プログラム実行過程の理解支援のための流動的参照に注目したデータ構造可視化

林 みなみ　田邉 裕大　増原 英彦

プログラム中のデータを可視化するツールでは、実行中のある時点に存在するオブジェクトとオブジェクトに対する参照をノードリンク図として表示するものが一般的である．このような表示からプログラムの実行過程を理解しようとすると (1) 参照が一様なリンクとして表示されていることと (2) 表示が一時点の状態であるために、時間とともに変化してゆく参照を追跡することが難しい．本研究は (1) 時間とともに変化するような参照は参照先の色付けによって表示し (2) それを一定期間のアニメーションとして表示する手法を提案する．ライブプログラミング環境 Kanon を拡張する形で提案を手法を実現し、データ構造を扱う正しいあるいは誤りを混入したプログラムに対してアニメーションが期待通りに表示されることを確認した．

1　はじめに

プログラムの実行状態の可視化では、プログラムの実行中に得られる情報を加工し、開発者に有用な情報として提示する。既存研究では実行ポイントのソースコード上での強調をベースに, メソッドの呼び出し関係を表す木構造 [9] [7] [4]，変数や式の値のテキスト・表・グラフ形式での表示 [9] [3] [7] [6]，オブジェクトと参照関係を表すノードリンク図 [3] [10] を表示するものがある．これらの一群の研究はプログラムの実行に伴って変化する状態を可視化対象とするため, 実行中のある一時点の状態だけでなく，複数の実行時点にわたる状態の変化をどう表示するのかも重要である．

実行状態の可視化は，プログラミング教育，アルゴリズム動作説明, デバッグといった文脈で, プログラムの動作を利用者が頭の中で逐次的に再現する負担を減らす目的で行われてきた．例えば，実行ポイントに対応するソースコードと変数を束縛する値を同時に表示すれば，代入や条件分岐によって値がどう変わり，次にどの文が実行されるかを確認できる．また，関数呼び出しや値の履歴を示せば，制御フローや想定外の挙動が生じた箇所を追いやすくなる．

本研究では, データ構造上を高頻度で走査するようなプログラムの可視化を目指し, 実行状態としてデータ構造を構成するオブジェクトとその参照関係に注目する．データ構造を扱うプログラムの可視化では，個々の変数値だけでなく，複数のオブジェクトがどのように生成され，互いにどの参照で結ばれているかが重要になる．

データ構造を対象とした実行状態の可視化では，データ構造を構成する要素とそれらの関係を一覧表示する方法として, ノードリンク図が用いられてきた [2] [10] [1]．ノードリンク図ではオブジェクトの接続関係や変数の参照先を視覚的に把握可能にし、データ構造の理解を容易にする。しかし，探索や走査のようにデータ構造の形が大きく変わらない処理では，重要なのは構造そのものよりも，現在処理が着目しているオブジェクトや, アルゴリズムの処理待ち集合が時間とともにどう変化したかである．このような情報は，既存研究がベースとする一時点のノードリンク図だけからは読み取りにくい．

本研究では，データ構造ライブプログラミング環境 Kanon(第 2 節) に基づいて，従来の可視化方式の下

A Data Structure Visualization Technique Focusing on Transient References to Support Understanding of Program Execution

林 みなみ　田邉 裕大　増原 英彦, 東京科学大学情報理工学院, School of Computing, Institute of Science Tokyo.

でプログラムの実行過程を理解しようとする場合の問題を明らかにする (第 3 節)．この問題に対し，データ構造の形を構成する参照とアルゴリズムの進行に伴って変化する参照を区別し，後者を強調してアニメーション表示する手法を提案 (第 4 節) し，Kanon 上での実現方法を示す (第 5 節)．作成した処理系は，深さ優先探索と，探索順序または訪問回数に誤りを含む二分木探索プログラムを用いたケーススタディによってその動作を確認した (第 6 節)．論文は関連研究を紹介 (第 7 節) した後に，まとめと今後の課題を述べる (第 8 節)．

2 背景: ライブデータ構造プログラミング環境 Kanon

本研究では，オブジェクト構造や参照関係がプログラムの各時点で可視化可能な環境の具体例として Kanon を用いる [10]．Kanon は，データ構造を扱うプログラムを対象としたライブプログラミング環境である．利用者がプログラムを編集すると，Kanon は編集後のプログラムを自動的に実行し，ソースコード上のカーソル位置に対応する実行時状態をノードリンク図として表示する．これにより，利用者はプログラムを編集しながら，その実行時点におけるオブジェクト間の参照関係や変数の参照先を確認できる．

Kanon において表示される「ある一時点」とは，プログラムの実行が利用者の置いたカーソル位置に到達した時点である．利用者がカーソルを別のソースコード位置へ移動すると，Kanon はその位置が実行された時点の状態を選択し，右上のノードリンク図を即座に更新する．ループや再帰呼び出しなどにより同じカーソル位置が複数回実行された場合には，一つのカーソル位置に複数の実行時点が対応する．この場合，利用者は画面右下のコンテキスト図に表示されたノードをクリックすることで，いずれかの実行時点を選択し，その時点におけるオブジェクト構造と参照関係をノードリンク図に表示できる．

図 1 に，Kanon によるデータ構造可視化の例を示す．この例では，左側にソースコード，右上にオブジェクトと参照関係，右下に実行コンテキストを表す図が表示されている．この可視化が表すのは，ある一時点におけるオブジェクト構造と参照関係である．

右上のノードリンク図では，プログラム実行中のオブジェクトがノードとして，オブジェクト間のフィールド参照が水色の矢印として表示される．また，`this` や局所変数からオブジェクトへの参照は緑色の矢印として表示される．図 1 の例では，`left` と `right` に対応する水色の矢印から木の形を，`current` などに対応する緑色の矢印から各変数の参照先を確認できる．

3 動機

Kanon のようなデータ構造を可視化できるツールを用いてプログラムの振舞いを理解することを考える。特に、本研究では，データ構造の形を変更せずに走査を行うプログラムを対象とする．

データ構造の形だけを確認しても，どのオブジェクトがどの順序で処理されたかを把握することはできないという問題がある．なぜならば、このようなプログラムでは，オブジェクト間のフィールド参照は変化しない一方で，現在処理中のオブジェクトを指す変数の参照先や，処理待ちのオブジェクトを保持する配列の内容が変化する．

具体例として，二分木を走査し，最大値を持つオブジェクトを探索するプログラムを考える．このプログラムは，処理待ちのオブジェクトを配列 `queue` に保持し，各反復において `queue` から一つのオブジェクトを取り出して `current` に代入し，その値を現在までの最大値と比較した後，子オブジェクトを `queue` に追加する．本章では，このうち木の走査過程に着目する．走査が意図どおりに行われているかを確認するには,「すべてのオブジェクトが処理されたか」,「どの順序で処理されたか」「，および `queue` のどの位置から要素が取り出されたか」を把握する必要がある．

図 2 に，このプログラムを従来の Kanon で可視化した例を示す．この図から，その実行時点で `current` が参照しているオブジェクトと，`queue` に保持されている処理待ちのオブジェクトを確認できる．一方で，従来の Kanon の可視化から探索過程を読み取る際には，次の三つの問題がある．

第一に，この図からは，`current` が過去にどのオブジェクトを参照していたかを読み取ることができ

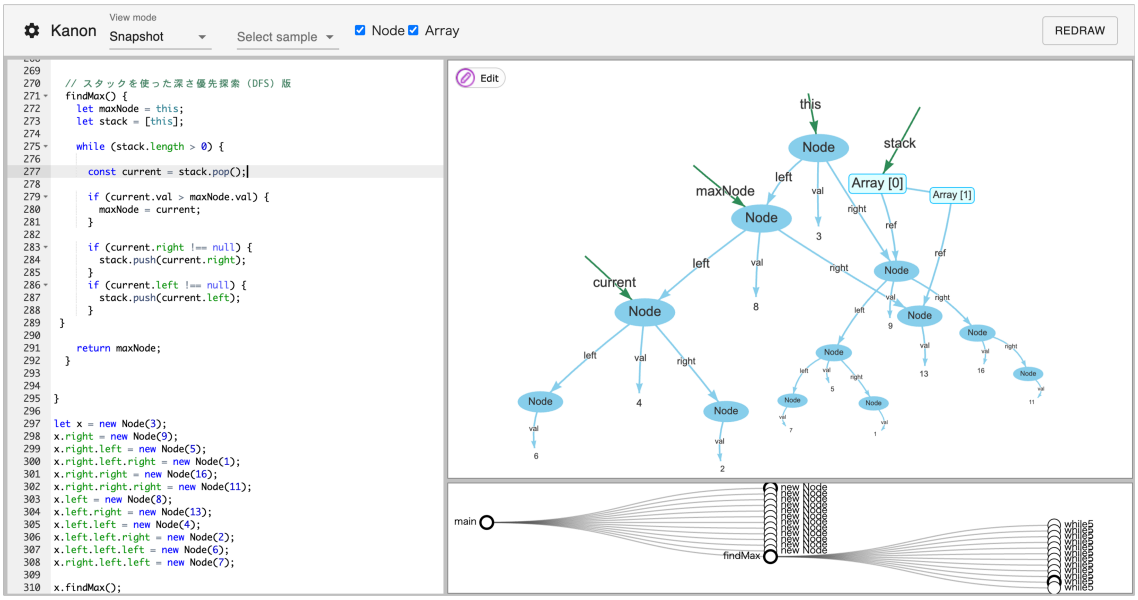


図 1: Kanon の画面構成とデータ構造可視化の例

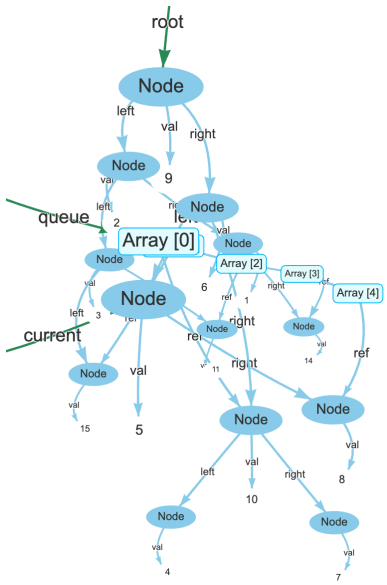


図 2: 従来の Kanon による二分木探索の可視化

ない．どのオブジェクトがどの順序で処理対象になったかを確認するには，異なる実行時点の可視化を順に表示し，`current` の参照先を見比べる必要がある．

第二に，図には木を構成するフィールド参照と，変数や配列要素からの参照が多数の矢印として表示される．そのため，利用者は多数のノードと矢印の中から，`current` の参照先や，`queue` の各要素が参照するオブジェクトを探す必要がある．

第三に，一時点の可視化だけでは，`queue` に保持されたオブジェクトの順序と，それらが実際に処理される順序との関係を把握しにくい．各配列要素の参照先を確認できても，先頭側と末尾側のどちらから要素が取り出されるかを判断するには，取り出し前後の実行時状態を比較する必要がある．この比較を実行全体にわたって繰り返すことは，利用者の負担となる．

以上のように，従来の Kanon では，ある実行時点における変数の参照先や配列の内容を確認できる一方で，アルゴリズムの進行に関係する参照を多数の参照関係の中から見つけ，その時間的変化を追跡することは難しい．

4 提案手法

本研究では，データ構造の形を構成する参照と，アルゴリズムの進行に伴って一時的に対象を指す参照とを区別し，後者をノードリンク図上で強調するとともに，その時系列変化をアニメーションとして提示する手法を提案する．そこで，4.1 節で，参照に基づいた

可視化アルゴリズムの開発においては，参照関係は 2 種類に分類できることを議論する．そして，4.2 節でその分類に基づいて，データ構造と参照の表示方法を改善し，それらの変化をアニメーションとして提示することを提案する．

4.1 形状的参照と流動的参照

データ構造を扱うプログラム中にはデータ構造中の要素を参照するデータが出現するが、それらは (1) データ構造中の要素から要素を参照しているもの (以降、**形状的参照**と呼ぶ) と (2) データ構造の外から、データ構造中の要素を参照しているもの (以降、**流動的参照**と呼ぶ) に分類できる。

図 3 に，形状的参照と流動的参照に分類できることを例で示す．上段は，二分木を深さ優先順に走査する様子を模式的に示したものである．各ノードから左または右の子ノードへの参照は，二分木を構成するノード間の親子関係を表すため，形状的参照に該当する．走査中には，各時点において一つのノードが訪問対象となり，アルゴリズムの進行に伴って次のノードへ移る．このように，現在の訪問対象を示す参照は流動的参照に該当する．

下段は，左端のノードを出発点，右端のノードを目的地とする最短経路探索を模式的に示したものである．この探索では，探索を終えたノードと，今後探索する候補となるノードとを区別する．図中の赤色の領域は，その時点までに探索を終えたノードを表し，そのうち強調された一つのノードが現在の訪問対象である．一方，青色の領域は，今後訪問する候補となるノードを表す．なお，この図では簡単のため，候補となるノードの処理順序を省略している．

ノード間の黒い矢印は，グラフを構成するノード間の接続関係を表すため，形状的参照に該当する．一方，現在の訪問対象を示す参照と，今後訪問する候補を示す参照関係は，探索の進行に伴って変化するため，流動的参照に該当する．

前節で述べた動機は，ここで導入した形状的参照と流動的参照を用いて次のように整理できる．

> 従来の Kanon では，ある実行時点における形状的参照と流動的参照が，同じノードリンク図上に矢印として表示される．フィールド参照と変数参照は矢印の色によって区別されるが，形状的参照と流動的参照という役割の違いは表示に反映されない．そのため，利用者は，多数の形状的参照の中から流動的参照を見つけた上で，異なる実行時点を順に表示し，その参照先の変化を比較する必要がある．また，配列に保持された複数の処理候補を示す流動的参照については，各実行時点の参照先を確認できても，処理候補の順序と実際に処理される順序との関係を把握するために，取り出し前後の状態を繰り返し比較

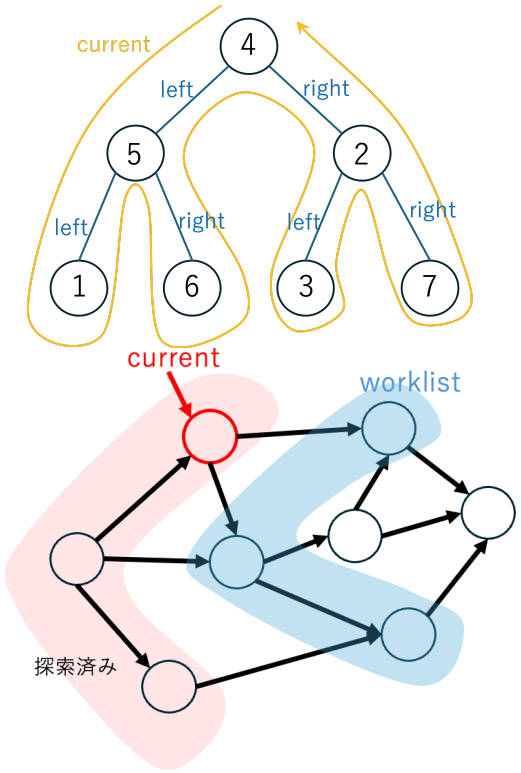


図 3: 形状的参照と流動的参照の違い．上段は二分木走査の例であり，ノード間の親子関係を表す参照が形状的参照，その時点における訪問対象を示す参照が流動的参照である．下段は最短経路探索の例であり，ノード間の接続関係を表す黒い矢印が形状的参照，現在の訪問対象および今後訪問する候補を示す参照関係が流動的参照である．

する必要がある．

4.2 提案手法

そこで本研究では，形状的参照によって表されるデータ構造の形を残しながら，流動的参照を視覚的に区別し，その変化を時系列順に提示する手法を提案する．

本手法では，利用者がプログラム中の流動的変数から追跡したい変数を指定する．システムは，指定された流動的変数を起点とする流動的参照を取得し，その参照に対応するオブジェクトをノードリンク図上で強調する．さらに，流動的参照の変化点に対応する可視化を時系列順に提示する．以下では，流動的参照に基づく強調表示と，アニメーションによる実行過程の提示について順に説明する．

4.2.1 流動的参照に基づく強調表示

本手法では，流動的参照を形状的参照から視覚的に区別し，流動的参照に含まれるオブジェクトとその変化を示すために，(1) 色付け，(2) グラデーション表示，(3) 年輪表示の三つの表示方法を用いる．色付けは，オブジェクトがどの流動的変数に対応しているかを表す．グラデーション表示は，オブジェクトが参照された時刻，または配列内における順序を表す．年輪表示は，同一のオブジェクトに複数の流動的参照が重なっていることを表す．

(1) 色付け

利用者が指定した流動的変数ごとに異なる色を割り当て，その変数を起点とする流動的参照に含まれるオブジェクトに対応するノードを色付けする．流動的変数が単一のオブジェクトを直接参照する場合には，その参照先に対応するノードを色付けする．

流動的変数が配列を参照する場合には，変数から配列オブジェクトへの参照と，配列の各要素からオブジェクトへの参照を流動的参照として取得する．そのうえで，配列の各要素が参照しているオブジェクトに対応するノードを色付けする．これにより，利用者は，形状的参照によって構成されたデータ構造の中から，指定した流動的変数に関係するオブジェクトを確認できる．

(2) グラデーション表示

グラデーション表示では，同じ流動的変数に対応する色の濃淡によって，流動的参照に関する順序を示す．本手法では，参照時刻に基づく表示と，配列内のインデックスに基づく表示の二つを用意し，利用者が目的に応じて切り替えられるようにする．一つの表示において，色の濃淡が参照時刻とインデックスの両方を同時に表すことはない．

参照時刻に基づく表示では，現在の流動的参照に含まれているオブジェクトを最も濃く表示する．過去に流動的参照に含まれていたオブジェクトについては，最後に含まれていた時刻が古いものほど薄く表示する．これにより，一つのノードリンク図上で，流動的参照に含まれるオブジェクトがどのように変化してきたかを確認できる．

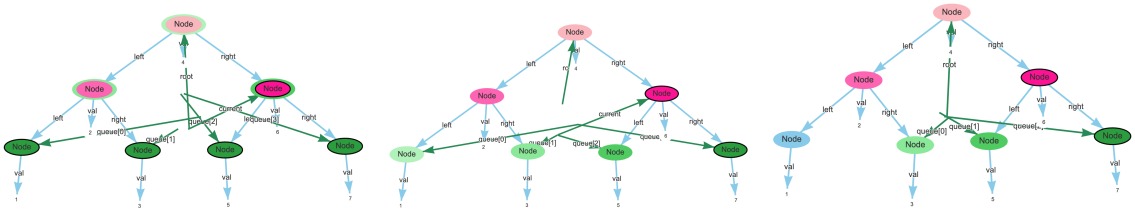
配列内のインデックスに基づく表示は，配列を参照する流動的変数に対して用いる．この表示では，配列の各要素が参照するオブジェクトを，その要素のインデックスに応じた濃淡で表示する．本手法では，インデックスが小さい要素を薄く，インデックスが大きい要素を濃く表示する．これにより，配列に保持されているオブジェクトだけでなく，それらが配列内でどのような順序に並んでいるかを確認できる．

ただし，一時点のインデックスに基づく表示だけでは，配列の先頭側と末尾側のどちらから要素が取り出されるかを判断できない．そこで，インデックスに基づく表示を，後述するアニメーションと組み合わせる．

(3) 年輪表示

同一のオブジェクトが，複数の流動的参照に現在または過去に含まれている場合には，一つの色だけでは各流動的参照を区別できない．そこで本手法では，流動的変数ごとに割り当てた色のリングを，同一のノード上に同心円状に重ねて表示する．各リングの色は対応する流動的変数を表し，その濃淡は参照時刻に基づくグラデーションと同じ規則に従う．これにより，同一のオブジェクトが複数の流動的参照にどのように関係しているかを確認できる．

図 4 に，三つの表示方法を適用した例を示す．いずれの図でも，流動的変数 `current` に対応する流動



(a) 参照時刻に基づく時系列表示　(b) 配列内のインデックス順に基づく表示　(c) 図 4b の次の変化点の可視化

図 4: 流動的参照の表示方法．流動的変数 current に対応する参照をピンク色，queue に対応する参照を緑色で示す．

的参照をピンク色，queue に対応する流動的参照を緑色で示している．

図 4a は，参照時刻に基づくグラデーション表示の例である．current の現在の参照先は最も濃いピンク色で表示されており，現在は val フィールドの値が 6 であるオブジェクトを参照していることが分かる．また，ピンク色の濃淡から，current が過去に参照していたオブジェクトを確認できる．値が 6 であるオブジェクトにはピンク色と緑色の表示が重なっており，このオブジェクトが現在は current に参照されている一方で，過去には queue に保持されていたことを年輪表示から確認できる．

図 4b は，配列内のインデックス順に基づく表示である．この図から，queue には，インデックスの小さい順に，val フィールドの値が 1，3，5，7 であるノードが格納されていることを確認できる．なお，インデックスの小さい方を薄く，大きい方を濃く表示している．

4.2.2　アニメーションによる実行過程の提示

前節で述べた強調表示により，一つの実行時状態における流動的参照の現在の状態や，直近の履歴を確認できる．しかし，一つのノードリンク図だけでは，流動的参照がどのような順序で変化したかを確認することはできない．そこで本手法では，流動的参照の変化に対応するノードリンク図を，アニメーションとして時系列順に提示する．

なお，本研究では，流動的参照が変化した時点を変化点と呼ぶ．単一のオブジェクトを直接参照する流動的参照では，参照先のオブジェクトが変化した時点が変化点となる．配列を介して複数のオブジェクトを参照する流動的参照では，配列に保持されたオブジェクトまたはその順序が変化した時点が変化点となる．

アニメーションでは，各変化点における色付け，グラデーション表示，および年輪表示を順に提示する．これにより，利用者は，current の参照先が変化する順序や，queue に保持された処理待ちのオブジェクトとその順序の変化を観察できる．

配列内のインデックスに基づくグラデーション表示とアニメーションを組み合わせることで，配列のどの位置からオブジェクトが取り出されたかを確認できる．図 4b では，queue に，val フィールドの値が 1，3，5，7 であるオブジェクトが，この順序で保持されている．このうち，値が 1 であるオブジェクトは，配列の先頭である queue[0] から参照されている．

次の変化点を示す図 4c では，値が 1 であるオブジェクトが Shift によって queue の先頭から取り出され，値が 3，5，7 であるオブジェクトが残っている．したがって，二つの可視化を続けて観察することで，queue の先頭にあった要素が取り出されたことを確認できる．

5　実装

本研究では，第 4 節で提案した流動的参照の強調表示とアニメーションによる時系列提示を，ライブプログラミング環境 Kanon 上に実装した．本章では，流動的参照の参照履歴を取得する処理，色付け，グラデーション表示および年輪表示による強調表示，ならびに変化点に基づくアニメーションの実装について述

図 5: Maximum node search using iterative breadth-first traversal

```
 1: function FindMaxNode(root)
 2:     if root = null then
 3:         return null
 4:     end if
 5:     queue ← [root]
 6:     current ← null
 7:     maxNode ← root
 8:     while queue is not empty do
 9:         current ← Shift(queue)
10:         if current.val > maxNode.val then
11:             maxNode ← current
12:         end if
13:         if current.left ≠ null then
14:             Push(queue, current.left)
15:         end if
16:         if current.right ≠ null then
17:             Push(queue, current.right)
18:         end if
19:     end while
20:     return maxNode
21: end function
```

べる．

以下では，アルゴリズム 5 に示す，二分木を反復的に走査して最大値を持つオブジェクトを求めるプログラムを例として用いる．このプログラムでは，各反復において配列から取り出したオブジェクトを `current` に代入し，そのオブジェクトの値を現在までの最大値と比較する．以下では，利用者が `current` を流動的変数として指定した場合を例として，流動的参照の取得，強調表示，およびアニメーションの実装を説明する．なお，指定された流動的変数が配列を参照する場合に処理が異なる箇所については，その都度説明する．

5.1 流動的参照履歴の取得

本実装では，追跡対象のメソッド呼び出しに対応するスナップショットから，指定された流動的変数を起点とする流動的参照を取得し，時系列の参照履歴を作成する．選択されたメソッド呼び出しから実行された再帰呼び出しや補助メソッドの呼び出しに対応するスナップショットも，追跡範囲に含める．

流動的変数は，静的スコープと変数名の組によって識別する．例えば，アルゴリズム **??**の 6 行目で初期化される `current` は，`FindMaxNode` の本体と変数名 `current` の組として識別される．別のメソッド内に同名の変数が存在する場合には，異なる追跡対象として扱う．一方，同じメソッドが再帰的に呼び出された場合には，各実行フレームに存在する同名の変数を一つの追跡対象としてまとめる．

`current` が単一のオブジェクトを直接参照するため，各スナップショットから `current` の参照先となるオブジェクトの識別子を取得する．取得した識別子をスナップショットの実行順に並べることで，`current` が参照してきたオブジェクトの履歴を作成する．

一方，指定された流動的変数が配列を参照する場合には，変数から配列オブジェクトへの参照と，配列の各要素からオブジェクトへの参照を取得する．参照履歴には，各要素が参照するオブジェクトの識別子を，配列内の順序を保った要素列として記録する．

5.2 強調表示の実装

前節で作成した参照履歴に含まれるオブジェクトの識別子を，Kanon のノードリンク図上のノードへ対応付ける．以下では，色付け，グラデーション表示，および年輪表示への反映方法を述べる．

流動的変数ごとに色を割り当て，参照履歴に記録されたオブジェクトの識別子と一致するノードへ，対応する色を設定する．流動的変数が配列を参照する場合には，配列の各要素が参照するオブジェクトに対応するノードを描画対象とする．

参照時刻に基づく表示では，表示対象となるスナップショットまでの履歴から，各オブジェクトが流動的参照に最後に含まれていた時点を求め，その時点に基づいて色の濃淡を決定する．配列内のインデックスに

基づく表示では，表示対象のスナップショットに記録された要素列から各オブジェクトのインデックスを取得し，その値に基づいて色の濃淡を決定する．

同じノードに複数の流動的変数による強調表示が対応する場合には，それぞれの色をリングとして描画する．各オブジェクトがそれぞれの流動的参照に最後に含まれていた時点を比較し，より新しい履歴に対応するリングを内側に配置する．リングの濃淡は，参照時刻に基づくグラデーション表示と同じ方法で決定する

5.3 アニメーションの実装

変化点の判定

単一のオブジェクトを直接参照する流動的変数については，隣接するスナップショット間で参照先オブジェクトの識別子を比較する．識別子が異なる場合には，後のスナップショットを変化点として記録する．同じオブジェクトを参照したまま実行が進んだ場合には，変化点として扱わない．

アルゴリズム **??**では，9 行目の代入によって `current` の参照先が更新される．この代入後の参照先が直前のスナップショットと異なる場合に，そのスナップショットが `current` の変化点となる．

指定された流動的変数が配列を参照する場合には，配列要素が参照するオブジェクトの識別子を，配列内の順序を保った要素列として比較する．要素の追加，削除，または並べ替えによって要素列が変化した場合には，後のスナップショットを変化点として記録する．一方，異なる配列オブジェクトが代入された場合でも，配列要素が参照するオブジェクトとその順序が同じであれば，変化点として扱わない．

複数の流動的変数が指定された場合には，各流動的変数について求めた変化点の和集合を，アニメーション全体の変化点とする．

時系列提示

アニメーションでは，記録した変化点に対応するスナップショットを実行順に切り替え，ノードリンク図と強調表示を更新する．スナップショットの切り替えに合わせて，ソースコード上の実行位置とメソッドの呼び出し関係も更新する．

`Play Animation` ボタンは，変化点に対応するスナップショットを時系列順に自動再生する．次実行状態ボタンは，現在のスナップショットに隣接する次の実行時状態へ移動する．次変化点ボタンは，現在位置より後にある最も近い変化点へ移動する．

アニメーション中のノードの移動を抑えるため，最初のスナップショットにおける各オブジェクトの座標を，後続するスナップショットにも適用する．同じ識別子を持つオブジェクトを同じ座標に配置することで，すべてのスナップショットにおいてノードの配置を固定する．

6 ケーススタディを用いた評価

本章では，提案手法を適用した Kanon により，流動的参照の遷移が可視化上に現れることをケーススタディによって確認する．本評価の目的は，利用者が指定した流動的変数に対応する流動的参照について，正常な実行過程における参照先の遷移と，誤りを含む実行過程における異常な遷移が視認できることを確認することである．そのため，本評価ではユーザ実験ではなく，複数の実装に対して提案手法を適用し，どのような情報を可視化から読み取れるかを確認するケーススタディを行った．

6.1 実験設定

対象として，二分木から指定した値を探索するプログラムを用いた．本評価では，利用者が書きたい正しいプログラムを DFS 探索プログラムとし，同じ入力に対して同じ結果を返す 12 種類の探索プログラムを用意した．具体的には，正しい DFS 実装として，再帰 DFS，stack を用いる反復 DFS，結果ノードを保持する DFS，コンパクトな再帰実装の四種類を用意した．また，誤りを含む実装として，本来 DFS として書くべきところを BFS として実装してしまった四種類と，左の子または左部分木を誤って 2 回探索対象に加える誤りを混入させた四種類を用意した．

各実装に対して，アルゴリズムの進行を表す変数を流動的参照として指定した．再帰実装では現在の引数ノードを表す `node` または `root` を指定し，反復実装では現在処理中のノードを表す `current` と，探索候補を保持する `stack` または `queue` を指定した．

結果ノードを保持する実装では，現在処理中のノードを表す current を指定した．

データ構造を走査するアルゴリズムを記述またはデバッグする際，利用者は,「探索が意図した順序で進んでいるか」，また,「候補集合が意図した規則に従って処理されているか」を確認する必要がある．しかし，従来の Kanon による一時点の可視化だけでは，これらを読み取ることは難しい．

そこで本評価では，利用者が意図した探索順序や候補集合の処理規則が，提案手法の可視化上にどのような流動的参照の遷移として現れるかを確認する．第一に，単一のオブジェクトを指す流動的参照の遷移から，現在処理しているノードと探索の進行を確認できることとする．この観点には，正常な実装における探索順序だけでなく，利用者が意図した DFS とは異なる BFS 的な進行や，同一のノードまたは部分木が重複して処理される様子を確認できることも含む．第二に，複数のオブジェクトからなる候補集合を表す stack または queue の遷移から，候補集合への要素の追加順序と取り出し順序を確認できることとする．

表 1 に，各ケースで指定した流動的参照と，これらの観点に対する確認結果を示す．表中の ○ は可視化から確認できたこと，× は可視化から確認しにくかったこと，– はその実装では該当する状態または誤りが存在しないことを表す．

6.2 結果

正しい DFS 実装では，4 種類全てで指定した流動的参照の遷移を可視化できることを確認した．再帰 DFS では，node の参照履歴から，現在処理中のノードが左部分木から右部分木へ遷移する様子を確認できた．stack DFS では，current に加えて stack を指定することで，探索候補の変化を確認できた．結果保持 DFS では，current と result を対応付けることで，探索中のノードと発見されたノードを区別して表示できた．コンパクトな再帰実装では，root の参照履歴から探索順序は確認できたが，候補集合や結果ノードに対応する変数が存在しないため，可視化される情報は引数ノードの履歴に限定された．

BFS 誤実装では，再帰 DFS, キューを使った BFS, 結果保持 DFS では，利用者が意図した DFS とは異なる探索順序が可視化上に現れることを確認した．current と queue を持つ実装では，同じ深さのノードを順に処理する様子が確認でき，本来 DFS として実装したかったプログラムが BFS 的な実行過程となっていることを認識できた．

一方，短い BFS 実装では，実装上のバグにより queue の履歴を意図した形で取得できなかった．その結果，複数時点の queue[0] が重なって表示され，アニメーションで提示される変化点も少なかったため，可視化のみからは訪問順序や候補集合の変化を十分に読み取ることはできなかった．

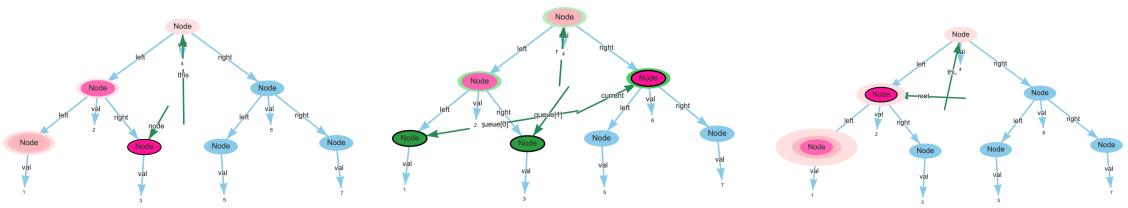
重複誤実装では，4 種類全てで流動的参照の遷移から，同一ノードまたは同一部分木が重複して処理される様子を確認した．再帰 DFS では，同じ部分木が繰り返し探索される様子を確認できた．stack DFS では，同じノードが stack に複数回追加され，その後複数回 current となる様子を確認できた．結果保持 DFS では，重複探索の様子に加え，最終的に発見されたノードが result として保持されることを確認できた．コンパクトな再帰実装でも，root の参照履歴から同一部分木の重複探索を確認できた．

図 6 に，再帰で書いた正しい DFS，BFS 誤実装，および重複誤実装に対して提案手法を適用した可視化例を示す．図 6a では，node に対応する現在の参照先が濃い ピンク，過去の参照先が薄い ピンク で表示されており，探索が左部分木内を進んだことを確認できる．図 6b では，current の参照先が ピンク，queue に含まれる候補ノードが 緑 で表示されており，同じ深さのノードを候補として保持しながら探索する BFS 的な実行過程を確認できる．図 6c では，root の参照履歴が同一部分木上に重なって表示されており，同じ部分木が複数回探索されたことを確認できる．

以上の結果から，単一のオブジェクトを参照する流動的参照の遷移を強調することで，現在処理中のノードと探索の進行を確認できることを示した．これにより，正しい DFS の探索順序に加え，BFS 誤実装における意図と異なる探索順序や，重複誤実装における同一ノードまたは同一部分木の重複処理も認識でき

表 1: 探索プログラムに対するケーススタディ

No.	種別	実装	指定した流動的参照	探索の進行	候補集合	主な確認内容
1	正しい DFS	再帰 DFS	`node`	○	–	現在ノードが左部分木から右部分木へ遷移する様子
2	正しい DFS	stack DFS	`current`, `stack`	○	○	候補集合内のノードが次の処理対象へ遷移する様子
3	正しい DFS	結果保持 DFS	`current`	○	–	探索中のノードと発見されたノードの区別
4	正しい DFS	短い再帰実装	`root`	○	–	引数として渡されるノードの遷移
5	BFS 誤実装	再帰的 BFS	`current`, `queue`	○	○	同じ深さのノードを先に処理する BFS 的な進行
6	BFS 誤実装	queue BFS	`current`, `queue`	○	○	候補集合内のノードが FIFO 順に処理される様子
7	BFS 誤実装	結果保持 BFS	`current`, `queue`	○	○	BFS 的な候補処理と発見されたノードの保持
8	BFS 誤実装	短い BFS 実装	`queue`	×	×	実装上のバグにより，訪問順序と候補集合の変化を正しく可視化できない
9	重複誤実装	再帰 DFS	`root`	○	–	同一部分木が複数回探索される様子
10	重複誤実装	stack DFS	`current`, `stack`	○	○	候補集合への重複追加と同一ノードの重複処理
11	重複誤実装	結果保持 DFS	`current`	○	–	同一部分木の重複探索と発見されたノードの保持
12	重複誤実装	短い再帰実装	`root`	○	–	同一部分木が複数回探索される様子



(a) 正しい再帰 DFS　(b) BFS 誤実装　(c) 重複誤実装

図 6: ケーススタディにおける可視化例．(a) では，`node` の現在の参照先と参照履歴から，正しい DFS の探索過程を確認できる．(b) では，`current` と `queue` の強調表示から，DFS とは異なる BFS 的な探索過程を確認できる．(c) では，`root` の参照履歴から，同一部分木が重複して探索されたことを確認できる．

た．また，`stack`，`queue` を指定した実装では，探索候補の変化や発見されたノードを確認できることを示した．

一方，短い BFS 実装では，実装上のバグにより `queue` の履歴を正しく取得できず，可視化から探索の進行と候補集合の変化を十分に確認できなかった．

また，実装上のバグにより評価対象としなかった短い BFS 実装を除く 11 種類の実装を通して，条件式の成立，返り値，`null` 参照，および再帰呼び出しからの戻りなど，参照先の変化として直接表れにくい振る舞いは，可視化のみからは読み取りにくいことを確認した．これは，本手法が流動的参照の参照先および候補集合の変化を可視化対象としているためである．これらの制御上の情報を，流動的参照の履歴やソース

コード上の実行位置と組み合わせて提示することは，プログラム理解をさらに支援するための今後の課題である．

7 関連研究

プログラムの実行時情報を可視化し，プログラム理解やデバッグを支援する研究は数多く行われている．Jeliot 3 [9] や Python Tutor [3] は，プログラムの実行過程における実行位置や変数値を段階的に提示することで，利用者がプログラムの実行の進み方を追跡できるようにする．Theseus [7] は，JavaScript エディタ上で関数の実行回数や実行時の値を提示することで，ソースコードと実行時挙動の対応付けを支援する．Omnicode [6] は，live programming 環境において，プログラム中の変数値の変化を常時提示することで，実行中の値の推移を把握できるようにする．In Situ Visualization [5] は，ソースコード上に実行時情報に基づく可視化を埋め込むことで，コードと実行時状態の対応付けを支援する．これらの研究は，実行時情報を利用者に提示することでプログラム理解を支援する点で本研究と関連する．一方で，本研究は，値や実行回数を提示するだけでなく，オブジェクト構造や参照関係の可視化上で，利用者が注目する変数の参照先や候補集合がどのように変化したかを強調する点が異なる．

データ構造やオブジェクト構造を可視化する研究も，本研究と関連する．Dynamic Object Viewers [2] は，データ構造を通常のデバッガ表示ではなく，対象に応じた視覚表現として提示する．Heapviz [1] は，Java プログラムの実行中に得られるヒープスナップショットをグラフとして可視化し，オブジェクト間の参照関係の理解やデバッグを支援する．また，BugVis [8] は，データ構造可視化と time-travel debugging を組み合わせ，過去の実行状態をさかのぼりながらデータ構造の変化を確認できるようにする．これらの研究は，オブジェクトやデータ構造を視覚的に提示することでプログラム理解やデバッグを支援する点で本研究と近い．一方で，本研究は，データ構造全体の状態や過去の状態を提示するだけでなく，アルゴリズムの進行を理解する上で重要な変数に着目し，その変数が指す対象や候補集合の時間的な変化を強調する点に特徴がある．

8 結論と今後の課題

本研究では，データ構造をノードリンク図によって可視化する場合に、プログラムの実行過程を追跡するための改良方法を提案した。具体的には、(1) プログラム中に出現するオブジェクトへの参照を、データ構造の形を決定する形状的参照と、時間とともに変化する流動的参照とに区別し、前者は (従来通りの) ノード間のリンクとして、後者は参照点のノードへの色付けを行うことと (2) それをアニメーションによって提示することを提案した。これによって、従来は多数のリンクに埋没していた「動く参照」を複数の可視化画面を見比べることで理解されていた実行過程が、動画として容易に視認できることを狙っている。

本研究では，提案手法を Kanon 上で実現し、現在処理している対象，次に処理される候補集合を可視化上で追跡できるため，アルゴリズムの進行や誤った挙動を把握しやすくなることが確認できた．今後の課題として，条件式の成立，返り値，`null` 参照，再帰からの戻りなど，参照先の変化として直接表れにくい振る舞いの情報を併せて提示することは検討が必要だ．

謝辞 本研究を進めるにあたり、プログラミング研究室の皆様、特に叢悠悠助教や吉尾拓真氏にも多くの助言を頂きました．この場を借りて感謝申し上げます．

参 考 文 献

[1] Aftandilian, E. E., Kelley, S., Gramazio, C., Ricci, N., Su, S. L., and Guyer, S. Z.: Heapviz: Interactive Heap Visualization for Program Understanding and Debugging, *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, New York, NY, USA, Association for Computing Machinery, 2010, pp. 53–62.

[2] Cross, II, J. H., Hendrix, T. D., Jain, J., and Barowski, L. A.: Dynamic Object Viewers for Data Structures, *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, New York, NY, USA, Association for Computing Machinery, 2007, pp. 4–8.

[3] Guo, P. J.: Online Python Tutor: Embeddable Web-Based Program Visualization for CS Educa-

tion, *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, New York, NY, USA, Association for Computing Machinery, 2013, pp. 579–584.

[4] Hayatpur, D., Wigdor, D., and Xia, H.: CrossCode: Multi-level Visualization of Program Execution, *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, Association for Computing Machinery, 2023, pp. 593:1–593:13.

[5] Hoffswell, J., Satyanarayan, A., and Heer, J.: Augmenting Code with In Situ Visualizations to Aid Program Understanding, *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, CHI '18, New York, NY, USA, Association for Computing Machinery, 2018.

[6] Kang, H. B. and Guo, P. J.: Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations, *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, New York, NY, USA, Association for Computing Machinery, 2017, pp. 737–745.

[7] Lieber, T., Brandt, J. R., and Miller, R. C.: Addressing Misconceptions About Code with Always-on Programming Visualizations, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, New York, NY, USA, Association for Computing Machinery, 2014, pp. 2481–2490.

[8] Mönch, K. and Minas, M.: BugVis: A Debugger Supporting Time-Travel and Data Structure Visualization, *Proceedings of the 6th European Conference on Software Engineering Education*, ECSEE '25, New York, NY, USA, Association for Computing Machinery, 2025, pp. 195–199.

[9] Moreno, A., Myller, N., Sutinen, E., and Ben-Ari, M.: Visualizing Programs with Jeliot 3, *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, New York, NY, USA, Association for Computing Machinery, 2004, pp. 373–376.

[10] Oka, A., Masuhara, H., and Aotani, T.: Live, Synchronized, and Mental Map Preserving Visualization for Data Structure Programming, *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2018, New York, NY, USA, Association for Computing Machinery, 2018, pp. 72–87.