，ソフトウェア工学の基礎 XXVI (FOSE 2019)，pp.103–108 (2019).
[3] Ishizue, R., Sakamoto, K., Washizaki, H., Fukazawa, Y.: PVC.js: visualizing C programs on web browsers for novices, Heliyon, Vol.6, No.4 (2020).
[4] 小山秀明，山田俊行：変数の値の変化の可視化によるプログラム理解支援，情報処理学会論文誌，Vol.10，No.4，pp.1–11 (2017).
[5] Sorva, J., Karavirta, V., and Malmi, L.: A Review of Generic Program Visualization Systems for Introductory Programming Education, TOCE, Vol.13, No.4, pp.1–64 (2013).
[6] 吉田敦，蜂巣吉成，沢田篤史，張漢明，野呂昌満：属性付き字句系列に基づくソースコード書き換え支援環境，情報処理学会論文誌，Vol.53，No.7，pp.1832–1849 (2012).
[7] 柴田望洋：新・明解 C 言語 入門編，p.218，SB クリエイティブ株式会社 (2014).
[8] PythonTutor：Online Compiler, Visual Debugger, and AI Tutor for Python, Java, C, C++, and JavaScript(online)，入手先 ⟨https://pythontutor.com/⟩ (2024.05.22).
[9] GitHub：NetBSD(online)，入手先 ⟨https://github.com/NetBSD/src/tree/567c8efbdbecb6bc61f75356575fd2bfb358f379/common/lib/libc/string⟩ (2024.05.22).